Herbert Jaeger

# Machine Learning (WMAI010-05)

Lecture Notes V 1.26, Dec 18, 2024

Master Program in Artificial Intelligence Rijksuniversiteit Groningen, Bernoulli Institute

## Contents

1	Introduction					
	1.1	Human Versus Machine Learning	6			
	1.2	The two super challenges of ML - from an eagle's eye	9			
	1.3	Looking at Human Intelligence, Again	18			
	1.4	A Remark on "Modeling"	21			
	1.5	The Machine Learning Landscape	23			
<b>2</b>	Dec	ision trees and random forests	29			
	2.1	A toy decision tree	29			
	2.2	Formalizing "training data"	31			
	2.3	Learning decision trees: setting the stage	34			
	2.4	Learning decision trees: the core algorithm	37			
	2.5	Dealing with overfitting	41			
	2.6	Variants and refinements	42			
	2.7	Random forests	43			
3	Elementary supervised temporal learning					
	3.1	Recap: linear regression	49			
	3.2	Temporal learning tasks	54			
	3.3	Time series prediction tasks	58			
	3.4	Signal-based vs. state-based timeseries modeling	59			
	3.5	Takens' theorem	62			
4	Basic methods for dimension reduction					
	4.1	Set-up, terminology, general remarks	65			
	4.2	K-means clustering	68			
	4.3	Principal component analysis	70			
	4.4	Mathematical properties of PCA and an algorithm to compute PCs	74			
	4.5	Summary of PCA based dimension reduction procedure	76			
	4.6	Eigendigits	76			
	4.7	Self-organizing maps	78			
	4.8	sion reduction	84			
<b>5</b>	Dise	crete symbolic versus continuous real-valued	89			
6	The bias-variance dilemma and how to cope with it					
	6.1	Training and testing errors	95			
	6.2	The menace of overfitting – it's real, it's everywhere	98			
	6.3	An abstract view on supervised learning	103			
	6.4	Tuning model flexibility	105			
	6.5	Finding the right modeling flexibility by cross-validation	113			

	6.6	Why it is called the bias-variance dilemma	. 116				
7	Rep 7.1 7.2 7.3 7.4	epresenting and learning distributions       1         1       Optimal classification       1         2       Representing and learning distributions       1         3       Mixture of Gaussians; maximum-likelihood estimates by EM algorithms       1         4       Parzen windows       1					
8	<b>Bay</b> 8.1 8.2 8.3 8.4	esian model estimation         The ideas behind frequentist statistics         The ideas behind Bayesian statistics         Case study: modeling proteins         A brief summary	<b>150</b> . 150 . 152 . 158 . 162				
9	Sam 9.1 9.2 9.3 9.4 9.5 9.6	pling algorithms         What is "sampling"?         Sampling by transformation from the uniform distribution         Rejection sampling         Proto-distributions         MCMC sampling         Application example: determining evolutionary trees	<b>164</b> . 164 . 166 . 168 . 170 . 172 . 182				
10	10.1 10.2 10.3	Bayesian networks	. 190 . 211 . 211				
11	<b>Onl</b> 11.1 11.2 11.3 11.4	ine adaptive modeling The adaptive linear combiner	<b>213</b> . 214 . 218 . 223 . 236				
12	Feed 12.1 12.2 12.3	Iforward neural networks: the Multilayer Perceptron         MLP structure	<b>242</b> . 245 . 248 . 250				
13	Ope	en doors in $ML$ — leading where?	256				

Α	Elementary mathematical structure-forming operationsA.1Pairs, tuples and indexed familiesA.2Products of setsA.3Products of functions	<b>269</b> 269 270 270
В	Joint, conditional and marginal probabilities	271
С	The argmax operator	277
D	Expectation, variance, covariance, and correlation of numerica random variables	$\frac{1}{277}$
$\mathbf{E}$	Derivation of Equation 32	281

A note on mathematical background that is required. Machine learning programs process "training data", and these data often come in the format of Excel files (yes! down-to-earth engineers and financial admins indeed very often save their valuable "raw data" often in MS Excel!), which are just *matrices* when seen with the mathematical eye of a machine learner. Furthermore, neural networks are shaped by connecting neurons with weighted "synaptic links", and these weights are again naturally sorted in matrices. And the main operation that a neural network actually does is formalized by a matrix-vector multiplication. Besides neural networks, many other (if not most) machine learning systems operate on vector-formatted data. So it's matrices and vectors all over the place, no escape possible. You will need a robust understanding of basic linear algebra to survive or even enjoy this course. We will arrange a linear algebra crash refresher early in the course. A good free online resource is the book "Mathematics for Machine Learning" (Deisenroth, Faisal, and Ong 2019).

Furthermore, though to a lesser degree, also some familiarity with statistics and probability is needed. You find a summary of the must-knows in the appendix of these lecture notes, and again a tutorial exposition in Deisenroth, Faisal, and Ong 2019.

Finally, a little (not much) calculus is needed to top it off. If you are familiar with the notion of partial derivatives, that should do it. In case of doubt - again it's all (and more) in Deisenroth, Faisal, and Ong 2019.

## 1 Introduction

## 1.1 Human Versus Machine Learning

Humans learn. Animals learn. Societies learn. Machines learn. It looks like "learning" were a universal phenomenon and all we had to do is to develop a solid scientific theory of "learning", turn that into algorithms and then let "learning" happen on computers. Wrong wrong wrong. Human learning is very different from animal learning (and amoebas learn different things in different ways than chimpanzees), societal learning is quite another thing as human or animal learning, and machine learning (ML) is as different from any of the former, just as cars are different from horses.

Human learning is as impossible to understand as you are yourself — look into a mirror and think of *all* the things you can *do*, all of your body motions from tying your shoes to playing the guitar; *all* the thoughts you can think from "aaagrhhh!" to "I think therefore I am"; achievements personal, social, academic; *all* the things you can remember including your first kiss and what you did 20 seconds ago (you started reading this paragraph, in case you forgot); your plans for tomorrow and the next 40 years; well, just *everything* about you — almost everything of that wild collection is the result of a fabulous mixing of some kind of learning with other miracles and wonders of life. To fully understand human learning, a scientist would have to integrate *at least* the following fields and phenomena:

body, brain, sensor & motor architecture  $\cdot$  physiology and neurophysiology  $\cdot$  body growth  $\cdot$  brain development  $\cdot$  motion control from eye gaze stabilization to dance choreography  $\cdot$  exploration, curiosity, play  $\cdot$  creativity  $\cdot$  social interaction  $\cdot$  drill and exercise and rote learning  $\cdot$  reward and punishment, pleasure and pain  $\cdot$  the universe, the earth, the atmosphere, water, food, caves  $\cdot$  evolution  $\cdot$  dreaming  $\cdot$  remembering  $\cdot$  forgetting  $\cdot$  aging  $\cdot$  other people, living  $\cdot$  other people, long dead  $\cdot$  machines, tools, buildings, toys  $\cdot$  words and sentences  $\cdot$  concepts and meanings  $\cdot$  letters and books and schools  $\cdot$  traditions  $\ldots$ 

Recent spectacular advances in machine learning may have nurtured the impression that machines come already somewhat close. Specifically, neural networks with many cascaded internal processing stages (so-called *deep* networks) have been trained to solve problems that were considered close to impossible only a few years back. A showcase example (one that got me hooked) is automated image caption (technical report: Kiros, Salakhutdinov, and Zemel 2014). At http://www.cs.toronto.edu/~nitish/nips2014demo you can find stunning examples of caption phrases that have been automatically generated by a neural network based system which was given photographic images as input. Well, while most examples are amazing, some other examples are also stunningly wrong. Fig-

ure 1 shows some screenshots. This is a demo from 2014. Since deep learning is evolving incredibly fast, it's already quite outdated today and current image caption generators come much closer to perfection. But back in 2014 this was a revelation. Other fascinating examples of deep learning are face recognition (Parkhi, Vedaldi, and Zisserman 2015), online text translation (Bahdanau, Cho, and Bengio 2015), inferring a Turing machine (almost) from input-output examples (Graves et al. 2016), or playing the game of Go at and beyond the level of human grand-masters (Silver et al. 2016). I listed these examples when I wrote the first edition of these lecture notes and I am too lazy to update this list every year, — I guess almost all of you have had direct contact with one of these large language models or fabulous image classification / generation tools that are now everywhere.

So, apparently machine learning algorithms come close to human performance in several tasks or even surpass humans, and these performance achievements have been *learnt* by the algorithms, — thus, machines today can learn like humans??!? The answer is NO. ML researchers (that is, the really good ones, not the average TensorFlow user) are highly aware of this. Outside ML however, naive spectators (from the popular press, politics, or other sciences) often conclude that since learning machines can achieve similar feats as humans, they also learn like humans. It takes some effort to argue why this is not so (read Edelman 2015 for a refutation from the perspective of cognitive psychology). I cannot embark on this fascinating discussion here. Very roughly speaking, it's the same story again as with chess-playing algorithms: the best chess programs win against the best human chess players, but not by fair means — chess programs are based on larger amounts of data (recorded chess matches) than humans can memorize, and chess programs can do vastly more computational operations per second than a human can do. Brute force wins over human brains at some point when there is enough data and processing bandwidth. Progress has accelerated in the last years because increasingly large training datasets have become available and fast enough computing systems have become cheap enough.

This is not to say that powerful "deep learning" just means large datasets and fast machines. These conditions are necessary but not sufficient. In addition, also numerous algorithmical refinements and theoretical insights in the area of statistical modeling had to be developed. Some of these algorithmical/theoretical concepts will be presented in this course.

Take-home message: The astonishing learning achievements of today's ML are based on statistical modeling insight, very cleverly refined algorithms, raw processing power, brutally sized training datasets, and a lot of personal experience and trial-and-error optimization. It's technology and maths, not neurobiology or psychology. Dismiss any romantic ideas about ML that you may have had. ML is data technology stuff for sober engineers. But you are allowed to become very excited about that stuff, and that stuff can move mountains.

#### Tags

- bicyclers
- ridersbikers
- equestrians
- bicyclists

#### **Nearest Caption in the Training Dataset**

a couple of people on four horses walking the streets .

#### **Generated Captions**

- a couple of men and horses on a street .
- some people are riding horses on a street.
- a group of people walking on horseback on a street.
   a couple of people with horses walk down the street.
- a group of people riding horses down a street .

#### Tags

- docks
- pontoon
- dockside
- boats
- pontoons

#### Nearest Caption in the Training Dataset

a group of people on a bridge beside a boat .

#### **Generated Captions**

- several people are on a dock in the water .
- a group of people are on a boat in the distance
- some people on a boat in the harbor .
- a group of people are on the dock on a large boat
  some people are on a bridge over a boat.

#### Tags

- ledges
- ledge
- flagstone
- porches
- casters

#### **Nearest Caption in the Training Dataset**

three men are sawing a tree into sections .

#### **Generated Captions**

- the two men are trying to use a tree .
- two men sitting in front of a wood tree .
- four men are working on a concrete structure .
- two men are working on wood .
  a man and a child in a wooden park are sitting on a farm







Figure 1: Three screenshots from the image caption demo at http://www.cs. toronto.edu/~nitish/nips2014demo — a demo from the 'early' years of deep learning where it dawned upon the machine learning community that there was a revolution going on. A 'deep learning' system was trained on some tens of thousands of photos showing everyday scenes. Each photo in the training set came with a few short captions provided by humans. From these training data, the system learnt to generate tags and captions for new photos. The tags and captions on the left were produced by the trained system upon input of the photos at the right.

## 1.2 The two super challenges of ML - from an eagle's eye

In this section I want to explain, on an introductory, pre-mathematical level, that large parts of ML can be understood as the art of estimating probability distributions from data. And that this art faces a double super challenge: the unimaginably *complex geometry* of real-world data distributions, and the extreme *scarcity of information* provided by real-world data. I hope that after reading this section you will believe that machine learning is impossible. Since however GPT-4 exists, where is the trick? ... A cliff hanger, sorry — I want to make you read on (detailed secrets of the solution in Section 6).

### 1.2.1 The superchallenge of complex geometry

In order to highlight the data geometry challenge, I will use the demo task highlighted in Figure 1 as an example — an image caption generating system developed in Toronto. Let us follow the sacred tradition of ML and introduce an acronym that is not understandable to outsiders: TICS = the Toronto Image Caption System.

The TICS dataset and learning task and demo implementation come from one of the three labs that have pioneered the field that is now known as *Deep Learning* (DL, our next acronym), namely Geoffrey Hinton's lab at the University of Toronto (the other two widely acclaimed DL pioneers are Yoshua Bengio, Université de Montréal, and Yann LeCun, Courant Institute of Mathematics, New York; Hinton, Bengio and LeCun received the 2018 Turing award – the Nobel prize in computer science). The TICS demo is a beautiful case to illustrate some fundamentals of machine learning.

After training, TICS, at first sight, implements a *function*: test image in, caption (and tags) out. If one looks closer, the output however is not just a single caption phrase, but a list of several captions ("generated captions" in Figure 1) rank-ordered by probability: captions that TICS thinks are more probable are placed higher in the list. Indeed TICS computes a relative probability value for each suggested caption. This probability value is not shown in the screenshots.

Let us look at this in a little more detail.

TICS captions are based on a finite vocabulary of English words. For simplicity let us assume that the length of captions that TICS generates is always 10 words. The neural networks that purr under the hood of TICS cannot process "words" – the only data type that they can handle is real-valued vectors. One of the innovations which boosted DL is a method to represent words by vectors such that *semantic similarity* of words is captured by *metric closeness* of vectors: the vector representing the word "house" lies close to the vector for "building" but far away from the vector for "mouse". A landmark paper where you can learn more about this technique is Mikolov et al. 2013, an accessible tutorial is Minnaar 2015, and the current standard algorithm for computing these *word embedding vectors*, the BERT algorithm, is documented in Devlin et al. 2018. A typical size for such semantic word vectors is a dimension of a few 100's – let's say, TICS uses 300dimensional word vectors (I am too lazy to check out the exact dimension that was used). Thus, a caption can be represented by a sequence of ten 300-dimensional vectors, which boils down to a single 3000-dimensional vector, that is, a point  $c \in \mathbb{R}^{3000}$ .

Similarly, an input picture sized  $600 \times 800$  pixels with 3 color channels is represented to TICS as a vector u of dimension  $3 \times 600 \times 800 = 1,440,000$ .

Now TICS generates, upon a presentation of an input picture vector u, a list of what it believes to be the five most probable captions for this input. That is, TICS must have an idea of the probability ratios of different caption candidates. Formally, if C denotes the random variable (RV) that returns captions c, and if Udenotes the RV that produces sample input pictures u, TICS must compute ratios of conditional probabilities of the form  $P(C = c_i | U = u)/P(C = c_j | U = u)$ . In the semantic word vector space, these ratios become ratios of the values of probability density functions (pdf) over the caption vector space  $\mathbb{R}^{3000}$ . For every input image u, TICS must have some representation of the 3000-dimensional pdf describing the probabilities of caption candidates for image u.

Now follow me on a little excursion into geometric imagination.

Consider some specific vector  $c \in \mathbb{R}^{3000}$  which represents a plausible 10-word caption for image u, that is, the pdf value p(c) is relatively large. What happens to the pdf value if we move away from c by taking a little step  $\delta \in \mathbb{R}^{3000}$  of length, say,  $\|\delta\| = 0.1$ , that is, how does  $p(c + \delta)$  compare to p(c)? This depends on the direction in which  $\delta$  points. In a few directions,  $p(c + \delta)$  will be about as large as p(c). This happens when  $\delta$  points toward another caption vector c'which has one word replaced by another word that is semantically close. For example, consider the caption "A group of people on a bridge beside a boat". The last 300 elements in the 3000-dimensional vector c coding this caption stand for the word boat. Replacing this caption by "a group of people on a bridge beside a ship" gives another codevector  $c' \in \mathbb{R}^{3000}$  which is the same as c except in the last 300 components, which have been replaced by the semantic word vector for ship. Then, if  $\delta$  points from c toward c' (that is,  $\delta$  is a fraction of c' - c),  $p(c + \delta)$  will not differ from p(c) in a major way.

If you think a little about it, you will come to the conclusion that such  $\delta$  which leave  $p(c+\delta)$  roughly at the same level are always connected with replacing words by semantically related words. Other change directions  $\delta^*$  will make no semantical sense and/or destroy the grammatical structure of the caption phrase encoded by c, as for instance in these two one-word-replacement examples "A group of people garlic a bridge beside a boat" and "A group of of on a bridge beside a boat". The pdf value  $p(c + \delta^*)$  will drop dramatically compared to p(c) in those cases. Obviously, there are vastly more sense/grammar destructive word replacements than there are sense-and-grammar maintaining ones.

Now, in a 10-word caption, how many replacements of some word with a related one exist? Some words will be grammatical function words ("a", "'of" etc.) which

admit only a small number of replacements, or none at all. The words that carry semantic meaning ("group", "people" etc.) typically allow for a few sense-making replacements. Let us be generous and assume that a word in a 10-word caption, on average, can be replaced by 5 alternative words such that after the replacement, the new caption still is a reasonable description of the input image.

This means that around c there will be  $5 \cdot 10 = 50$  directions in which the relatively large value of p(c) stays large. Assuming these 50 directions are given by linearly independent  $\delta$  vectors, we find that around c there is a 50-dimensional affine linear subspace S of  $\mathbb{R}^{3000}$  in which we can find high p values in the vicinity of c, while in the 2950 directions orthogonal to S, the value of p will drop fast if one moves away from c.

Ok., this was a long journey to a single geometric finding: locally around some point c where the pdf is relatively large, the pdf will stay relatively large only a small fraction of the directions - these directions span a low-dimensional linear subspace around c. If you move a little further away from c on this low-dimensional "sheets", following the lead of high pdf values, you will find that this high-probability surface will take you on a curved path - these high-probability sheets in  $\mathbb{R}^{3000}$  are not flat but curved.

The mathematical abstraction of such relatively high-probability, low-dimensional, curved sheets embedded in  $\mathbb{R}^{3000}$  is the concept of a *manifold*. Machine learning professionals often speak of the "data manifold" in a general way, in order to indicate that the geometry of high-probability areas of real-world pdfs consists of "thin" (low-dimensional), curved sheet-like domains curled into the embedding data space. It is good for ML students to know the clean mathematical definition of a manifold. Although the geometry of real-world pdfs will be less clean than this mathematical concept, it provides useful intuitions, and advanced research in DL algorithms frequently starts off from considering manifold models.

So, here is a brief intro to the mathematical concept of a manifold. Consider an *n*-dimensional real vector space  $\mathbb{R}^n$  (for the TICS output space of caption encodings this would be n = 3000). Let  $m \leq n$  be a positive integer not larger than *n*. An *m*-dimensional manifold  $\mathcal{M}$  is a subset of  $\mathbb{R}^n$  which locally can be smoothly mapped to  $\mathbb{R}^m$ , that is, at each point of  $\mathcal{M}$  one can smoothly map a neighborhood of that point to a neighborhood of the origin in the *m*-dimensional Euclidean coordinate system (Figure 2A).

1-dimensional manifolds are just lines embedded in some higher-dimensional  $\mathbb{R}^n$  (Figure 2B), 2-dimensional manifolds are surfaces, etc. Manifolds can be wildly curved, knotted (as in Figure 2C), or fragmented (as in Figure 2B). Humans cannot visually imagine manifolds of dimension greater than 2.

The "data manifold" of a real-world data source is not a uniquely or even well-defined thing. For instance, returning to the TICS example, the dimension of the manifold around a caption c would depend on an arbitrary threshold fixed by the researcher – a direction  $\delta$  would / would not lead out of the manifold if the pdf decreases in that direction faster than this threshold. Also (again using



Figure 2: A An *m*-dimensional manifold can be locally "charted" by a bijective mapping to a neighborhood of the origin in  $\mathbb{R}^m$ . Example shows a curved manifold of dimension m = 2 embedded in  $\mathbb{R}^3$ . B Some examples of 1-dimensional manifolds embedded in  $\mathbb{R}^2$  (each color corresponds to one manifold — manifolds need not be connected). C A more wildly curved 2-dimensional manifold in  $\mathbb{R}^3$  (the manifold is the surface of this strange body).

the caption scenario) around some good captions c the number of "good" local change directions will differ from the number around another, equally good, but differently structured caption c'. For these and other reasons, claiming that data distributions are shaped like manifolds is a strongly simplifying abstraction.

Despite the fact that this abstraction does not completely capture the geometric complexity of how real-world data points are distributed in their data spaces, the geometric intuitions behind the manifold concept have led to substantial insight into the (mal-)functioning of machine learning methods. *Adversarial attacks* on deep networks is a good example of the usefulness of the data manifold concept. Take a look at Figure 3 – taken from a much-cited paper (I. J. Goodfellow, Shlens, and Szegedy 2014, 13500 cites on Google Scholar 2022) which explores the phenomenon of adversarial attacks. The left panel shows a photo of a panda (one would think). It is given as input to a deep network that had been trained to classify images. The network correctly classifies it as "panda", with a "confidence" of 58%. The middle panel shows a pattern that looks like random noise. If a very small multiple (factor 0.007) of this pattern is added to the panda picture, one gets the picture shown in the right panel. For the human eye there is no change. However, the neural network now classifies it as "gibbon" with a dead-sure confidence level of 99%.



Figure 3: Manifold magic: Turning a panda into a gibbon. For explanation see text. Picture taken from I. J. Goodfellow, Shlens, and Szegedy 2014

What has happened here? Well, this is a manifold story. Let's say that the panda image is sized  $600 \times 600$  pixels (I didn't check) with three color channels. Thus, mathematically, such images are points in  $\mathbb{R}^{1080000}$ . When the neural network was trained, it was presented with a large number of example images, that is, a point cloud in  $\mathbb{R}^{1080000}$ . The outcome of the learning algorithm is an estimate (up to an undetermined scaling factor) of a pdf in  $\mathbb{R}^{1080000}$ . Geometrically speaking, this pdf is highly concentrated along a low-dimensional manifold, call it  $\mathcal{M}$ . The dimension of this manifold is given by the number of neurons in the most narrow layer of the neural network, say m = 1000 (this would be a standard order of magnitude in such deep networks; I didn't check). Thus, the original panda picture corresponds to a point u on a 1000-dimensional manifold which is curled into a 1,080,000-dimensional embedding space. In terms of dimensions, the manifold uses only about one out of thousand dimensions! Now, add that noise-like image (call it  $\delta$ ) to u, getting the rightmost panel picture as  $u + \delta$ . If  $\delta$  is prepared in a way that it points in a direction orthogonal to  $\mathcal{M}$ , the value of the pdf will shrink dramatically. The trained network, when presented with input  $u + \delta$ , will "think" that it has never seen anything like this  $u + \delta$  image, and will return a random classification – "gibbon" in this case. In fact, the "noise" pattern in Figure 3 isn't random noise at all; it had been cleverly computed to lead the neural network state away from the trained manifold  $\mathcal{M}$ .

Adversarial examples are today widely used in a number of different ways to improve the quality of deep learning applications. Check out Section 7.13 in the bible of deep learning (I. Goodfellow, Bengio, and Courville 2016) for a primer if you are interested.

Back to our TICS example. At the first stage in their neural processing pipeline, the designers of this learning system set up a layered neural network whose input layer (the "retina" of the network) had 1,440,000 neurons (one neuron per color pixel), and whose output layer had 4096 neurons. This means that the 1,440,000 dimensional raw input image vectors were projected on a manifold that had at most 4096 dimensions (or less, if another layer in this network had fewer neurons, like the 1000 that we assumed above). The right "folding" of this manifold was effected by the neural network training.

Takehome summary: real-world data distributions are typically concentrated in very thin (low-dimensional, compared to the dimension of the embedding space), badly curled-up sheets in the embedding (high-dimensional) data space  $\mathbb{R}^n$ . Machine learning models, such as the TICS system, must contain some kind of model of this distribution. Such models will feature a concentration of probability mass in very thin, curled sheets. A *universal challenge* for all but the most elementary machine learning methods is to develop modeling techniques that can represent such complex, manifold-like distribution geometries, and "learn" to align the model distribution with the data distribution. A frequently found keyword that points to this bundle of ideas and challenges is "data manifold". One of the reasons for the breakthrough powers of *deep learning* methods is that the "deep" neural networks used there are able to model extremely curled-up data manifolds.

### 1.2.2 The superchallenge of lack of information

I continue to use the TICS system to illustrate this second challenge. What follows will again require an effort in geometrical thinking, but that's the nature of vector data and the information contained in them.

The data used for training TICS contained about 30,000 photos taken from Flickr, each of which was annotated by humans with 5 captions. Figure 4 shows two training examples. Each photo was sized  $600 \times 800$  pixels, with three color channels, making a total of 1,440,000 numbers as we have already stated. We will consider the intensity values as normalized to the range [0, 1], which makes each photo a vector in the unit hypercube  $[0, 1]^{1,440,000}$ . Assuming as before that captions can be coded as 3000-dimensional vectors, which we will also take as normalized to a numeric range of [0, 1], a photo-caption pair makes a vector in  $[0, 1]^{1,440,000+3,000} = [0, 1]^{1,443,000}$ . In order to simplify our discussion we assume that each training image came with a single caption only. Thus, the training data consisted of 30,000 points in the 1,443,000-dimensional unit hypercube.

It is impossible for humans to visualize, imagine or even hallucinate the wild ways of how points can be spatially distributed in a 1,443,000-dimensional vector space. In Figure 5 I attempt a visualization of the TICS data scenario in a 3-dimensional projection of this 1,443,000-dimensional space – 3 being the largest spatial dimension that our brains can handle. The rendering in Figure 5 corresponds to a dataset where each "photo" is made of merely two grayscale pixels. Each such "photo" is thus a point in the square  $[0, 1]^2$  (light blue area in the figure, spanned by the two pixel intensities  $x_1, x_2$ ). Each caption is coded by a single number  $y \in [0, 1]$ . We furthermore simplify the situation by assuming that the training dataset contains only two photos  $u_1, u_2$ . The caption coding vector of a photo is here reduced to a single number, plotted above the photo's coordinates





Gray haired man in black suit and yellow tie working in a financial environment. A graying man in a suit is perplexed at a business meeting.

- A businessman in a yellow tie gives a frustrated look.
- A man in a yellow tie is rubbing the back of his neck.
- A man with a yellow tie looks concerned.



A butcher cutting an animal to sell.

- A green-shirted man with a butcher's apron uses a knife to carve out the hanging carcass of a cow.
- A man at work, butchering a cow.
- A man in a green t-shirt and long tan apron hacks apart the carcass of a cow

while another man hoses away the blood. Two men work in a butcher shop; one cuts the meat from a butchered cow, while the other hoses the floor.

Figure 4: Two images and their annotations from the training dataset. Taken from Young et al. 2014.

on the y-axis (blue crosses). The two blue crosses in the figure thus represent the information contained in the training data.

Now consider a new test image  $u^*$  (orange diamond in the figure). The TICS system must determine a suitable caption for  $u^*$ , that is, an appropriate value  $y^*$ - a point somewhere on the orange broken ??-line in the figure.

But all that TICS knows about captions is contained in the two training data points (blue crosses). If you think about it, there seems to be *no way* for TICS to infer from these two points where  $y^*$  should lie. Any placement along the ??-line is logically possible!

In order to determine a caption position along the ??-line, TICS must add some optimization criterion to the scenario. For instance, one could require one of the following conditions:

- 1. Make  $y^*$  that point on the ??-line which has the smallest total distance to all the training points.
- 2. One might wish to grant closer-by training images a bigger impact on the caption determination than further-away training images. Thus, if  $d(u_i, u^*)$ denotes the distance between training image  $u_i$  and test image  $u^*$ , set  $y^*$ to the weighted mean of the training captions  $y_i$ , where the weights are inversely proportional to the distance of the respective images:

$$y^* = \frac{1}{\sum_i d(u_i, u^*)^{-a}} \sum_i d(u_i, u^*)^{-a} y_i.$$

The parameter a modulates how the impact of further-away images decays with distance.

- 3. Place a spherical 3-dimensional Gaussian with variance  $\sigma^2$  around every blue cross. Determine  $p(y^*)$  to be that point on the ??-line where the sum of these Gaussians is maximal.
- 4. Consider all smoothly curved line segments L in the 3-dimensional data cube in Figure 5 which connect the blue crosses and cut through the ??-line.



Figure 5: Training data for TICS (highly simplified). The photo dataspace (light blue square spanned by pixel intensities  $x_1, x_2$ ) is here reduced to 2 dimensions, and the caption dataspace to a single one only (y-axis). The training dataset is assumed to contain two photos only (blue diamonds  $u_1, u_2$ ), each with one caption (blue crosses with y-values  $y_1, y_2$ ). A test image  $u^*$  (orange diamond) must be associated by TICS, after it has been trained, with suitable caption which lies somewhere in the y-direction above  $u^*$  (dashed orange ??-line).

For any such L, consider the integral  $\gamma(L)$  of absolute curvature along L. Find that curved line segment  $L_{opt}$  which minimizes  $\gamma(L)$ . Then declare the crossing of this  $L_{opt}$  with the ??-line to be  $y^*$ . – Like in the second proposal above, one would also want to emphasize the impact of close-by images, which would lead to a weighted curvature integral.

I collect some observations from this list:

- All of these optimization criteria make some intuitive sense, but they would lead to different results. The generated captions will differ depending on which criterion is chosen.
- The criteria rest on quite different intuitions. It is unclear which one would be better than another one, and on what grounds one would compare their relative merits. A note in passing: finding comparison criteria to assess relative merits of different statistical estimation procedures is a task that

constitutes an entire, important branch of statistics. For an introduction you might take a look at my lecture notes on "Principles of statistical modeling", Section 19.3.1 "Comparing statistical procedures" (https://www.ai.rug. nl/minds/uploads/LN\_PSM.pdf).

- Criteria 2 and 3 require the choice of design parameters  $(a, \sigma^2)$ . The other criteria could be upgraded to include reasonable design parameters too. It is absolutely typical for machine learning algorithms to have such *hyperparameters* which must be set by the experimenter. The effects of these hyperparameters can be very strong, determining whether the final solution is brilliant or useless.
- Turning a mathematical optimization criterion into a working algorithm may be challenging. For criteria 1, 2 and 3 in the list, this would be relatively easy. Turning criterion 4 into an algorithm looks fearsome.
- Each of the criteria leads to implicit constraints on the geometric shape of learnable data manifolds.

Some methods in ML are cleanly derived from optimization criteria, with mathematical theory backing up the design of algorithms. An example of such wellunderstood methods are *decision trees*, which will be presented first in our course. Other branches employ learning algorithms that are so complex that one loses control over what is actually optimized, and one has only little insight in the geometrical and statistical properties of the models that are delivered by these methods. This is sadly true for deep learning methods.

There is something like an irreducible arbitrariness of choosing an optimization condition, or, if one gives up on fully understanding what is happening, an arbitrariness in the design of complex learning algorithms. This turns machine learning into a personal intuition thing, even into an art.

To put this all in a nutshell: the available training data do not include the information of how the information contained in them should be "optimally" extracted, or *what* kind of information should be extracted. This could be called a *lack of epistemic information* (my private terminology; epistemology is the branch of philosophy that is concerned with the question by which methods of reasoning humans acquire knowledge, and with the question what "knowledge" is in the first place - check out https://en.wikipedia.org/wiki/Epistemology if you are interested in these deep, ancient, unsolved questions).

The discussion of lacking epistemic information is hardly pursued in today's ML, although there have been periods in the last decades when fierce debates on such issues have been led.

But there is also another lack of information which *is* a standard theme in today's ML textbooks. This issue is called the *curse of dimensionality*. I will highlight this curse with our TICS demo again.

In mathematical terminology, our super-simplified 3-dimensional TICS system from Figure 5 is able (after training) to compute a function  $f : [0,1]^2 \rightarrow [0,1]$ which computes a caption  $f(u^*)$  for every input test image  $u^* \in [0,1]^2$ . The only information TICS has at learning time is contained in the training data points (blue crosses in our figure).

Looking at Figure 5, estimating a function  $f : [0,1]^2 \rightarrow [0,1]$  from just the information contained in the two blue crosses is clearly a dramatically underdetermined task. You may argue that the version of the TICS learning task which I gave in Figure 5 has been simplified to an unfair degree, and that the real TICS system had not just 2, but 30,000 training images to learn on.

But, in fact, for the full-size TICS the situation is even much, *much* worse than what appears in Figure 5:

- In our caricature demo there were 2 training images scattered in a 2-dimensional (pixel intensity) space. That is, as many data points as dimensions. In contrast, in TICS there are 30,000 images scattered in a 1,440,000-dimensional pixel intensity space: that is, about 50 times more dimensions than data points! How should it be possible at all to estimate a function from [0, 1]<sup>1440000</sup> to [0, 1]<sup>3000</sup> when one has far, far fewer training points than dimensions!?
- There is another fact about high-dimensional spaces which aggravates the situation. In the *n*-dimensional unit hypercube, the greatest possible distance between two points is equal to the longest diagonal in this hypercube,  $\sqrt{n}$ , which amounts to 1200 in the TICS image space. This growth of length of the main diagonal with dimension *n* carries over to a growth of average distances between random points in *n*-dimensional hypercubes. In simple terms, the higher the dimension, the wider will training data points lie away from each other.

The two conditions together – fewer data points than dimensions, and large distances between the points – make it appear impossible to estimate a function on  $[0, 1]^{1440000}$  from just those 30,000 points.

This is the dreaded "curse of dimensionality".

In plain English, the curse of dimensionality says that in high-dimensional data spaces, the available training data points will be spread exceedingly thinly in the embedding data space, and they will lie far away from each other. Metaphorically speaking, training data points are a few flickering stars in a vast and awfully empty universe. But, most learning tasks require one to fill the empty spaces with information — a pdf.

## 1.3 Looking at Human Intelligence, Again

An adult human has encoded in his/her brain a very useful, amazingly accurate and detailed model of that human's outer world, which allows the human to almost immediately create a "situation model" of his/her current physical and social environment, based on the current, ever-new sensor input. This situation model is a condensed, meaningful representation of the current environment. This is analog to the TICS scenario (after training), where the new input is a test image and the situation model corresponds to a distribution over captions.

After what we have seen above, how is it possible for a human brain to learn a meaningful and reliable representation of the world (that is, a manifold in brain state space) at all?

In fact, we do not know. But neuroscientists, psychologists and cognitive scientists have come up with a number of ideas.

An adult human has had a childhood and youth's time to learn most of his/her world model. The "training data" are the sensory impressions collected in, say, the first 25 years of the human's life. Thinking of a sensory impression as a vector, it is hard to say what is the dimension of this vector. A lower bound could be found in the number of sensory neural fibers reaching the brain. The optical nerve has about 1,000,000 fibres. Having two of them, plus all sorts of other sensory fibers reaching the brain (from the nose, ears, body), let us boldly declare that the dimension of sensor inputs to the brain is 3 million. Now, the learning process for a human differs from TICS learning in that sensory input arrives in a continuous stream, not in isolated training images. In order to make the two scenarios comparable, assume furthermore that a human organizes the continuous input stream into "snapshots" at a rate of 1 snapshot "picture" of the current sensory input per second (cognitive psychologists will tell us that the rate is likely less). In the course of 25 years, with 12 wake hours per day, this makes about 25  $\times$  360  $\times$  12  $\times$  3600  $\approx$  390 million "training snapshot images". This gives a ratio of number of training points over data dimension of 390 million / 3 million = 130, which looks so much better than the ratio of  $30,000 / 1,443,000 \approx 0.02$  in TICS's case.

But even the ratio of 130 data points per dimension is still hopeless, and the curse of dimensionality strikes just as well. Why?

For simplicity we again assume that the 3-million dimensional sensor image vectors are normalized to a range of [0, 1], making a sensor impression a point in  $[0, 1]^{3000000}$ . Statisticians, machine learners, and many cognitive scientists would tell us that the "world model" of a human can be considered to be a probability distribution over the sensory image space  $[0, 1]^{3000000}$ . This is a hypercube with  $2^{3000000}$  corners. In order to estimate a probability distribution over an *n*-dimensional hypercube, a statistician would demand to have many more datapoints than the cube has corners (to see why, think about estimating a probability distribution over the one-dimensional unit hypercube [0, 1] from data points on that interval, then extrapolate to higher dimensions). That is, a brain equipped with ordinary statistical methods would demand to have many more than  $2^{3000000}$  training data points to distil a world model from that collection of experiences. But there are only 390 million such datapoints collected in 25 years. The ratio

390 million /  $2^{3000000}$  is about  $2^{-2999972}$ . That is, a human would have to have a youth lifetime of about  $25 \times 2^{2999972} \approx 2^{2999977}$  years in order to learn a reliable world model – if we ask statisticians.

Still, the human brain (and the human around it, with the world around the human around that brain) somehow can do it in 25 years. Cognitive scientists believe that the key to this magic lies in the evolutionary history of man. Through millions of years of incremental, evolutionary brain structure optimization, starting from worm brains or earlier, the human brain is *pre-structured* in exactly such ways that it comes with a built-in-by-birth data manifold geometry which reduces the 3 million-dimensional raw sensor data format to a *much* lower-dimensional manifold surface. Then, 390 million data points may be enough to cover this manifold densely enough for meaningful distribution estimates.

The question which built-in experience pre-shapings a human brings to the table at birth time has a long tradition in philosophy and psychology. A recent line of work that brings this tradition to bear on machine learning is in the research of Joshua Tenenbaum – check out, for instance, Tenenbaum, Griffiths, and Kemp 2006 if you are interested.

In recent years, besides evolutionary predisposition, another line of thinking has started to be discussed as a solution to the question why humans (and maybe machines) can learn from embarrassingly small training samples. This is the approach of *causal modeling*. It has its origins not in biology or psychology, but in statistics, more specifically in the technique to model complex probability distributions with *graphical models*. We will treat them later in this course (Section 10). For now, I can only give an explanation in vague intuitive terms. The core idea of causal modeling is that in our real world, any phenomenon or experience is causally influencing only a very small portion of other pheonomena or experiences. When I walk and my foot hits a stone, the stone will be kicked away and my foot will hurt and I will make an extra step. That's it. The rest of the universe remains unaffected, and all of the other zillions of experiences that form my personal life training dataset simply have nothing to do with this stone hitting event. When walking through our lives, the huge collection of sensory experiences that we accumulate does not form an unorganized point cloud in some high-dimensional space of sensory experiences. We know, from bodily experience, which of these data points are *causally connected* to which others. To stay in the abstract geometrical picture: The data points are connected with causal impact lines. Each data point is causally connected only to a small number of others. These connecting lines span low-dimensional neighborhoods around each data point: a guide for guessing a local manifold geometry. — We will be able to formalize this after we have learnt about graphical models. If you already know a little about them and want to study ahead: a beautiful introduction to causal modeling is given by Schölkopf, Locatello, et al. 2021.

## 1.4 A Remark on "Modeling"

Machine learning in its modern form aims at modeling real-world systems, just like what the natural sciences aim for. But the motivation is different.

Physicists, chemists, biologists et al. want to *understand* reality through their models. Their models should tell the truth – in suitable abstraction – about reality; the models should be *veridical* (from Latin, "saying the truth"). The inner workings of a model should reflect the inner workings of reality. For instance, a detailed chemical model of a reaction, which changes a substance A into a substance B, should give an account of the kinetic and quantum-mechanical substeps that actually take place in this reaction. Newton's laws of motion explain the dance of planets by formulas made of variables which should correspond to physically real quantities – like gravitational forces, masses or velocities. Models whose inner mechanisms or mathematical variables are intended to capture real mechanisms and real quantities are called *analytical* models.

Machine learners, in contrast, are application oriented. They want *useful* models. ML models of pieces of reality must *function* well in their respective application context. The inner workings of an ML model need not mirror the inner workings of the modeled system. Machine learning models are thus almost always *blackbox* models (a possible exception being Bayesian networks, will be treated in Session 9 of this course). A blackbox model of some real-world system just captures the externally observable input-output behavior of the modeled system, but it may use any internal mathematical or algorithmical tricks that its designer can think of. Neural networks are a striking example. A neural network trained for predicting the next day's stock index will be made of hundreds or even millions of interacting variables ("neuron activations") which have no corresponding counterpart in the reality of stock markets.

Figure 6 sketches how blackbox models work. They are derived ("learnt", "estimated") from data emitted by the source system, and they should generate synthetic data that "fit" (have a similar distribution as) the real-world data. And that's it. The structure of the blackbox model need not agree in any way with the structure of the source system – in Figure 6, this is a robot, while the model is a neural network whose structure has no correspondence whatsoever in the robot design.

Analytical and blackbox models have complementary merits:

- Setting up the structure of an analytical model requires from the modeler to have insight into the concerned laws of nature – an expert's business. A blackbox model can be successfully created by a village idiot with access to TensorFlow.
- A blackbox model requires training data. There are cases where such data are not available in sufficient quantity. Then analytical modeling is the only way to go.



Figure 6: How blackbox models work. For explanation see text.

- As a rule, analytical models are much more compact than blackbox models. Compare  $E = m c^2$  with the TICS system, which consists of several modules, some of which are neural networks with hundreds of thousands of variables.
- When the target system that one wants to model is very complex, there is typically no chance for an analytical model. Example: an analytical model of how humans describe images in captions would have to include accounts of the human brain and the cultural conditions of language use. No way. Only blackbox models can be used here.
- The great really very great advantage of analytical models is that they generalize to all situations within their scope. The laws of gravitation can be applied to falling apples as well as to the majestic whirling of galaxies. If a blackbox model would have been trained by Newton on data collected from the sun and planets motions, this model would be exclusively applicable to our planetary system, not to apples and not to galaxies.

An interesting and very relevant modeling task is to model the earth's atmosphere for weather forecasting. This modeling problem has been intensely worked on for many decades, and an interesting mix of analytical and blackbox methods marks the state of the art. If there is time and interest I will expand on this in the tutorial session.

I conclude this section with a tale from my professional life, which nicely illustrates the difference between analytical and blackbox modeling. I was once called to consult a company in the chemical industry. They wanted a machine learning solution for the following problem. In one of their factories they had built a production line whose output was a certain artificial resin. Imagine a large hall full of vessels, heaters, pumps and values and tubes. The production process of that resin was prone to a particularly nasty possible failure: if the process would not be controlled correctly, some intermediate products might solidify. The concerned vessels, tubes and valves would be blocked and could not be cleared – necessitating to disassemble the facility and replace the congested parts with new ones. Very expensive! The chemical engineers had heard of the magic of neural networks and wanted one of these, which should give them early warnings if the production process was in danger of drifting into the danger zone. I told them that this was (maybe) possible if they could provide training data. What training data, please? Well, in order to predict failure, a neural network needs examples of this failure. So, could the engineers please run the facility through a reasonably large number of solidification accidents, a few hundreds maybe for good statistics? Obviously, that was that. Only analytical modeling would do here. A good analytical model would be able to predict any kind of imminent solidification situations. But that wasn't an option either because the entire production process was too complex for an accurate enough analytical model. Now put yourself into the skin of the responsible chief engineer. What should he/she do to prevent the dreaded solidification to happen, ever? Another nice discussion item for our tutorial session.

## 1.5 The Machine Learning Landscape

ML as a field, which perceives itself as a field under this name, is relatively young, say, about 40 years (related research was called "pattern recognition" earlier). It is interdisciplinary and has historical and methodological connections to neuroscience, cognitive science, linguistics, mathematical statistics, AI, signal processing and control; it uses mathematical methods from statistics (of course), information theory, signal processing and control, dynamical systems theory, mathematical logic and numerical mathematics; and it has a very wide span of application types. This diversity in traditions, methods and applications makes it difficult to study "Machine Learning". Any given textbook, even if it is very thick, will reflect the author's individual view and knowledge of the field and will be partially blind to other perspectives. This is quite different from other areas in computer science, say for example formal languages / theory of computation / computational complexity where a widely shared repertoire of standard themes and methods cleanly define the field.

I have a brother, Manfred Jaeger, who is a machine learning professor at Aalborg university (http://people.cs.aau.dk/~jaeger/). We naturally often talk with each other, but never about ML because I wouldn't understand what he is doing and vice versa. We have never met at scientific conferences because we attend different ones, and we publish in different journals.

The leading metaphors of ML have changed over the few decades of the field's

existence. The main shift, as I see it, was from "cognitive modeling" to "statistical modeling". In the 1970-1980s, a main research motive/metaphor in ML (which was hardly named like that then) was to mimic human learning on computers, which connected ML to AI, cognitive science and neuroscience. While these connections persist to the day, the mainstream self-perception of the field today is to view it very soberly as the craft of estimating complex probability distributions with efficient algorithms and powerful computers.

My personal map of the ML landscapte divides it into four main segments with distinct academic communities, research goals and methods:

- Segment 1: Theoretical ML. Here one asks what are the fundamental possibilities and limitations of inferring knowledge from observation data. This is the most abstract and "pure maths" strand of ML. There are cross-connections to the theory of computational complexity. Practical applicability of results and efficient algorithms are secondary. Check out https://en.wikipedia.org/wiki/Computational\_learning\_theory and https://en.wikipedia.org/wiki/Statistical\_learning\_theory for an impression of this line of research.
- Segment 2: Symbolic-logic learning, data mining. Here the goal is to infer symbolic knowledge from data, to extract logical rules from data, to infer facts about the real world expressed in fragments of first-order logic or other logic formalisms, often enriched with probabilistic information. Neural networks are rarely used. A main motive is that these resulting models be human-understandable and directly useful for human end-users. Key terms are "knowledge discovery", "data mining", or "automated knowledge base construction". This is the area of my brother's research. Check out https: //en.wikipedia.org/wiki/Data\_mining or https://en.wikipedia.org/ wiki/Inductive\_logic\_programming or Suchanek et al. 2013 for getting the flavor. This is an application-driven field, with applications e.g. in bioinformatics, drug discovery, web mining, document analysis, decision support systems.

A beautiful example is the PaleoDeepDive project described in Peters et al. 2014. This large-scale project aimed at making paleontological knowledge easily searchable and more realiable. Paleontology is the science of extinct animal species (also of extinct plant species, but we skip that part here). Its raw data are fossil bones. It is obviously difficult to reliably classify a handful of freshly excavated bones, which are typically poorly preserved, as belonging to a particular species – first, because one usually doesn't dig out a complete skeleton, and second because extinct species are not known in the first place. The field is plagued by misclassifications and terminological uncertainties – often a newly found set of bones is believed to belong to a newly discovered species, for which a new name is created, although in reality other fossil findings, already named differently by other researchers in other

publications, belong to the same species. In the PaleoDeepDive project, the web was crawled to retrieve virtually all scientific pdf documents relating to paleontology – including documents that had been published in pre-digital times and were just image scans. Using optical character recognition and image analysis methods at the front end, these documents were made machine readable, including information contained in tables and images. Then, unsupervised, logic-based methods were used to identify suspects for double naming of the same species, and also the opposite: single names for distinct species – an important contribution to purge the evolutionary tree of the animal kingdom.

- Segment 3: Signal and pattern modeling. This is the most diverse sector in my private partition of ML and it is difficult to characterize it globally. The basic attitude here is one of quantitative-numerical blackbox modeling. Our TICS demo would go here. The raw data are mostly numerical (like physical measurement timeseries, audio signals, images and video). When they are symbolic (texts in particular), one of the first processing steps typically encodes symbols to some numerical vector format. Neural networks are widely used and there are some connections to computational neuroscience. The general goal is to distil from raw data a numerical representation (often implicit) of the data distribution which lends itself to efficient application purposes, like pattern classification, time series prediction, or motor control, to name a few. Human-user interpretability of the distribution representation is not easy to attain, but has lately become an important subject of research. Like Segment 2, this field is decidedly application-driven. Under the catchword "deep learning" a subfield of this area has recently received a lot of attention.
- Segment 4: Agent modeling and reinforcement learning. The overarching goal here is to model entire intelligent agents humans, animals, robots, software agents that behave purposefully in complex dynamical environments. Besides learning, themes like motor control, sensor processing, decision making, motivation, knowledge representation, communication are investigated. An important kind of learning that is relevant for agents is *reinforcement learning* that is, an agent is optimizing its action-decision-making in a lifetime history based on reward and penalty signals. The outcome of research often is *agent architectures*: complex, multi-module "box-and-arrow diagrams" for autonomous intelligent systems. This is likely the most interdisciplinary corner of ML, with strong connections to cognitive science, the cognitive neurosciences, AI, robotics, artificial life, ethology and philosophy.

It is hard to judge how "big" these four segments are in mutual comparison. Surely Segment 1 receives much less funding and is pursued by substantially fewer researchers than segments 2 and 3. In this material sense, segments 2 and 3 are both "big". Segment 4 is bigger than Segment 1 but smaller than 2 or 3. My own research lies in 3 and 4. In this course I focus on the third segment — you should be aware that you only get a partial glimpse of ML.

Another very common textbook subdivision of ML, partly orthogonal to my private 4-section partition, is based on three fundamental kinds of learning tasks:

- **Supervised learning.** Training data are "labelled pairs"  $(x_n, y_n)$ , where x is some kind of "input" and y is some kind of "target output" or "desired / correct output". TICS is a typical example, where the  $x_n$  are images and the  $y_n$  are captions. The learning objective is to obtain a mechanism which, when fed with new test inputs  $x_{\text{test}}$ , returns outputs  $y_{\text{test}}$  that generalize in a meaningful way from the training sample. The underlying mathematical task is to estimate the conditional distributions  $P_{Y|X}$  from the training sample (check the end of Appendix B for a brief explanation of this notation). The learnt input-output mechanism is "good" to the extent that upon input  $x_{\text{test}}$  it generates outputs that are distributed according to the true conditional distribution  $P(Y = y \mid X = x_{\text{test}})$ , just as we have seen in the TICS demo. Typical and important special cases of supervised learning are *pat*tern classification (the y are correct class labels for the input patterns x) or *timeseries prediction* (the y are correct continuations of initial timeseries x). Segment 3 from my private segmentation of ML is the typical stage for supervised learning.
- **Unsupervised learning.** Training data are just data points  $x_n$ . The task is to discover some kind of "structure" (clusters, regularities, symmetries, redundancies...) in the data distribution which can be used to create a compressed representation of the data. Unsupervised learning can become very challenging when data points are high-dimensional and/or when the distribution has a complex shape. Unsupervised learning is often used for *dimension reduction.* The result of an unsupervised learning process is a dimension-reducing, encoding function e which takes high-dimensional data points x as inputs and returns low-dimensional encodings e(x). This encoding should preserve most of the information contained in the original inputs x. That is, there should also exist a *decoding* function d which takes encodings e(x) as inputs and transforms them back to the high-dimensional format of x. The overall information loss in the encoding-decoding process should be small, that is, one wishes to obtain  $x \approx d(e(x))$ . A discovery of underlying rules and regularities is the typical goal for data mining applications, hence unsupervised learning is the main mode for Segment 2 from my private dissection of ML. Unsupervised methods are often used for data preprocessing in other ML scenarios, because most ML techniques suffer from curse of dimensionality effects and work better with dimension-reduced input data.

- **Reinforcement learning.** The set-up for reinforcement learning (RL) is quite distinct from the above two. It is always related to an agent that can choose between different *actions* which in turn change the *state* of the environment the agent is in, and furthermore the agent may or may not receive *rewards* in certain environment states. RL thus involves at least the following three types of random variables:
  - action random variables A,
  - world state random variables S,
  - reward random variables R.

In most cases the agent is modeled as a stochastic process: a temporal sequence of actions  $A_1, A_2, \ldots$  leads to a sequence of world states  $S_1, S_2, \ldots$ , which are associated with rewards  $R_1, R_2, \ldots$ . The objective of RL is to learn a strategy (called *policy* in RL) for choosing actions that maximize the reward accumulated over time. Mathematically, a policy is a conditional distribution of the kind

$$P(A_n = a_n \mid S_1 = s_1, \dots, S_{n-1} = s_{n-1}; A_1 = a_1, \dots, A_{n-1} = a_{n-1}),$$

that is, the next action is chosen on the basis of the "lifetime experience" of previous actions and the resulting world states. RL is naturally connected to my Segment 4. Furthermore there are strong ties to neuroscience, because neuroscientists have reason to believe that individual neurons in a brain can adapt their functioning on the basis of neural or hormonal reward signals. Last but not least, RL has intimate mathematical connections to a classical subfield of control engineering called *optimal control*, where the (engineering) objective is to steer some system in a way that some long-term objective is optimized. An advanced textbook example is to steer an interplanetary missile from earth to some other planet such that fuel consumption is minimized. Actions here are navigation maneuvres, the (negative) reward is fuel consumption, the world state is the missile's position and velocity in interplanetary space.

The distinction between supervised and unsupervised learning is not clearcut. Training tasks that are globally supervised (like TICS) may benefit from, or plainly require, unsupervised learning subroutines for transforming raw data into meaningfully compressed formats (like we saw in TICS). Conversely, globally unsupervised training mechanisms may contain supervised subroutines where intermediate "targets" y are introduced by the learning system. Furthermore, today's advanced ML applications often make use of *semi-supervised* training schemes. In such approaches, the original task is supervised: learn some input-output model from labelled data  $(x_n, y_n)$ . This learning task may strongly benefit from including additional unlabelled input training points  $\tilde{x}_m$ , helping to distil a more detailed model of the input distribution  $P_X$  than would be possible on the basis of only the labelled  $x_n$  data. Again, TICS is an example: the data engineers who trained TICS used 70K un-captioned images in addition to the 30K captioned images to identify that 4096-dimensional manifold more accurately.

Also, reinforcement learning is not independent of supervised and unsupervised learning. A good RL scheme often involves supervised or unsupervised learning subroutines. For instance, an agent trying to find a good policy will benefit from data compression (= unsupervised learning) when the world states are high-dimensional; and an agent will be more capable of choosing good actions if it possesses an input-output model (= supervised learning) of the environment — inputs are actions, outputs are next states. *Deep reinforcement learning* is a flourishing branch of deep learning where learning agents encode some of their knowledge in deep neural networks.

Today machine learning is often identified with deep learning — at least, in the general public perception. Deep learning is (almost) exclusively based on neural networks as "learning machines". In our course we will treat neural networks only in one of thirteen sections, and only on a very basic level. I hope that the other sections reveal ML as a field that is much broader than deep learning. I also want to bring to the front a number of general insights and challenges that are the same in deep learning and other branches of ML. If you are particularly interested in deep learning, I must refer you to the specialization course on deep learning in our Master program, or to textbooks. There are two deep learning textbooks, both open-access and online, which I want to recommend. The first is the original deep learning "bible" which was written by some of the very pioneers of the field in 2016, when the basic methods of deep learning had been established, had matured, and had become famous and widely used. It's the book *Deep Learning* by I. Goodfellow, Bengio, and Courville 2016. The second is the book *Dive into Deep* Learning by Zhang et al. 2021, which, like the 2016 bible, gives a comprehensive introduction to the basics of deep learning, but furthermore an extensive overview of the current state-of-the-art architectures and algorithms, with detailed instructions concerning their efficient implementation. Since the field is developing very fast, the 2021 book has grown to 1000+ pages, 220 pages more than the 2016 book. Both books include extended introductions to the mathematical methods that underpin deep learning.

## 2 Decision trees and random forests

This section describes a class of machine learning models that is classical and simple and intuitive and useful. Decision trees and random forests are not super fashionable in these deep learning times, but practicians in data analysis use them on a daily basis. The main inventions in this field have been made around 1980-2000. In this chapter I rely heavily on the decision tree chapters in the classical ML textbooks of Mitchell 1997 and Duda, P. E. Hart, and Stork 2001, and for the random forest part my source is the landmark paper by Breiman 2001 which, as per Nov 15, 2021, has been cited 81000 times (Google Scholar). Good stuff to know, apparently.

Note: In this chapter of the lecture notes I will largely adhere to the notation used by Breiman 2001 in order to make it easier for you to read that key paper if you want to dig deeper. If in your professional future you want to use decision tree methods, you will invariably use them in random forests, and in order to understand what you are actually doing, you will have to read Breiman 2001 (like hundreds of thousands before you). Unfortunately, the notation used by Breiman is inconsistent, which makes the mathematical part of that paper hard to understand. My hunch is that most readers skipped the two mathy sections and read only the experimental sections with the concrete helpful hints for algorithm design, and with gorgeous demo results. Inconsistent or even plainly incorrect mathematical notation (and mathematical thinking) happens a lot in "engineering applied math" papers and makes it difficult to really understand what the author wants to say (see my ramblings in my lecture notes on "Principles of Statistical Modeling", Chapter 14, "A note on what you find in textbooks", online at https://www.ai.rug.nl/ minds/uploads/LN\_PSM.pdf). Therefore I will not use Breiman's notation oneto-one, but modify it a little to make it mathematically more consistent.

## 2.1 A toy decision tree

Figure 7 shows a simple decision tree. It can be used to classify fruit. If you have a fruit in your hands that you don't know by name, then you can use this decision tree to find out what fruit you have there, in an obvious way:

- Start at the root node of the tree. The root node is labeled with a property that fruit have (color? is the root property used in the figure).
- Underneath the root node you find child nodes, one for each of the three possible color attributes *green*, *yellow*, *red*. Decide which color your fruit has and proceed to that child node. Let us assume that your fruit was yellow. Then you are now at the child node labelled **shape**?.
- Continue this game of moving downwards in the tree according to the direction decisions taken at each node according to the attributes of the fruit in

your hand. If you reach a leaf node of this tree, it will be labeled with the type of your fruit. If your fruit is yellow, round and small, it's a lemon!



Figure 7: A simple decision tree. Taken from Duda, P. E. Hart, and Stork 2001.

For a given classification task, there are many different decision trees that give identical results. A decision tree is not unique. You could, for instance, create another tree whose root node queries the **size**? property, and which would lead to exactly the same ultimate decisions as the tree shown in the figure (exercise!).

If each internal (= non-leaf) node in a decision tree has exactly two child nodes, one speaks of a binary decision tree. These are particularly pleasant for mathematical analysis and algorithm design. Every decision tree can be transformed into an equivalent binary decision tree with an obvious procedure. Figure 8 shows the binary tree obtained from the fruit tree of Figure 7.

This is obviously a toy example. But decision trees can become really large and useful. A classical example are the botanic field guide books used by biologists to classify plants (Figure 9). I am a hobby botanist and own three such decision trees in book format. Together they weigh about 1.5 kg and I have been using them ever since I was eighteen or so.

In the basic versions of decision tree learning, they are learnt from data which are given as class-labeled attribute vectors. For instance, training data for our fruit classification tree might look like this:

nr.	Color	Size	Taste	Weight	Shape	Class
1	green	big	sweet	heavy	round	Watermelon
2	green	?	?	heavy	oval	Watermelon
3	red	small	sweet	light	round	Cherry
4	red	small	sweet	light	round	Apple
3000	red	big	?	medium	round	Apple



Figure 8: A binary version of the decision tree shown in Figure 7. Taken from Duda, P. E. Hart, and Stork 2001.

Here we get a first inkling that decision tree learning might not be as trivial as the final result in Figure 7 makes it appear. The above fruit training data table has missing values (marked by "?"); not all properties from the training data are used in the decision tree (property "Weight" is ignored); some examples of the same class have different attribute vectors (first two rows give different characterizations of watermelons); some identical attribute vectors have different classes (rows 3 and 4). In summary: real-world training data will be partly redundant, partly inconsistent and will contain errors and gaps. All of this points in the same direction: statistical analyses will be needed to learn decision trees.

## 2.2 Formalizing "training data"

Before we describe decision tree learning, I want to squeeze in a little rehearsal of mathematical concepts and their notation.

In my notation in this chapter I will be leaning on Breiman's notation as far as possible. Breiman's notation is unfortunately incomplete and inconsistent. Here I use a complete, correct notation, adhering to the standards of mathematical probability theory. I am aware that many participants of this course will be unfamiliar with probability theory and its formal notation standards. Because machine learning ultimately rests on probability theory, I strongly recommend that, if you want to pursue a professional career in machine learning, at some point you learn to master the basics of probability theory (for instance, from my lecture notes for the legacy course "Principles of Statistical Modeling", online: https://www.ai.rug.nl/minds/uploads/LN\_PSM.pdf). Only if you understand probability you can get a full grasp on machine learning. But it is clear that many students will not have the time or inclination to learn about probability theory.



Figure 9: Another decision tree. The right image shows a page from this botany field guide. What you see on this page is, in fact, a small section of a large binary classification tree. Image source: iberlibro.com, booklooker.de.

Therefore I will try to introduce notation and concepts in two ways throughout this course: in a rigorous "true math" version, and in an intuitive "makes some sense" version.

I called the fruit attribute data table above a *sample*. Samples are mathematical objects of key importance in statistics and machine learning (where they are also called "training data"). Samples are always connected with *random variables* (RVs). Here is how.

First, the intuitive version. As an empirical fruit scientist, you would obtain a "random draw" to get the training data table in a very concrete way: you would go to the fruit market, collect 3000 fruit "at random", observe and note down their color, size, taste, weight and shape attributes in an Excel table, and for each of the 3000 fruit you also ask the vendor for the name of the fruit to get an almost but not quite surely correct class label which you also note down in the last column of the table.

The mathematical representation of observing and noting down the attributes of the *i*-th fruit that you have picked (where  $i = 1, \ldots, 3000$ ) is  $X_i(\omega)$ .  $X_i$  is the random variable which is the mathematical model of carrying out the *i*-th observation that you do on a fruit market. This  $X_i$  could be described as the procedure "pick a random fruit and observe and report the attribute vector". The  $\omega$  in  $X_i(\omega)$  is the occasion when you actually executed the procedure  $X_i$  – say,  $\omega$  stands for the concrete data collection trip when you went to the Vismarkt in Groningen last Tuesday and visited the fruit stands.  $X_{i+1}(\omega)$  would be the attribute vector of the next fruit that you pick up on that data collection trip. If the entire procedure of collecting 3000 fruit specimen were executed on another occasion – for instance, a week earlier, or by your friend on the same day – this would be mathematically represented by another  $\omega'$ . For instance,  $X_i(\omega)$  might be the attribute vector that you observed for the *i*-th fruit last Tuesday,  $X_i(\omega')$ would be the attribute vector that you observed for the *i*-th fruit one week earlier,  $X_i(\omega'')$  would be the attribute vector that your friend observed for the *i*-th last Tuesday when he did the whole thing in parallel with you, etc. In mathematical terminology, these "observation occasions"  $\omega$  are called *elementary events*.

Similarly,  $Y_i(\omega)$  is the fruit class name that you were told by the vendor when you did the data sampling last Tuesday;  $Y_i(\omega')$  would be the *i*th fruit name you were told when you did the whole exercise a week earlier already, etc.

The three thousand pairs  $(X_i(\omega), Y_i(\omega))$  correspond to the rows in the data table. I will call each of these data (row) vectors  $(X_i(\omega), Y_i(\omega))$  a *data point*. Database people and practical field workers call these vectors *records*. To get the entire table as a single mathematical object – namely, the *sample* – one combines these single fruit data points by a *product* operation, obtaining  $(\mathbf{X}(\omega), \mathbf{Y}(\omega))$ , where  $\mathbf{X} = \bigotimes_{i=1,\dots,N} X_i$  and  $\mathbf{Y} = \bigotimes_{i=1,\dots,N} Y_i$ . Products of random variables are used abundantly in probability — you find them explained in a little more detail in Appendix A.

And here is the rigorous probability theory account of the  $(\mathbf{X}(\omega), \mathbf{Y}(\omega))$  notation. (This material is not mandatory and will not be asked in exercises or exams — it is just a glimpse of the raw beauty of mathematics, for the ones who are interested. The sketch I can give here is super condensed. I tell the same story, with many detailed examples and explanations, in my 'Statistical Modeling' lecture notes on 100 pages.) As always in probability theory and statistics, we have an underlying probability space  $(\Omega, \mathfrak{A}, P)$ . In this structure,  $\Omega$  is the set of all possible elementary events  $\omega \in \Omega$ .  $\mathfrak{A}$  is a subset structure imposed on the set  $\Omega$ called a  $\sigma$ -field, and P is a probability measure on  $(\Omega, \mathfrak{A})$ . The random variable  $\mathbf{X}$ is a function which returns samples, that is collections of N training data points. The expression  $\mathbf{X}(\omega)$  is a function call: the argument  $\omega$  is (intuitively speaking) an observation event, and the result  $\mathbf{X}(\omega)$  of this function call is a sample. We assume that all samples  $\omega$  that could be drawn have N data points; this assumption is a matter of convenience and simplifies notation and mathematical analysis a lot. It is a good exercise to formalize the structure of a sample in more detail. A single data point consists of an attribute vector  $\mathbf{x}$  and a class label y. In formal notation, we have m properties  $Q_1, \ldots, Q_m$  (like 'color' or 'size'), and each property  $Q_j$  (where  $j \in \{1, \ldots, m\}$ ) has a set  $A_j$  of possible attribute values (like 'red', 'green' as attributes of 'color' etc.). Thus,  $\mathbf{x} \in A_1 \times \ldots \times A_m$ . We denote the set of possible class labels by C. A data point  $(X_i(\omega), Y_i(\omega))$  is thus an element of the set  $(A_1 \times \ldots \times A_m \times C)$ . See Appendix A for the mathematical notation used in creating product data structures. With  $S_Z$  we denote the sample space of any random variable Z. The sample space for  $X_i$  is thus  $S_{X_i} = A_1 \times \ldots \times A_m$  and for  $Y_i$  it is  $S_{Y_i} = C$ . Because  $S_{X_i} = S_{X_j}, S_{Y_i} = S_{Y_j}$  for all  $1 \leq i, j \leq N$ , for simplicity we also write  $S_X$  for  $A_1 \times \ldots \times A_m$  and  $S_Y$  for C.

The entire training data table is an element of the sample space  $S_{\mathbf{X}\otimes\mathbf{Y}}$  of the product RV  $\mathbf{X}\otimes\mathbf{Y} = (\bigotimes_{i=1,\dots,N} X_i)\otimes(\bigotimes_{i=1,\dots,N} Y_i)$ . Just to exercise formalization skills: do you see that

$$S_{\mathbf{X}} \times S_{\mathbf{Y}} = (A_1 \times \ldots \times A_m)^N \times C^N = (A_1 \times \ldots \times A_m \times C)^N = S_{\mathbf{X} \otimes \mathbf{Y}}?$$

To round off our rehearsal of elementary concepts and notation, I repeat the basic connections between random variables, probability spaces and sample spaces: a random variable Z always comes with an underlying probability space  $(\Omega, \mathfrak{A}, P)$  and a sample space  $S_Z$ . The RV is a function  $Z : \Omega \to S_Z$ , and induces a probability distribution on  $S_Z$ . This distribution is denoted by  $P_Z$ .

I torture you with these exercises in notation like a Russian piano teacher will torture his students with technical finger-exercising *études*. It is technical and mechanical and the suffering student may not happily appreciate the necessity of it all, – and yet this torture is a precondition for becoming a virtuoso. The music of machine learning is played on the keyboard of probability, no escape.

I am aware that, if you did not know these probability concepts before, this condensed rehearsal is hardly understandable. Probability theory is one of the most difficult-to-grasp sectors of mathematics and it takes weeks of digesting and exercising to embrace the concepts of probability spaces, random variables and sample spaces (not to speak of  $\sigma$ -fields). I want to recommend again my lecture notes "Principles of Statistical Modeling", which I mentioned before. They come from a graduate course whose sole purpose was to give a slow, detailed, understandable, yet mathematically rigorous introduction to the basic concepts of probability theory and statistics — for non-math students from computer science and engineering.

## 2.3 Learning decision trees: setting the stage

After this excursion into the rough terrain of probability notation we can start to discuss decision tree learning.

Despite the innocent looks of decision trees, no universally "optimal" method is known how to learn them from training data. But it is known what are the basic design options for decision tree learning algorithms. There are a few key decisions the algorithm designer has to make in order to assemble a learning algorithm for decision trees, and standard algorithmic building blocks for each of the design decisions are known. We will now inspect these design options in turn.

Two things must be given before a learning algorithm can be assembled and learning can start: (A) the training data, and (B) an optimization criterion. While (A) is obvious, (B) deserves a few words of explanation.

We observe that decision tree learning is a case of supervised learning. The training data is a collection of labeled samples  $(\mathbf{x}_i, y_i)_{i=1,...,N} = (X_i(\omega), Y_i(\omega))_{i=1,...,N}$ , where  $\mathbf{x}_i \in A_1 \times \ldots A_m = S_X$  is an attribute vector and  $y_i \in C = S_Y$  is a class label. A decision tree represents a function  $h: S_X \to S_Y$  in an obvious way: using the sequential decision procedure outlined in Section 2.1, a vector of observed attributes leads you from the root node to some leaf, and that leaf gives a class label. In statistics, such functions  $h: S_X \to S_Y$  are generally called *decision functions* (also in other supervised learning settings that do not involve decision trees), a term that I will also sometimes use.

Decision tree learning is an instance of the general case of supervised learning: the training data are labeled pairs  $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$ , with  $\mathbf{x}_i \in S_X, y_i \in S_Y$ , and the learning aims at finding a decision function  $h: S_X \to S_Y$  which is "optimal" in some sense.

In what sense can such a function  $h: S_X \to S_Y$  (in this section: such a decision tree) be "optimal"? How can one quantify the "goodness" of functions of the type  $h: S_X \to S_Y$ ?

This is a very non-trivial question, as we will learn to appreciate as the course goes on. For supervised learning tasks, the key to optimizing a learning procedure is to declare a *loss function* at the very outset of a learning project. A loss function is a function

$$L: S_Y \times S_Y \to \mathbb{R}^{\ge 0},\tag{1}$$

which assigns a nonnegative real number to a pair of class labels. The loss function is used to compare the classification  $h(\mathbf{x})$ , returned by a decision function h on input  $\mathbf{x}$ , with the correct value y. After having learnt a decision function h from a training sample  $S_{\mathbf{X}} \times S_{\mathbf{Y}}$ , one can quantify the performance of h averaged over the training data, obtaining a quantity  $R^{emp}$  that is called the *empirical risk*:

$$R^{\mathsf{emp}}(h) = \frac{1}{N} \sum_{i=1,\dots,N} L(h(\mathbf{x}_i), y_i).$$
(2)

Most supervised machine learning procedures are built around some optimization algorithm that searches for decision functions which minimize the empirical risk, that is, which try to give a small average loss on the training data.

I emphasize that minimizing the empirical loss (or in another common wording, minimizing the "training error") by a clever learning algorithm will usually not lead to a very useful decision function. This is due to the problem of *overfitting*.

"Overfitting" means, intuitively speaking, that if the learning algorithm tries everything to be good on the training data (= small empirical risk), it will attempt to minimize the loss individually for each training data point. This means, in some way, to learn the training data "by heart" – encode an exact memory of the training data points in the learnt decision function h. This is not a good idea because later test input data points will be different, and the decision function obtained by "rote learning" will not know how to generalize to the new data points. The ultimate goal of supervised machine learning is not to minimize the empirical loss (that is, to have small loss on the training data), but to minimize the loss on future test data which were not available at training time. Thus, the central optimization criterion for supervised learning methods is to find decision functions h that have a small risk

$$R(h) = E[L(h(X), Y)], \tag{3}$$

where E is the statistical expectation operator (see Appendix D). That is, a good decision function h should incur a small expected loss – it should have small "testing error". At a later point in this course we will analyse the problem of avoiding overfitting in more detail. For now it is enough to be aware that overfitting is a very serious threat, and that in order to avoid it one should not allow the models h to dress themselves too closely around the training data points.

For a classification task with a finite number of class labels, like in our fruit example, a natural loss function is one that simply counts misclassifications:

$$L^{\mathsf{count}}(h(\mathbf{x}), y) = \begin{cases} 0, & \text{if } h(\mathbf{x}) = y \\ 1, & \text{if } h(\mathbf{x}) \neq y \end{cases}$$
(4)

This *counting loss* is often a natural choice, but in many situations it is not appropriate. Consider, for instance, medical diagnostic decision making. Assume you visit a doctor with some vague complaints. Now compare two scenarios.

- Scenario 1: after doing his diagnostics the doctor says, "sorry to tell you but you have cancer and you should think about making your will" and this is a wrong diagnosis; in fact you are quite healthy.
- Scenario 2: after doing his diagnostics the doctor says, "good news, old boy: there's nothing wrong with you except you ate too much yesterday" and this is a wrong diagnosis; in fact you have intestinal cancer.

These two errors are called the "false positive" and the "false negative" decisions. A simple counting loss would optimize medical decision making such that the average number of any sort of error is minimized, regardless whether it is a false negative or a false positive. But in medicine, false negatives should be avoided as much as possible because their consequences can be lethal, whereas false positives will only cause a passing anxiety and inconvenience. Accordingly, a loss function used to optimize medical decision making should put a higher penality (= larger L values) on false negatives than on false positives.
Generally, and specifically so in *operations research* and *decision support sys*tems, loss functions can become quite involved, including a careful balancing of ethical, financial or other factors. However, we will not consider complex loss functions here and stick to the simple counting loss. This loss is the one that guided the design of the classical decision tree learning algorithms. I am not aware of any work where another loss function was used to optimize or evaluate decision trees. Decision trees can be regarded as 'counting-loss minimization machines'.

And now, finally, we have set the stage for actually discussing learning algorithms for decision trees.

#### 2.4 Learning decision trees: the core algorithm

Ok., let us get concrete. We are given training data  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  (in real life often an Excel table), and – opting for the counting loss – we want to find an optimal (more realistically: a rather good) decision tree  $h^{\mathsf{opt}}$  which will minimize the expected misclassification ratio on new "test" data.

In order to distil this  $h^{\text{opt}}$  from the training data, we need to set up a *learning* algorithm which, on input of a training sample  $(\mathbf{x}_i, y_i)_{i=1,...,N}$ , outputs the ready-to-use decision tree  $h^{\text{opt}}$ .

All known learning algorithms for decision trees build  $h^{opt}$  incrementally from the root node downwards. The first thing a decision tree learning algorithm (DTLA) must do is therefore to decide which property is queried first, making it the root node.

To understand the following procedures, notice that a decision tree iteratively splits the training dataset in increasingly smaller, disjoint subsets. The root node can be associated with the entire training dataset – call it D. If the root node  $\nu_{\text{root}}$ queries the property  $Q_j$  and this property has  $k_j$  attributes  $a_1^j, \ldots, a_{k_j}^j$ , there will be  $k_j$  child nodes  $\nu_1, \ldots, \nu_{k_j}$  where node  $\nu_l$  will be covering all training datapoints that have attribute  $a_l^j$ ; and so forth down the tree. In detail: if  $\nu_{l_1 l_2 \cdots l_r}$  is a tree node at level r (counting from the root), and this node is associated with the subset  $D_{l_1 l_2 \cdots l_r}$  of the training data, and this node queries property  $Q_u$  which has attributes  $a_1^u, \ldots, a_{k_u}^u$ , then the child node  $\nu_{l_1 l_2 \cdots l_r l_s}$  will be that subset of  $D_{l_1 l_2 \cdots l_r}$ which contains those training data points that have attribute  $a_s^u$  of property  $Q_u$ .

The classical solution to the problem of selecting a property Q for the root node is to choose that property which leads to a "maximally informative" split of the training dataset in the first child node level. Intuitively, the data point subsets associated with the child nodes should be as "pure" as possible with respect to the classes  $c \in C$ .

In the best of all cases, if there are q different classes (that is, |C| = q), there would be a property  $Q_{\text{supermagic}}$  with q attributes which already uniquely identify classes, such that each first-level child node is already associated with a "pure" set of training examples all from the same class – say, the first child node covers only apples, the second only bananas, etc. This will usually not be possible, among other reasons because normally there is no property with exactly as many attributes as there are classes. Thus, "purity" of data point sets associated with child nodes needs to be measured in a way that tolerates class mixtures in each node. The measure that is traditionally invoked in this situation comes from information theory. It is the *entropy* of the class distribution within a child node. If  $D_l$  is the set of training points associated with a child node  $\nu_l$ , and  $n_i^l$  is the number of data points in  $D_l$  that are from class *i* (where  $i = 1, \ldots, q$ ), and the total size of  $D_l$  is  $n^l$ , then the entropy  $S_l$  of the "class mixture" in  $D_l$  is given by

$$S_l = -\sum_{i=1,\dots,q} \frac{n_i^l}{n^l} \log_2\left(\frac{n_i^l}{n^l}\right).$$
(5)

If in this sum a term  $\frac{n_i^l}{n^l}$  happens to be zero, by convention the product  $\frac{n_i^l}{n^l} \log_2\left(\frac{n_i^l}{n^l}\right)$  is set to zero, too. I quickly rehearse two properties of entropy which are relevant here:

- Entropies are always nonnegative.
- The more "mixed" the set  $D_l$ , the higher the entropy  $S_l$ . In one extreme case,  $D_l$  is 100% clean, that is, it contains only data points of a single class. Then  $S_l = 0$ . The other extreme is the greatest possible degree of mixing, which occurs when there is an equal number of data points from each class in  $D_l$ . Then  $S_l$  attains its maximal possible value of  $-q(1/q) \log_2(1/q) = \log_2 q$ .

When  $S_l$  is zero, the set  $D_l$  is "pure" — it contains examples of only one class. The larger  $S_l$ , the less pure  $D_l$ . The entropy measure  $S_l$  is often called an *impurity* measure.

The entropy of the root node is

$$S_{\text{root}} = -\sum_{i=1,\dots,q} \frac{n_i^{\text{root}}}{N} \log_2\left(\frac{n_i^{\text{root}}}{N}\right),$$

where  $n_i^{\text{root}}$  is the number of examples of class *i* in the total training data set *D* and *N* is the number of training data points. Following the terminology of Duda, P. E. Hart, and Stork 2001, I will denote the impurity measure of the root by  $i_{\text{entropy}}(\nu_{\text{root}}) := S_{\text{root}}$ .

Generally, if  $\nu$  is some node in the tree, and this node is associated with the training data point set  $D_{\nu}$ , and  $|D_{\nu}| = n$ , and the *q* classes are represented in  $D_{\nu}$  by subsets of size  $n_1, \ldots, n_q$ , the entropy impurity of  $\nu$  is given by

$$i_{\text{entropy}}(\nu) = -\sum_{i=1,\dots,q} \frac{n_i}{n} \log_2\left(\frac{n_i}{n}\right).$$
(6)

If the root node  $\nu_{\text{root}}$  queries the property  $Q_j$  and this property has k attributes, then the impurity mixing of classes averaged over all child nodes  $\nu_1, \ldots, \nu_k$  is given by

$$\sum_{l=1,\dots,k} \frac{|D_l|}{N} i_{\text{entropy}}(\nu_l),$$

where  $D_l$  is the point set associated with  $\nu_l$ .

Subtracting this average class mixing found in the child nodes from the class mixing in the root, one obtains the *information gain* achieved by opting for property  $Q_j$  for the root node:

$$\Delta i_{\text{entropy}}(\nu_{\text{root}}, Q_j) = i_{\text{entropy}}(\nu_{\text{root}}) - \sum_{l=1,\dots,k} \frac{|D_l|}{N} i_{\text{entropy}}(\nu_l).$$
(7)

It can be shown that this quantity is always nonnegative. It is maximal when all child nodes have pure data point sets associated with them.

Similarly, for any other node  $\nu$  labeled with property Q, where Q has k attributes, and the size of the set associated with  $\nu$  is n and the sizes of the sets associated with the k child nodes  $\nu_1, \ldots, \nu_k$  are  $n_1, \ldots, n_k$ , the information gain for node  $\nu$  is

$$\Delta i_{\text{entropy}}(\nu, Q) = i_{\text{entropy}}(\nu) - \sum_{l=1,\dots,k} \frac{n_l}{n} i_{\text{entropy}}(\nu_l).$$
(8)

The procedure to choose a property for the root node is to compute the information gain for all properties and select the one which maximizes the information gain.

This procedure of choosing that query property which leads to the greatest information gain is repeated tree-downwards as the tree is grown by the DTLA. A node is not further expanded and thus becomes a leaf node if (i) either the training dataset associated with this node is 100% pure (contains only examples from a single class), or if (ii) one has reached the level at which all available properties have been queried on the path from the root to that node. In case (i) the leaf is labeled with the unique class of its associated data point set. In case (ii) it is labeled with the class that has the largest number of representatives in the associated data point set.

Information gain is the most popular and historically first criterion used for determining the query property of a node. But other criteria are used too. I mention three.

The first is a normalized version of the information gain. The motivation is that the information gain criterion (8) favours properties that have more attributes over properties with fewer attributes. This requires a little explanation. Generally speaking, properties with many attributes statistically tend to lead to purer data point sets in the children nodes than properties with only few attributes. To see this, compare the extreme cases of a property with only a single attribute, which will lead to a zero information gain, with the extreme case of a property that has many more attributes than there are data points in the training dataset, which will (statistically) lead to many children node data point sets which contain only a single example, that is, they are 100% pure. A split of the data point set  $D_{\nu}$ into a large number of very small subsets is undesirable because it is a dooropener for overfitting. This preference for properties with more attributes can be compensated if the information gain is normalized by the entropy of the data set splitting, leading to the *information gain ratio* criterion, which here is given for the root node:

$$\Delta_{\text{ratio}} i_{\text{entropy}}(\nu_{\text{root}}, Q_j) = \frac{\Delta i_{\text{entropy}}(\nu_{\text{root}}, Q_j)}{-\sum_{l=1,\dots,k} \frac{|D_l|}{N} \log_2 \frac{|D_l|}{N}}.$$
(9)

The second alternative that I mention measures the impurity of a node by its *Gini impurity*, which for a node  $\nu$  associated with set  $D_{\nu}$ , where  $|D_{\nu}| = n$  and the subsets of points in  $D_{\nu}$  of classes  $1, \ldots, q$  have sizes  $n_1, \ldots, n_q$ , is

$$i_{\mathsf{Gini}}(\nu) = \sum_{1 \le i, j \le q; i \ne j} \frac{n_i}{n} \, \frac{n_j}{n} = 1 - \sum_{1 \le i \le q} \left(\frac{n_i}{n}\right)^2,$$

which is the error rate when a category decision for any example point in  $D_{\nu}$  is made randomly according to the class distribution within  $D_{\nu}$ .

The third alternative is called the *misclassification impurity* in Duda, P. E. Hart, and Stork 2001 and is given by

$$i_{\text{misclass}}(\nu) = 1 - \max\{\frac{n_l}{n} \mid l = 1, \dots, q\}.$$

This impurity measure is the minimum (taken over all classes c) probability that a point from  $D_{\nu}$  which is of class c is misclassified if the classification is randomly done according to the class distribution in  $D_{\nu}$ .

Like the entropy impurity, the Gini and misclassification impurities are nonnegative and equal to zero if and only if the set  $D_{\nu}$  is 100% pure. Like it was done with the entropy impurity, these other impurity measures can be used to choose a property for a node  $\nu$  through the gain formula (8), plugging in the respective impurity measure for  $i_{\text{entropy}}$ .

This concludes the presentation of the core DTLA. It is a *greedy* procedure which incrementally constructs the tree, starting from its root, by always choosing that property for the node currently being constructed which maximizes the information gain (7) or one of the alternative impurity measures.

You will notice some unfortunate facts:

• An optimization scheme which works by iteratively applying *local* greedy optimization steps will generally not yield a *globally* optimal solution.

- We started from claiming that our goal is to learn a tree which minimizes the count loss (or any other loss we might opt for). However, none of the impurity measures and the associated local gain optimization is connected in a mathematically transparent way with that loss. In fact, no computationally tractable method for learning an empirical-loss-minimal tree exists: the problem is NP-complete (Hyafil and Rivest 1976).
- It is not clear which of the impurity measures is best for the goal of minimizing the empirical loss.

In summary, we have here a *heuristic* learning algorithm (actually, a family of algorithms, depending on the choice of impurity measure) which is based on intuition and the factual experience that it works reasonably well in practice. Only heuristic algorithms are possible due to the NP-completeness of the underlying optimization problem.

## 2.5 Dealing with overfitting

If a decision tree is learnt according to the core algorithm, there is the danger that it overfits. This means that it performs well on the training data – it has small empirical loss – which boils down to a condition where all leaf nodes are rather pure. A zero empirical loss is easily attained if the number of properties and attributes is large. Then it will be the case that every training example has a unique combination of attributes, which leads to 100% pure leaf nodes, and every leaf having only a small set of training examples associated with it, maybe singleton sets. Then one has zero misclassifications of the training examples. But, intuitively speaking, the tree has just memorized the training set. If there is some "noise" in the attributes of data points, new examples (not in the training set) will likely have attribute combinations that lead the learnt decision tree on wrong tracks.

We will learn how to analyze and systematically fight overfitting later in the course. At this moment I only point out that when a learnt tree T overfits, one can typically obtain from it another tree T' which overfits less, by pruning T. Pruning a tree means to select some internal nodes and delete all of their children. This makes intuitive sense: the deeper one goes down in a tree learnt by the core algorithm, the more does the branch reflect "individual" attribute combinations found in the training data.

To make this point particularly clear, consider a case where there are many properties, but only one of them carries information relevant for classification, while all others are purely random. For instance, for a classification of patients into the two classes "healthy" and "has\_cancer", the binary property "has\_increased\_ leukocyte\_count" carries classification relevant information, while the binary property "given\_name\_starts\_with\_A" is entirely unconnected to the clinical status. If there are enough such irrelevant properties to uniquely identify each patient in the training sample, the core algorithm will (i) most likely find that the relevant property "has\_increased\_leukocyte\_count" leads to the greatest information gain at the root and thus use it for the root decision, and (ii) subsequently generate tree branches that lead to 100% pure leaf nodes. Zero training error and poor generalization to new patients results. The best tree here would be the one that only expands the root node once, exploiting the only relevant property.

With this insight in mind, there are two strategies to end up with trees that are not too deep.

The first strategy is *early stopping*. One does not carry the core algorithm to its end but at each node that one has created, one decides whether a further expansion would lead to overfitting. A number of statistical criteria are known to decide when it is time to stop expanding the tree; or one can use cross-validation (explained in later chapters in these lecture notes). We do not discuss this further here – the textbook by Duda explains a few of these statistical criteria. A problem with early stopping is that it suffers from the *horizon effect*: if one stops early at some node, a more fully expanded tree might exploit further properties that are, in fact, relevant for classification refinement underneath the stopped node.

The second strategy is *pruning*. The tree is first fully built with the core algorithm, then it is incrementally shortened by cutting away end sections of branches. An advantage of pruning over early stopping is that it avoids the horizon effect. Duda says (I am not a decision tree expert and can't judge) that pruning should be preferred over early stopping "in small problems" (whatever "small" means).

## 2.6 Variants and refinements

The landscape of decision tree learning is much richer than what we can explore in a single lecture. Two important topics that I omit (treated briefly in Duda's book):

- Missing values. Real-world datasets typically have missing values, that is, not all training or test examples will have attribute values filled in for all properties. Missing values require adjustments both in training (an adapted version of impurity measures which accounts for missing values) and in testing (if a test example leads to a node with property Q and the example misses the required attribute for Q, the normal classification procedure would abort). The Duda book dedicates a subsection to recovery algorithms.
- Numerical properties. Many real-world properties, like for instance "velocity" or "weight", have a continuous numerical range for attribute values. This requires a split of the continuous value range into a finite number (often two) of discrete bins which then serve as discrete attributes. For instance, the range of a "body\_weight" property might be split into two ranges "<50kg" and "≥50kg", leading to a new property with only two attributes. These ranges

can be further split into finer-grained subproperties which would be queried downstream in the tree, for instance querying for "<45kg" vs. "45kg $\leq$ 50kg underneath the "<50kg" node. Heuristics must be used to determine splitting boundaries which lead to well-performing (correct and cheap = not too many nodes) trees. Again – consult Duda.

All in all, there is a large number of design decisions to be made when setting up a decision tree learning algorithm, and a whole universe of design criteria and algorithmic sub-procedures is available in the literature. Some combinations of such design decisions have led to final algorithms which have become branded with names and are widely used. Two algorithms which are invariably cited (and which are available in statistical and ML programming toolboxes) are the ID3 algorithm and its more complex and higher-performant successor, the C4.5 algorithm. They had been introduced by decision tree pioneer Ross Quinlan in 1986 and 1993, respectively. The Duda book gives brief descriptions of these classical algorithms.

# 2.7 Random forests

Decision tree learning is a subtle affair. If one designs a DTLA by choosing a specific combination of the many design options, the tree that one gets from a training dataset D will be influenced by that choice of options. It will not be "the optimal" tree which one might obtain by some other choice of options. Furthermore, if one uses pruning or early stopping to fight overfitting, one might well end up with trees that, while not overfitting, are underfitting – that is, they do not exploit all the information that is in the training data. In summary, whatever one does, one is likely to obtain a decision tree that is significantly sub-optimal.

This is a common situation in machine learning. There are only very few machine learning techniques where one has full mathematical control over getting the best possible model from a given training dataset, and decision tree learning is not among them. Fortunately, there is also a common escape strategy for minimizing the quality deficit inherent in most learning designs: *ensemble methods*. The idea is to train a whole collection (called "ensemble") of models (here: trees), each of which is likely suboptimal (jargon: each model is obtained from a "weak learner"), but if their results on a test example are diligently combined, the merged result is much better than what one gets from each of the models in the ensemble. It's the idea of crowd intelligence.

"Ensemble methods" is an umbrella term for a wide range of techniques. They differ, obviously, in the kind of the individual models, like for instance decision trees or neural networks. Second, they vary in the methods of how one generates diversity in the ensemble. Ensemble methods work well only to the extent that the individual models in the ensemble probe different aspects in the training data – they should look at the data from different angles, so to speak. Thirdly, ensemble methods differ in the way how the results of the individual models are combined. The most common way is *majority voting* — the final classification decision is the

one made most often by the individual models. A general theory for setting up an ensemble learning scheme is not available – we are again thrown back to heuristics. The Wikipedia articles on "Ensemble learning" and "Ensemble averaging (machine learning)" give a condensed overview.

Obviously, ensemble methods can only be used when the computational cost of training a single model is rather low. This is the case for decision tree learning. Because training an individual tree will likely not give a competitive result but is cheap, it is common practice to train not a single decision tree but an entire ensemble – which in the case of tree learning is called a *random forest*. In fact, random forests can yield competitive results (for instance, compared to neural networks) at moderate cost, and are therefore often used in practical applications.

The definite reference for random forests is Breiman 2001. The paper has two parts. In the first part, Breiman gives a mathematical analysis of why combining decision trees does not lead to overfitting, and derives an instructive upper bound on the expected generalization error (= risk, see Section 2.3). These results were in synchrony with mainstream work in other areas of machine learning theory at the time and made this paper *the* theory anchor for random forests. However, I personally find the math notation used by Breiman opaque and hard to penetrate, and I am not sure how many readers could understand it. The second, larger part of this paper describes several variants and extensions of random forest algorithms, discusses their properties and benchmarks some of them against the leading classification learning algorithms of the time, with favourable outcomes. This part is an easy read and the presented algorithms are not difficult to implement. My hunch is that it is the second rather than the first part which led to the immense impact of this paper.

Here I give a summary of the paper, in reverse order, starting with the practical algorithm recommended by Breiman. After that I give an account of the most conspicuous theoretic results in transparent standard probability notation.

In ensemble learning one must construct many different models in an automated fashion. One way to achieve this is to employ a stochastic learning algorithm. A stochastic learning algorithm can be seen as a learning algorithm which takes two arguments. The first argument is the training data D, the same as in ordinary, non-stochastic learning algorithms. The second argument is a random vector, denoted by  $\Theta$  in Breiman's paper, which is set to different random values in each run of the algorithm.  $\Theta$  can be seen as a vector of control parameters in the algorithm; different random settings of these control parameters lead to different outcomes of the learning algorithm although the training data are always the same, namely D.

Breiman proposes two ways in order to make tree learning stochastic:

**Bagging.** In each run of the learning algorithm, the training dataset is resampled with replacement. That is, from D one creates a new training dataset D' of the same size as D by randomly copying elements from D into D'. This is a general approach for ensemble learning, called *bagging*. The Wikipedia

article on "Bootstrap aggregating" gives an easy introduction if you are interested.

**Random feature selection** is the term used by Breiman for a randomization technique where, at each node whose query property has to be chosen during tree growing, a small subset of all still unqueried properties is randomly drawn as candidates for the query property of this node. The winning property among them is determined by the information gain criterion.

Combining these two randomness-producing mechanisms with a number of other design decisions not mentioned here, Breiman obtains a stochastic learning algorithm which, with ensembles of size 100 and majority voting, outperformed other ML methods that were state-of-the-art at that time.

The random vector  $\Theta$ , which is needed for a concise specification and a mathematical analysis of the resulting stochastic tree learning algorithm, is an essentially arbitrary encoding of the random choices made for bagging and random feature selection.

The main theoretical result in Breiman's paper is his Theorem 2.3:

$$PE^* \le \bar{\varrho} \left(1 - s^2\right) / s^2.$$

This theorem certainly added much to the impact of the paper, because it gives an intuitive guidance for the proper design of random forests, and also because it connected random decision tree forests to other machine learning methods which were being mathematically explored at the time when the paper appeared. I will now try to give a purely intuitive explanation, and then conclude this section with a clean-math explanation of this theorem.

- The quantity *PE*<sup>\*</sup>, called *generalization error* by Breiman, is the probability that a random forest (in the limit of its size going to infinity) makes wrong decisions. Obviously one wants this probability to be as small as possible.
- The quantity  $\bar{\varrho}$  is a certain statistical correlation measure which measures how often, on average across the tree population in a forest, the various trees will arrive at similar classifications on test data points. This measure is maximal if all trees always yield the same classification results, and it is minimal if, roughly speaking, their classification results are statistically independent.
- The quantity s, which satisfies  $0 \le s \le 1$ , called by Breiman the strength of the stochastic tree learning algorithm, measures a certain "discerning power" of the average tree in a forest, that is, how large is the probability gap between an average tree making the right decision and the probability for the tree to make a wrong decision. s ranges in [0, 1] and if the strength is equal to the maximal value of 1, then all trees in the forest always make

correct decisions on all test data points. If it is zero, then the trees make more incorrect decisions than correct ones on average across the forest and test data points.

Note that the factor  $(1-s^2)/s^2$  ranges between zero and infinity and is monotonously increasing with decreasing s. It is zero with maximal strength s = 1, and it is infinite if the strength is zero.

The theorem gives an upper bound on the generalization error observed in (asymptotically infinitely large) random forests. The main message is that this bound is a product of a factor  $\bar{\varrho}$  which is smaller when the different trees in a forest vary more in their response to test inputs, with a factor  $(1-s^2)/s^2$  which is smaller when the trees in the forest place a larger probability gap between correct and the second most common decision across the forest. The suggestive message of all of this is that in designing a stochastic decision tree learning algorithm one should

- aim at maximizing the response variability across the forest, while
- attempting to ensure that trees mostly come out with correct decisions.

If one succeeds to generate trees that always give correct decisions, one has maximal strength s = 1 and the generalization error is obviously zero. This will usually be impossible. Instead, the stochastic tree learning algorithm will produce trees that have a residual error probability, that is, s < 1. Then the first factor implies that one should aim at a stochastic tree generation mechanism which (while fixing the strength) show great variability in their response behavior.

The remainder of this section is only for those of you who are familiar with probability theory, and this material will not be required in the final exam. I will give a mathematically transparent account of Breiman's Theorem 2.3. This boils down to an exercise in clean mathematical formalism. We start by formulating the ensemble learning scenario in rigorous probability terms. There are two probability spaces involved, one for generating data points, and the other for generating the random vectors  $\Theta$ . I will denote these two spaces by  $(\Omega, \mathfrak{A}, P)$  and  $(\Omega^*, \mathfrak{A}^*, P^*)$ respectively, with the second one used for generating  $\Theta$ . The elements of  $\Omega, \Omega^*$ will be denoted by  $\omega, \omega^*$ , respectively.

Let X be the RV which generates attribute vectors,

$$X: \Omega \to A_1 \times \ldots \times A_m,$$

and Y the RV which generates class labels,

$$Y: \Omega \to C.$$

The random parameter vectors  $\Theta$  can be vectors of any type, numerical or categorical, depending on the way that one chooses to parametrize the learning

algorithm. Let T denote the space from which  $\Theta$ 's can be picked. For a forest of K trees one needs K random vectors  $\Theta_j$ , where  $j = 1, \ldots, K$ . Let

$$Z_j: \Omega^* \to T$$

be the RV which generates the *j*-th parameter vector, that is,  $Z_j(\omega^*) = \Theta_j$ . The RVs  $Z_j$  (j = 1, ..., K) are independent and identically distributed (iid).

In the remainder of this section, we fix some training dataset D. Let  $h(\cdot, \Theta)$  denote the tree that is obtained by running the tree learning algorithm with parameter vector  $\Theta$ . This tree is a function which outputs a class decision if the input is an attribute vector, that is,

$$h(\mathbf{x}, \Theta) \in C.$$

When  $\Theta$  is fixed, we can view h as a function of the RV X, thus

$$h(X,\Theta):\Omega\to C$$

is a random variable.

Now the stage is prepared to re-state Breiman's central result (Theorem 2.3) in a transparent notation.

Breiman starts by introducing a function mr, called margin function for a random forest,

$$mr: (A_1 \times \ldots \times A_m) \times C \to \mathbb{R}$$
  
(**x**, y)  $\mapsto P^*(h(\mathbf{x}, Z) = y) - \max_{\tilde{y} \neq y} \{P^*(h(\mathbf{x}, Z) = \tilde{y})\},\$ 

where Z is a random variable distributed like the  $Z_j$ . Note that  $P^*(h(\mathbf{x}, Z) = y)$  is the probability that a randomly picked tree from the ensemble will classify  $\mathbf{x}$  correctly; and that  $\max_{\tilde{y}\neq y} \{P^*(h(\mathbf{x}, Z) = \tilde{y}) \text{ is the probability that a randomly picked tree from the ensemble will wrongly classify <math>\mathbf{x}$  as  $\tilde{y}$ , where  $\tilde{y}$  is the most frequent misclassification across the ensemble.

If you think about it you will see that  $mr(\mathbf{x}, y) > 0$  if and only if a very large forest (in the limit of its size going to infinity) will classify the data point  $(\mathbf{x}, y)$ correctly, and  $mr(\mathbf{x}, y) < 0$  if the forest will come to a wrong collective decision.

On the basis of this margin function, Breiman defines the generalization error  $PE^*$  by

$$PE^* = P(mr(X, Y) < 0).$$

The expectation of mr over data examples is

$$s = E[mr(X, Y)].$$

s is a measure to what extent, averaged over data point examples, the correct answer probability (over all possible decision trees) is larger than the highest

probability of deciding for any other answer. Breiman calls *s* the *strength* of the parametrized stochastic learning algorithm. The stronger the stochastic learning algorithm, the greater the probability margin between correct and wrong classifications on average over data point examples.

Breiman furthermore introduces a raw margin function  $rmg_{\Theta}$ , which is a function of X and Y parametrized by  $\Theta$ , through

$$rmg_{\Theta}(\omega) = \begin{cases} 1, & \text{if } h(X(\omega), \Theta) = Y(\omega), \\ -1, & \text{if } h(X(\omega), \Theta) \text{ gives the maximally probable among the} \\ & \text{wrong answers in an asymptotically infinitely large forest,} \\ 0, & \text{else.} \end{cases}$$
(10)

Define  $\rho(\Theta, \Theta')$  to be the correlation (= covariance normalized by division with standard deviations) between  $rmg_{\Theta}$  and  $rmg_{\Theta'}$ . For given  $\Theta, \Theta'$ , this is a number in [-1, 1]. Seen as a function of  $\Theta, \Theta', \rho(\Theta, \Theta')$  maps each pair  $\Theta, \Theta'$  into [-1, 1], which gives a RV which we denote with the same symbol  $\rho$  for convenience,

$$\varrho(Z, Z'): \Omega^* \to [-1, 1],$$

where Z, Z' are two independent RVs with the same distribution as the  $Z_j$ . Let

$$\bar{\varrho} = E[\varrho(Z, Z')]$$

be the expected value of  $\rho(Z, Z')$ . It measures to what extent, in average across random choices for  $\Theta, \Theta'$ , the resulting two trees  $h(\cdot, \Theta)$  and  $h(\cdot, \Theta')$  have both the same correct or wrong decision averaged over data examples.

And here is, finally, Breiman's Theorem 2.3:

$$PE^* \le \bar{\varrho} (1 - s^2)/s^2,$$
 (11)

whose intuitive message I discussed earlier.

# 3 Elementary supervised temporal learning

In this section I give an introduction to an elementary temporal data learning method. It combines simplicity with a very broad practical usefulness: using linear regression to learn a map which transforms temporal input signals to output signals. In many scenarios, this simple technique is all one needs. It can be programmed and executed in a few minutes (really!) and you should run this technique as a first baseline whenever you start a serious learning task that involves time series data.

This section deals with *numerical* timeseries where the data format for each time point is a real-valued scalar or vector. This includes the majority of all learning tasks that arise in the natural sciences, in engineering, robotics, speech and in video processing.

Methods for dealing with *symbolic* timeseries (in particular texts, but also discrete action sequences of intelligent agents / robots, DNA sequences and more) can be obtained by encoding symbols into numerical vectors and then apply numerical methods. This is the way taken by deep learning with neural networks, in particular large language models. However, there exist also methods that operate on symbol sequences directly: all kinds of discrete-state dynamical systems, deterministic or stochastic, like finite automata, Markov chains, hidden Markov models, dynamical Bayesian networks, and more. I will not consider such methods in this section.

My secret educational agenda in this section is that this will let you rehearse linear regression - which is such a basic, simple, yet widely useful method that everybody should have a totally 100% absolutely unshakeable secure solid firm hold on it.

#### 3.1 Recap: linear regression

A remark on notation: throughout these lecture notes, vectors are column vectors unless otherwise noted (some widely used special vectors are traditionally noted as row vectors – this holds specifically for the linear regression weight vector  $\mathbf{w}$  which we will be meeting in a minute). I use the prime ' to denote vector or matrix transpose.

I start with a generic rehearsal of linear regression, independent of temporal learning tasks. The linear regression task is specified as follows:

- **Given:** a collection  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  of N training data points, where  $\mathbf{x}_i \in \mathbb{R}^n$  and  $y_i \in \mathbb{R}$ .
- **Wanted:** a linear map from  $\mathbb{R}^n$  to  $\mathbb{R}$ , represented by a *regression weight vector* **w** (a row vector of size n), which solves the minimization problem

$$\mathbf{w} = \underset{\mathbf{w}^*}{\operatorname{argmin}} \sum_{i=1}^{N} (\mathbf{w}^* \mathbf{x}_i - y_i)^2.$$
(12)

In machine learning terminology, this is a supervised learning task, because the training data points include teacher outputs  $y_i$ . Note that the abstract data format  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  is the same as for decision trees, but here the data are numerical while for decision trees they are most often symbolic.

By a small modification one can make linear regression much more versatile. Note that any weight vector  $\mathbf{w}^*$  in (12) will map the zero vector  $\mathbf{0} \in \mathbb{R}^n$  on zero. This is often not what one wants to happen. If one enriches (12) by also training a bias  $b \in \mathbb{R}$ , via

$$(\mathbf{w}, b) = \underset{\mathbf{w}^{*}, b^{*}}{\operatorname{argmin}} \sum_{i=1}^{N} (\mathbf{w}^{*} \mathbf{x}_{i} + b^{*} - y_{i})^{2},$$

one obtains an *affine* linear function (linear plus constant offset). The common way to set up linear regression such that affine linear solutions become possible is to pad the original input vectors  $\mathbf{x}_i$  with a last component of constant size 1, that is, in (12) one uses n + 1-dimensional vectors  $[\mathbf{x}; 1]$  (using Matlab notation for the vertical concatentation of vectors). Then a solution of (12) on the basis of the padded input vectors will be a regression weight vector  $[\mathbf{w}, b] \in \mathbb{R}^{n+1}$ , with the last component b giving the offset.

We now derive a solution formula for the minimization problem (12). Most textbooks start from the observation that the objective function  $\sum_{i=1}^{N} (\mathbf{w}\mathbf{x}_i - y_i)^2$  is a quadratic function in the weights  $\mathbf{w}$ , and then use calculus to find the minimum of this quadratic function by setting its partial derivatives to zero. I will present another derivation which does not need calculus and reveals the underlying geometry of the problem more clearly.

Let  $\mathbf{x}_i = (x_i^1, \ldots, x_i^n)'$  be the *i*th input vector. The key to understand linear regression is to realize that the N values  $x_1^j, \ldots, x_N^j$  of the *j*-the component  $(j = 1, \ldots, n)$ , collected across all input vectors, make for an N-dimensional vector  $\varphi_j = (x_1^j, \ldots, x_N^j)' \in \mathbb{R}^N$ . Similarly, the N target values  $y_1, \ldots, y_N$  can be combined into an N-dimensional vector y. Figure 10 **Top** shows a case with N = 10 input vectors of dimension n = 4.

Using these N-dimensional vectors as a point of departure, geometric insight gives us a nice clue how **w** should be computed. To admit a visualization, we consider a case where we have only N = 3 input vectors that each have n = 2components. This gives two 3-dimensional vectors  $\varphi_1, \varphi_2$  (Figure 10 **Bottom**). The target values  $y_1, y_2, y_3$  are combined in a 3-dimensional vector y.

Notice that in machine learning, one should best have more input vectors than the input vectors have components, that is, N > n. In fact, a very coarse but not very stupid rule of thumb – with many exceptions – says that one should aim at N > 10 n (if this is not warranted, use unsupervised dimension reduction methods to reduce n). We will thus assume that we have fewer vectors  $\varphi_j$  than training data points. The vectors  $\varphi_j$  thus span an n-dimensional subspace in  $\mathbb{R}^N$  (greenish shaded area in Figure 10 **Bottom**).

Notice (easy exercise, do it!) that the minimization problem (12) is equivalent



Figure 10: Two visualizations of linear regression. **Top.** This visualization shows a case where there are N = 10 input vectors  $\mathbf{x}_i$ , each one having n = 4 vector components  $x_i^1, \ldots, x_i^4$  (green circles). The fourth component is a constant-1 bias input. The ten values  $x_1^j, \ldots, x_{10}^j$  of the *j*-th component (where  $j = 1, \ldots, 4$ ) form a 10-dimensional (row) vector  $\varphi_i$ , indicated by a green connecting line. Similarly, the ten target values  $y_i$  give a 10-dimensional vector y (shown in red). The linear combination  $y_{\text{opt}} = \mathbf{w}[\varphi_1; \varphi_2; \varphi_3; \varphi_4]$ , which gives the best approximation to y in the least mean square error sense, is shown in orange. Bottom. The diagram shows a case where the input vector dimension is n = 2 and there are N = 3input vectors  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$  in the training set. The three values  $x_1^1, x_2^1, x_3^1$  of the first component give a three dimensional vector  $\varphi_1$ , and the three values of the second component give  $\varphi_2$  (green). These two vectors span a 2-dimensional subspace  $\mathcal{F}$ in  $\mathbb{R}^N = \mathbb{R}^3$ , shown in green shading. The three target values  $y_1, y_2, y_3$  similarly make for a vector y (red). The linear combination  $y_{\text{opt}} = w_1 \varphi_1 + w_2 \varphi_2$  which has the smallest distance to y is given by the projection of y on this plane  $\mathcal{F}$  (orange). The vectors  $u_1, u_2$  shown in blue is a pair of orthonormal basis vectors which span the same subspace  $\mathcal{F}$ .

 $\mathrm{to}$ 

$$\mathbf{w} = \underset{\mathbf{w}^*}{\operatorname{argmin}} \| (\sum_{j=1}^n w_j^* \varphi_j) - y \|^2,$$
(13)

where  $\mathbf{w}^* = (w_1^*, \ldots, w_n^*)$ . We take another look at Figure 10 **Bottom**. Equation 13 tells us that we have to find the linear combination  $\sum_{j=1}^n w_j \varphi_j$  which comes closest to y. Any linear combination  $\sum_{j=1}^n w_j^* \varphi_j$  is a vector which lies in the linear subspace  $\mathcal{F}$  spanned by  $\varphi_1, \ldots, \varphi_n$  (shaded area in the figure). The linear combination that is closest to y apparently is the projection of y on that subspace. This is the essence of linear regression!

All that remains is to compute which linear combination of  $\varphi_1, \ldots, \varphi_n$  is equal to the projection of y on  $\mathcal{F}$ . Let us call this projection  $y_{\text{opt}}$ . We may assume that the vectors  $\varphi_1, \ldots, \varphi_n$  are linearly independent. If they would be linearly dependent, we could drop as many from them as is needed to reach a linearly independent set.

The rest is mechanical linear algebra.

Let  $X = (\varphi_1, \ldots, \varphi_n)'$  be the  $n \times N$  sized matrix whose rows are formed by the  $\varphi'_j$  (and whose colums are the  $\mathbf{x}_i$ ). Then X'X is a positive semi-definite matrix of size  $N \times N$  with a singular value decomposition  $X'X = U_N \Sigma_N U'_N$ . Since the rank of X is n < N, only the first n singular values in  $\Sigma_N$  are nonzero. Let  $U = (u_1, \ldots, u_n)$  be the  $N \times n$  matrix made from the first n columns in  $U_N$ , and let  $\Sigma$  be the  $n \times n$  diagonal matrix containing the n nonzero singular values of  $\Sigma_N$  on its diagonal. Then

$$X'X = U\Sigma U'. \tag{14}$$

This is sometimes called the *compact* SVD. Notice that the columns  $u_j$  of U form an orthonormal basis of  $\mathcal{F}$  (blue arrows in Figure 10 **Bottom**).

Using the coordinate system given by  $u_1, \ldots, u_n$ , we can rewrite each  $\varphi_j$  as

$$\varphi_j = \sum_{l=1}^n (u_l' \varphi_j) \ u_l = U U' \varphi_j.$$
(15)

Similarly, the projection  $y_{\text{opt}}$  of y on  $\mathcal{F}$  is

$$y_{\text{opt}} = \sum_{l=1}^{n} (u'_l y) \ u_l = UU'y, \tag{16}$$

But also  $y_{\text{opt}} = X' \mathbf{w}'$ . From (15) we get X' = UU'X', which in combination with (16) turns  $y_{\text{opt}} = X' \mathbf{w}'$  into

$$UU' y = UU'X' \mathbf{w}'$$

A weight vector  $\mathbf{w}$  solves this equation if it solves

$$U'y = U'X'\mathbf{w}'.\tag{17}$$

It remains to find a weight vector  $\mathbf{w}$  which satisfies (17). I claim that  $\mathbf{w}' = (X X')^{-1} X y$  does the trick, that is,  $U' y = U' X' (X X')^{-1} X y$  holds.

To see this, first observe that XX' is nonsingular, thus  $(XX')^{-1}$  is defined. Furthermore, observe that U'y and  $U'X'(XX')^{-1}Xy$  are *n*-dimensional vectors, and that the  $N \times n$  matrix  $U\Sigma$  has rank *n*. Therefore,

$$U'y = U'X'(XX')^{-1}Xy \iff U\Sigma U'y = U\Sigma U'X'(XX')^{-1}Xy.$$
(18)

Replacing  $U\Sigma U'$  by X'X (compare Equation 14) turns the right equation in (18) into  $X'Xy = X'XX'(XX')^{-1}Xy$ , which is obviously true. Therefore,  $\mathbf{w}' = (XX')^{-1}Xy$  solves our optimization problem (13).

We summarize our findings:

**Data.** A set  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  of *n*-dimensional input vectors  $\mathbf{x}_i$  and scalar targets  $y_i$ .

**Wanted.** An *n*-dimensional row weight vector  $\mathbf{w}$  which solves the linear regression objective from Equation 12.

**Step 1.** Sort the input vectors as columns into an  $n \times N$  matrix X and the targets into an N-dimensional vector y.

Step 2. Compute the (transpose of the) result by

$$\mathbf{w}' = (XX')^{-1} X y. \tag{19}$$

Some further remarks:

• What we have derived here generalizes easily to cases where the data are of the form  $(\mathbf{x}_i, \mathbf{y}_i)_{i=1,\dots,N}$  where  $\mathbf{x}_i \in \mathbb{R}^n, \mathbf{y}_i \in \mathbb{R}^k$ . That is, the output data are vectors, not scalars. The objective is to find a  $k \times n$  regression weight matrix W which solves

$$W = \underset{W^*}{\operatorname{argmin}} \ \sum_{i=1}^{N} \|W^* \mathbf{x}_i - \mathbf{y}_i\|^2.$$
(20)

The solution is given by

$$W' = (XX')^{-1} X Y,$$

where Y is the  $N \times k$  matrix that contains the  $\mathbf{y}'_i$  in its rows.

• For an  $a \times b$  sized matrix A, where  $a \ge b$  and A has rank b, the matrix  $A^+ := (A'A)^{-1} A'$  is called the (left) *pseudo-inverse* of A. It satisfies  $A^+ A = I_{b \times b}$ . It is often also written as  $A^{\dagger}$ .

• Computing the inversion  $(XX')^{-1}$  may suffer from numerical instability when XX' is close to singular. Remark: this happens more often than you would think - in fact, XX' matrices obtained from real-world, highdimensional data are *often* ill-conditioned (= close to singular). You should always feel uneasy when your program code contains a matrix inverse! A quick fix is to add a small multiple of the  $n \times n$  identity matrix before inverting, that is, replace (19) by

$$\mathbf{w}'_{\text{opt}} = (XX' + \alpha^2 I_{n \times n})^{-1} X y.$$
(21)

This is called *ridge regression*. We will see later in this course that ridge regression not only helps to circumvent numerical issues, but also offers a solution to the problem of overfitting.

• A note on terminology. Here we have described *linear* regression. The word "regression" is used in much more general scenarios. The general setting goes like this:

**Given:** Training data  $(\mathbf{x}_i, \mathbf{y}_i)_{i=1,...,N}$ , where  $\mathbf{x}_i \in \mathbb{R}^n, \mathbf{y}_i \in \mathbb{R}^k$ . **Also given:** a *search space* H containing candidate functions  $h : \mathbb{R}^n \to \mathbb{R}^k$ . **Also given:** a loss function  $L : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^{\geq 0}$ .

Wanted: A solution to the optimization problem

$$h_{\text{opt}} = \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{N} L(h(\mathbf{x}_i), \mathbf{y}_i)$$

In the case of linear regression, the search space  $\mathcal{H}$  consists of all linear functions from  $\mathbb{R}^n$  to  $\mathbb{R}^k$ , that is, it consists of all  $k \times n$  matrices. The loss function is the *quadratic loss* which you see in (20). When one speaks of linear regression, the use of the quadratic loss is implied.

Search spaces  $\mathcal{H}$  can be arbitrarily large and rich in modeling options – for instance,  $\mathcal{H}$  might be the space of all deep neural networks of a given structure and size.

Classification tasks look similar to regression tasks at first sight: training data there have the format  $(\mathbf{x}_i, c_i)_{i=1,\dots,N}$ . The difference is that that the target values  $c_i$  are not numerical but symbolic — they are class labels.

## 3.2 Temporal learning tasks

There are a number of standard learning tasks which are defined on the basis of numerical timeseries data. Those tasks involve training data consisting of

an input signal  $(\mathbf{u}(t))_{t\in\mathcal{T}}$ , where  $\mathcal{T}$  is an *ordered* set of time points and for every  $t\in\mathcal{T}, \mathbf{u}(t)\in\mathbb{R}^k$ ;

a "teacher" output signal  $(\mathbf{y}(t))_{t \in \mathcal{T}}$ , where  $\mathbf{y}(t) \in \mathbb{R}^m$ .

The two most common 'timeline' sets are  $\mathbb{R}$  (the real line for measuring continuous physical time) and  $\mathbb{Z}$  (integer timesteps, called 'discrete time', appropriate when data come in in regular intervals, which is the common situation for digitally sampled data), and intervals within these timeline sets. For instance,  $\mathcal{T} = \{0, 1, \ldots, T\}$  is a discrete-time, finite timeline set.

The learning task consists in training a system which operates in time and, if it is fed with the input signal  $\mathbf{u}(t)$ , produces an output signal  $\hat{\mathbf{y}}(t)$  which approximates the teacher signal  $\mathbf{y}(t)$  (in the sense of minimizing a loss  $L(\hat{\mathbf{y}}(t) - \mathbf{y}(t))$  in time average over the training data). A few examples for illustration:

- Input signal: an ECG (electro cardiogram) signal; desired output: a signal which is constant zero as long as the ECG is normal, and jumps to 1 if there are irregularities in the ECG.
- Input signal: room temperature measurements from a thermometer. Output: value 0 as long as room temperature input signal is above certain threshold, value 1 if it is below the threshold (this is the switching behavior of a thermostat, albeit a badly designed one what is missing to make this thermostat work nicely?).
- Input signal: weather data (condensed to very large vectors  $\mathbf{u}(t)$ ) obtained from weather monitoring stations, airplane and ship sensors, satellite observations of the atmosphere. Desired output signal: a vector y(t + 24h) characterizing the atmosphere 24 hours in the future (that is, perform weather forecasting).
- Input signal: a noisy radio signal with lots of statics and echos. Desired output signal: the input signal in a version which has been de-noised and where echos have been cancelled.
- Input signal: a speech signal in English. Desired output: a speech signal in Dutch.

These are quite different sorts of tasks. The ECG monitoring task would be called a *temporal classification* or *fault monitoring* task; the 0-1 switching of the badly designed thermostat is too simplistic to have a respectable name; radio engineers would speak of *de-noising* and *equalizing*, the weather forecasting is a *timeseries prediction* task, and speech translation is too royally complex to have a name besides "online speech translation". Interesting and important input-output signal transformation tasks there are as many as there are wonders under the sun.

In many cases, one may assume that the current output data point  $\mathbf{y}(t_0)$  only depends on inputs up to that time point, that is, on the input history  $(\mathbf{u}(t))_{t \leq t_0}$ . Specifically,  $\mathbf{y}(t)$  does *not* depend on future inputs. Input-output systems where

the output does not depend on future inputs are called *causal* systems in the signal processing world.

A causal system can be said to have *memory* if the current output  $\mathbf{y}(t)$  is not fully determined by the current input  $\mathbf{u}(t)$ , but is influenced by earlier inputs as well. I must leave the meaning of "influenced by" vague at this point; we will make it precise in a later section when we investigate stochastic processes in more detail. All examples except the (poorly designed) thermostat example have memory.

Often, the output  $\mathbf{y}(t)$  is influenced by long-ago input only to a negligeable extent, and it can be explained very well from only the input history extending back to a certain limited duration. All examples in the list above except the English translation task have such limited relevant memory spans. In causal, discrete-time systems with bounded memory span, the current output  $\mathbf{y}(t)$  thus depends on an input window  $\mathbf{u}(t-d+1), \mathbf{u}(t-d+2), \ldots, \mathbf{u}(t-1), \mathbf{u}(t)$  of d steps duration. Figure 11 (top) gives an impression.

In this situation, learning an input-output system is a regression learning task. One has to find a regression function f of the kind

$$f: (\mathbb{R}^k)^d \to \mathbb{R}^m \tag{22}$$

which minimizes a chosen loss function on average over the training data time points.

The most convenient option is to go for the quadratic loss, which turns the input-output timeseries learning task into a case of linear regression.

Note that while in Section 3.1 I assumed that the input data points for linear regression were vectors, in (22) they are  $k \times d$ -dimensional matrices. This is not a problem; all one has to do is to *flatten* the collection of d k-dimensional input vectors which lie in a window into a single  $d \cdot k$  dimensional vector, and then apply (20) as before.

Linear regression is often surprisingly accurate, especially when one uses large windows and a careful regularization (to be discussed later in this course) through ridge regression. When confronted with a new supervised temporal learning task, the first thing a seasoned pro does, is to run it through the machinery of windowbased linear regression. This takes a few minutes of writing code and some seconds of computing time and gives, at the very least, a baseline for comparing more sophisticated methods against — and often it gives even already a very good soluation already without more effort.

Fun fact: All the signal processing in your smartphones, which transforms the noisy, echo-replete antenna signal into a series of clean 0's and 1's, is done with linear regression algorithms — highly speed-optimized ones on specialized hardware for that matter, because the input window of the raw antenna signal easily has a dimension of tens of thousands of timesteps (you get such large vectors from digitally sampling the raw analog signal at Gigahertz frequency for a few milliseconds) and the updates  $\mathbf{y}(t) \rightarrow \mathbf{y}(t+1)$  must be done at Gigahertz frequency too at early processing stages close to the antenna. These algorithms approximately recompute



Figure 11: The principle of temporal system learning by window-based regression. The case of scalar input and output signals is shown. **Top:** an input signal u(t) (blue crosses) is transformed to an output signal y(t) (orange). y(t) here depends only on the current and the four preceding inputs. **Bottom:** The special case of timeseries prediction (single step prediction, y(t) = u(t + 1)). The memory window size in both diagrams is d = 5.

the weight vector  $\mathbf{w}_{opt}$  every few milliseconds (!) because when the smartphone is moving (as you walk or drive), the characteristics of the signal noise and echoes changes, and the de-noising and equilibration filtering must be recalibrated quickly all the time. Designing such very fast (approximate) algorithms to solve (21) is the core challenge for an entire branch of signal processing, called *adaptive online* signal processing, and fills textbooks like that of Farhang-Boroujeny 1998.

But, linear regression only can give linear regression functions. This is not good enough if the dynamical input-output system behavior has significant nonlinear components. Then one must find a nonlinear regression function f.

If that occurs, one can take resort to a simple method which yields nonlinear regression functions while not renouncing the conveniences of the basic linear regression learning formula (19). I discuss this for the case of scalar inputs  $u(t) \in \mathbb{R}$ . The trick is to add fixed nonlinear transforms to the collection of input arguments

 $u(t-d+1), u(t-d+2), \ldots, u(t)$ . A common choice is to add polynomials. To make notation easier, let us rename  $u(t-d+1), u(t-d+2), \ldots, u(t)$  to  $u_1, u_2, \ldots, u_d$ . If one adds all polynomials of degree 2, one obtains a collection of d + d(d+1)/2input components for the regression, namely

$$\{u_1, u_2, \dots, u_d\} \cup \{u_i u_j \mid 1 \le i \le j \le d\}.$$

If one wants even more nonlinearity, one can add further, higher-order polynomials. The idea to approximate nonlinear regression functions by linear combinations of polynomial terms is a classical technique in signal processing, where it is treated under the name of *Volterra expansion* or *Volterra series*. Very general classes of nonlinear regression functions can be approximated to arbitrary degrees of precision with Volterra expansions.

Adding increasingly higher-order terms to a Volterra expansion obviously leads to a combinatorial explosion. Thus one will have to use some pruning scheme to keep only those polynomial terms which lead to an increase of accuracy. There is a substantial literature in signal processing dealing with pruning strategies for Volterra series (google "Volterra pruning"). I personally would never try to use polynomials of degree higher than 2. If that doesn't give satisfactory results, I would switch to other modeling techniques, using neural networks for instance.

#### 3.3 Time series prediction tasks

Looking into the future can make you rich, or powerful, or satisfy your curiosity – these motivations are deeply rooted in human nature and thus it is no wonder that time series prediction is an important task for machine learning.

Time series prediction tasks come in many variants and I will not attempt to draw the large picture but restrict this treatment to timeseries with integer timesteps and vector-valued observations, as in the previous subsections.

Re-using terminology and notation, the input signal in the training data is, as before,  $(\mathbf{u}(t))_{t\in\mathcal{T}}$ . If one wants to predict this sequence of observations h timesteps ahead (h is called *prediction horizon*), the desired output  $\mathbf{y}(t)$  is just the input, shifted by h timesteps:

$$(\mathbf{y}(t))_{t\in\mathcal{T}} = (\mathbf{u}(t+h))_{t\in\mathcal{T}}.$$

A little technical glitch is that due to the timeshift h, the last h data points in the input signal  $(\mathbf{u}(t))_{t\in\mathcal{T}}$  cannot be used as training data because their h-step future would lie beyond the maximal time T.

Framed as an  $\mathbf{u}(t)$ -to- $\mathbf{y}(t)$  input-output signal transformation task, all methods that can be applied for the latter can be used for timeseries prediction too. Specifically, simple window-based linear regression (as in Figure 11 bottom) is again a highly recommendable first choice for getting a baseline predictor whenever you face a new timeseries prediction problem with numerical timeseries.

## 3.4 Signal-based vs. state-based timeseries modeling

In windows-based methods one tries to find some function that directly transforms the input signal (and parts of its recent history) to the desired output signal. If one starts thinking about it, it seems that in some cases this should be all but impossible.

Many temporal learning tasks concern the input-output modeling of a physical *dynamical system*. The input signal is not (somehow magically) directly transformed into the output signal, but there sits a complex intermediary system between the input and output signals which "transduces" the input signal and while it does so, heavily transforms it and may add information that comes from within the intermediary system. Just two examples:

- A radio reporter commenting a soccer game. Output of this reporter system: the reporter's speech signal. Input: what the reporter sees on the soccer field. Dynamical system in the middel: the reporter, especially his/her brain. – Two different reporters would comment the game differently, even if they see it from the same reporters' stand. Much of the specific information in the reporter's radio speech is not explainable by the raw soccer visual input, but arises from the reporter's specific knowledge and emotional condition. – Fun fact: attempting to developing machine learning methods for the automated commenting of soccer matches had been, for a decade or so around 1985, one of the leading research challenges in Germany's then still young AI research landscape. This received very heavy public funding, but the outcome was disappointing: the task was too difficult for the modeling methods available at that time. It still is... as of the time of writing this (November 2023) ... though considering the explosive progress in deep learning, maybe in a month (December 2023) it's solved. Or maybe not.
- Input signal: the radiowaves emitted from a transmitting antenna. Output signal: the radiowaves picked up by a receiving antenna. Physical system "in the middle": the physical world between the two antennas, inducing all kinds of static noise, echos, distortions, such that the received radio signal is a highly corrupted version of the cleanly transmitted one. Signal processing engineers call this part of the world-in-the-middle the *channel*. Does it not seem hopeless to model the input-output transformation of such super-complex physical channels? There can hardly be a more classical problem than this one; analysing signal transmission channels gave rise to Shannon's information theory (Shannon 1948).

Abstracting from these examples, consider how natural scientists and mathematicians describe dynamical systems with input and output. Here let us consider only discrete-time models with a time index set is  $T = \mathbb{N}$  or  $T = \mathbb{Z}$ . Three timeseries need to be taken care of simultaneously:

• an input signal  $\mathbf{u}(t)$ , where  $\mathbf{u}(t) \in \mathbb{R}^k$  (as before),

- an output signal  $\mathbf{y}(t)$ , where  $\mathbf{y}(t) \in \mathbb{R}^m$  (as before),
- a dynamical system state sequence  $\mathbf{x}(t)$ , where  $\mathbf{x}(t) \in \mathbb{R}^n$  (this is new).

In soccer reporting example,  $\mathbf{x}(t)$  would refer to some kind of brain state for instance, the vector of activations of all the reporter's brain's neurons. In the signal transmission example,  $\mathbf{x}(t)$  would be the state vector of some model of the physical world stretched out between the sending and receiving antenna.

A note in passing: variable naming conventions are a little confusing. In the machine learning literature, x's and y's in  $(\mathbf{x}, \mathbf{y})$  pairs usually mean the arguments and values (or inputs and outputs) of classification/regression functions. Both  $\mathbf{x}$  and  $\mathbf{y}$  are observed data. In the dynamical systems and signal processing literature (mathematics, physics and engineering), the variable name x typically is reserved for the state of a physical system that generates or "channels" (= "transduces", "filters") signals. The internal physical state of these systems is normally not fully observable and not part of training data. Only the input and output signals  $\mathbf{u}(t), \mathbf{y}(t)$  are observable data which are available for training models.

In a discrete-time setting, the temporal evolution of  $\mathbf{u}(t), \mathbf{x}(t), \mathbf{y}(t)$  is governed by two functions, the *state update map* 

$$\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{u}(t+1)), \tag{23}$$

which describes how the internal states  $\mathbf{x}(t)$  develop over time under the influence of input, and the *observation function* 

$$\mathbf{y}(t) = g(\mathbf{x}(t)),\tag{24}$$

which describes which outputs  $\mathbf{y}(t)$  can be observed when the physical system is in state  $\mathbf{x}(t)$ .

The input signal  $\mathbf{u}(t)$  is not specified by any equation, it is "just given".

Figure 12 visualizes the structural difference between the signal-based and the state-based input-output transformation models.

There are many other types of state update maps and observation functions, for instance ODEs and PDEs for continuous-time systems, automata models for discrete-state systems, or a host of probabilistic formalisms for random dynamical systems. For our present discussion, considering only discrete-time state update maps is good enough.

A core difference between signal-based and state-based timeseries transformations is the achievable memory timespans. In windowed signal transformations through regression functions, the memory depth is bounded by the window length. In contrast, the dynamical system state  $\mathbf{x}(t)$  of the intermediary system is potentially co-determined by input that was fed to the dynamical system in an arbitrary deep past – the memory span can be unbounded! This may seem counterintuitive if one looks at Figure 12 because at each time point t, only the input data point  $\mathbf{u}(t)$  from that same timepoint is fed to the dynamical system. But  $\mathbf{u}(t)$  leaves



Figure 12: Contrasting signal-based with state-based timeseries transformations. **Top:** Window-based determination of output signal through a regression function h (here the window size is 3). **Bottom:** Output signal generation by driving in intermediary dynamical system with an input signal. The state  $\mathbf{x}(t)$  is updated by f.

some trace on the state  $\mathbf{x}(t)$ , and this effect is forwarded to the next timestep through f, thus  $\mathbf{x}(t+1)$  is affected by  $\mathbf{u}(t)$ , too; and so forth. Thus, if one expects long-range or even unbounded memory effects, using state-based transformation models is often the best way to go.

Machine learning offers a variety of state-based models for timeseries transformations, together with learning algorithms. The most powerful ones are hidden Markov models (which we'll get to know in this course) and other dynamical graphical models (which we will not touch), and recurrent neural networks (which we'll briefly meet I hope).

In some applications it is important that the input-output transformation can be learnt in an *online adaptive* fashion. The input-output transformation is not trained just once, on the basis on a given, fixed training dataset. Instead, training never ends; while the system is being used, it continues to adapt to *changing* input-output relationships in the data that it processes. My favourite example is the online adaptive filtering (denoising, echo cancellation) of the radio signal received by a mobile phone. As I already mentioned, the denoising, echo-cancelling filter has to be re-learnt every few milliseconds. This is done with a window-based linear regression (window size several tens of thousands) and ingeniously simplified/accelerated algorithms. Because this is powerful stuff, we machine learners should not leave these admirable methods only to the electrical engineers (who invented them) but learn to use them ourselves. I will devote a session later in this course to these online adaptive signal processing methods.

#### 3.5 Takens' theorem

From what I just explained it may seem that signal-based and state-based inputoutput transformations are two quite different things. However, surprising connections exist. In certain ways and for certain systems, the two even coincide. This was discovered by the Dutch mathematician Floris Takens (1940-2010). His celebrated theorem (Takens 1981), now universally called *Takens Theorem*, is deep and beautiful, and enabled novel data analysis methods which triggered a flood of work in the study of complex system dynamics (in all fields biological, physical, metereological, economical, social, engineering). Also, Takens was a professor in Groningen, and founded a tradition of dynamical systems theory research in our mathematics department. Plus, finally, Takens-like theorems have recently been used to analyse why/how certain recurrent neural networks of the "reservoir computing" brand function so surprisingly well in machine learning (A. Hart, Hook, and Dawes 2020). These are all good enough causes for me to end this section with an intuitive explanation of Takens theorem, although this is not normally a part of a machine learning course.

Since the following contains allusions to mathematical concepts which I cannot assume are known by all course participants, the material in this section will not be tested in exams.

Floris Takens' original theorem was formulated in a context of continuous-time dynamics governed by differential equations. Many variants and extensions of Takens theorem are known today. To stay in tune with earlier parts of this section I present a discrete-time version. Consider the input-free dynamical system

$$\mathbf{x}(t+1) = f(\mathbf{x}(t)), \qquad (25)$$
$$y(t) = g(\mathbf{x}(t)),$$

where  $\mathbf{x}(t) \in \mathbb{R}^n$  and  $y(t) \in \mathbb{R}$  (a one-dimensional output). The background intuition behind this set-up is that  $\mathbf{x}$  models some physical dynamical system possibly quite high-dimensional, for instance  $\mathbf{x}$  being a brain state vector — while y is a physical quantity that is "measured", or "observed", from  $\mathbf{x}$  via an output function g.

If the state update dynamics (25) is run for a long time, in many systems the state vector sequence  $\mathbf{x}(t)$  will ultimately become confined to an *attracting* subset  $A \subset \mathbb{R}^n$  of the embedding space  $\mathbb{R}^n$ . In some cases, this subset will be a low-dimensional manifold of dimension m; in other cases ("chaotic attractors") it will have a fractal geometry which is characterized by a *fractal dimension*, a real number which I will likewise call m and which typically is also small compared to the embedding space dimension n. Now let us look at the observation sequence y(t) recorded when the system has "relaxed" to the attracting subset. It is a scalar timeseries. For any k > 1, this scalar timeseries can be turned into a k-dimensional timeseries  $\mathbf{y}(t)$  by delay embedding, as follows. Choose a delay of  $\delta$  timesteps. Then set  $\mathbf{y}(t) = (y(t), y(t - \delta), y(t - 2\delta), \dots, y(t - (k - 1)\delta)'$ . That is, just stack a few consecutive observations (spaced in time by  $\delta$ ) on top of each other. This gives a sequence of k-dimensional vectors  $\mathbf{y}(t)$ . You will recognize them as what we called "observation windows" earlier in this section.

Takens-type theorems now state that, under conditions specific to the particular version of the theorem, the geometry of the dynamics of the delay-embedded vectors is the same as the geometry of the dynamics of the original system, up to a smooth bijective transformation between the two geometries. I think a picture is really helpful here! Figure 13 gives a visual demo of Takens theorem at work.



Figure 13: Takens theorem visualized. Left plot shows the Lorenz attractor, a chaotic attractor with a state sequence  $\mathbf{x}(t)$  defined in an n = 3 dimensional state space. The center plot shows a 1-dimensional observation y(t) thereof. The right plot (orange) shows the state sequence  $\mathbf{y}(t)$  obtained from y(t) by three-dimensional delay embedding (I forget which delay  $\delta$  I used to generate this plot).

When I just said, "the geometry ... is the same ... up to smooth bijective transformation", I mean the following. Imagine that in Figure 13 (left) the blue trajectory lines were spun into a transparent 3-dimensional substrate which has rubber-like qualities. Then, by pushing and pulling and shearing this substrate (without rupturing it), the blue lines would take on new positions in 3D space — until they would *exactly* coincide with the orange ones in the right panel.

When two state trajectory plots can be transformed into each other by such a "rubber-sheet transformation" (a term used in dynamical systems textbooks), some geometric characteristics are identically preserved between the two plots (as in the the left and the right panel in the figure). Specifically, the so-called Lyapunov exponents, which are indicators for the stability (noise robustness) of the dynamics, and the (possibly fractal) dimension of the attracting set A are identical in the original dynamics and its delay-embedded reconstruction. The deep, nontrivial message behind Takens-like theorems is that one can trade time for space in certain dynamical systems. If by "space" one means dimensions of state vectors, and by "time" lengths of observation windows of a scalar observable, Takens-like theorems state that if the *spatial* dimension of the original attractor set (manifold or fractal) is m, then using a delay embedding of *temporal* window length larger than 2m will always recuperate the original geometry. Just for completeness: the fractal dimension of the Lorenz attractor is about m = 2.06, thus any delay embedding dimension exceeding 2m + 1 = 5.12 would certainly be sufficient for reconstruction from a scalar observation; it turns out in Figure 13 that an embedding dimension of 3 is already working.

In timeseries prediction tasks in machine learning, Takens theorem gives a justification why in deterministic dynamical systems without input it should be possible to predict the future of a scalar timeseries y(t) from its past. If the timeseries in question has been observed from a system like (25), Takens theorem establishes that the information contained in the last 2m + 1 timesteps before the prediction point t comprises the full information about the state of the system that generated y(t) and thus knowledge of the last 2m + 1 timesteps is enough to predict y(t + 1) with absolute accuracy.



Figure 14: Getting nice graphics from delay embeddings. Left: a timeseries recorded from the "Mackey-Glass" chaotic attractor. Right: plotting the trajectory of a delay-embedded version of the left signal.

Even if you forget about the Takens theory background, it is good to be familiar with delay-embeddings of scalar timeseries, because they help creating instructive graphics. If you transform a one-dimensional scalar timeseries y(t) into a 2-dimensional one by a 2-element delay embedding, plotting this 2-dimensional trajectory  $\mathbf{y}(t) = y(t - \delta), y(t)$  in its delay coordinates  $(y(t - \delta), y(t))$  will often give revealing insight into the structure of the timeseries – our visual system is optimized for seeing patterns in 2-D images, not in 1-dimensional timeseries plots. See Figure 14 for a demo.

# 4 Basic methods for dimension reduction

One way to take up the fight with the "curse of dimensionality", which I highlighted in Section 1.2.2, is to reduce the dimensionality of the raw input data before they are fed to subsequent learning algorithms. The dimension reduction ratio can be enormous.

In this Section I will introduce two standard, elementary methods for dimension reduction: K-means clustering and principal component analysis (PCA); and a method which is not often used in practice except sometimes for data visualization, but which is very possibly used by our brains: self-organizing feature maps. All three methods reduce the dimension of vector data, which come in the form of points  $\mathbf{x} \in \mathbb{R}^n$ , that is, this section is only about dimension reduction methods for numerical data. Dimension reduction is the archetypical unsupervised learning task.

### 4.1 Set-up, terminology, general remarks

We are given a family  $(\mathbf{x}_i)_{i=1,...,N}$  of "raw" data points, where  $\mathbf{x}_i \in \mathbb{R}^n$ . The goal of dimension reduction methods is to compute from each (high-dimensional) "raw" data vecor  $\mathbf{x}_i$  a (low-dimensional) vector  $\mathbf{f}(\mathbf{x}_i) \in \mathbb{R}^m$ , such that

- m < n, that is, we reduce the number of dimensions maybe even dramatically;
- the low-dimensional vectors  $\mathbf{f}(\mathbf{x}_i)$  should preserve from  $\mathbf{x}_i$  the specific information that is needed to solve the learning task that comes after this dimension-reducing data "preprocessing".

Terminology: The *m* component functions which constitute  $\mathbf{f}(\mathbf{x})$  are called *features*. A feature is just a (any) function  $f : \mathbb{R}^n \to \mathbb{R}$  which computes a scalar characteristic of input vectors  $\mathbf{x} \in \mathbb{R}^n$ . If one bundles together *m* such features  $f_1, \ldots, f_m$  one obtains a *feature map*  $(f_1, \ldots, f_m)' =: \mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$  which maps input vectors to *feature vectors*.

It is very typical, almost universal, for ML systems to include an initial data processing stage where raw, high-dimensional input patterns are first projected from their original pattern space  $\mathbb{R}^n$  to a lower-dimensional feature space. In TICS, for example, a neural network was trained in a clever way to reduce the 1,440,000-dimensional raw input patterns to a 4,096-dimensional feature vector.

The ultimate quality of the learning system clearly depends on a good choice of features. Unfortunately there does not exist a unique or universal method to identify "good" features. Depending on the learning task and the nature of the data, different kinds of features work best. Accordingly, ML research has come up with a rich repertoire of *feature extraction* methods.

On an intuitive level, a "good" set of features  $\{f_1, \ldots, f_m\}$  should satisfy some natural conditions:

- The number m of features should be small after all, one of the reasons for using features is dimension reduction.
- Each feature  $f_i$  should be *relevant* for the task at hand. For example, when the task is to distinguish helicopter images from winged aircraft photos (a 2-class classification task), the brightness of the background sky would be an irrelevant feature; but the binary feature "has wings" would be extremely relevant.
- There should be *little redundancy* each used feature should contribute some information that the others don't.
- A general intuition about features is that they should be rather cheap and easy to compute at the front end where the ML system meets the raw data. The "has wings" feature for helicopter vs. winged aircraft classification more or less amounts to actually solving the classification task and presumably is neither cheap nor easy to compute. Such highly informative, complex features are sometimes called *high-level features*; they are usually computed on the basis of more elementary, *low-level* features.
- Often features are computed stage-wise, low-level features first (directly from data), then stage by stage more complex, more directly task-solving, more "high-level cognition" features are built by combining the lower-level ones. *Feature hierarchies* are often found in ML systems. Example: in face recognition from photos, low-level features might extract coordinates of isolated black dots from the photo (candidates for the pupils of the person's eyes); intermediate features might give distance ratios between eyes, nose-tip, center-of-mouth; high-level features might indicate gender or age. Such feature hierarchies are implicit in feedforward neural networks. Each neuron in the first hidden layer can be seen as computing a feature from the input patterns; each neuron in the next hidden layer computes next-higher-level features that are computed from the first-level features, etc.

To sharpen our intuitions about features, let us hand-design some features for use in a simple image classification task. Consider a training dataset which consists of grayscale, low-resolution pictures of handwritten digits (Figure 15). Here and further in this section I will illustrate dimension reduction techniques with this simple "Digits" benchmark dataset which was first described in Duin and Tax 2000. It is also available at the UCI machine learning repository at https://archive.ics.uci.edu/dataset/72/multiple+features. This dataset contains 2000 grayscale images of handwritten digits, 200 from each class. The images are  $15 \times 16$  sized, making for n = 240 dimensional image vectors **x**. We assume they have been normalized to a real-valued pixel value range in [0, 1] with 0 = white and 1 = black. The Digits benchmark task is to train a classifier which classifies such images into the ten digit classes.



Figure 15: Some examples from the Digits dataset.

Here are some candidates for features that might be useful for this task.

- Mean brightness.  $f_1(\mathbf{x}) = \mathbf{1}'_n \mathbf{x} / n$  ( $\mathbf{1}_n$  is the vector of n ones). This is just the mean brightness of all pixels. Might be useful e.g. for distinguishing "1" images from "8" images because we might suspect that for drawing an "8" one needs more black ink than for drawing a "1". Cheap to compute but not very informative by itself for the task of digit classification.
- **Radiality.** An image  $\mathbf{x}$  is assigned a value  $f_2(\mathbf{x}) = 1$  if and only if two conditions are met: (i) the center horizontal pixel line crossing the image from left to right has a sequence of pixels that changes from black to white to black; (ii) same for the center vertical pixel line. If this double condition is not met, the image is assigned a feature value of  $f_2(\mathbf{x}) = 0$ .  $f_2$  thus has only two possible values; it is called a *binary feature*. We might suspect that only the "0" images have this property. This would be a slightly less cheap-to-compute feature compared to  $f_1$ , but likely more informative about classes.
- **Prototype matching.** For each of the 10 classes  $c_j$  (j = 1, ..., 10) define a prototype vector  $\pi_j$  as the mean image vector of all examples of that class contained in the training dataset:  $\pi_j = 1/N_j \sum_{\mathbf{x} \text{ is training image of class } j \mathbf{x}$ . Then define 10 features  $f_3^j$  by match with these prototype vectors:  $f_3^j(\mathbf{x}) = \pi'_j \mathbf{x}$ . We might hope that  $f_3^j$  has a high value for patterns of class j and low values for other patterns.

Hand-designing features can be quite effective. Human insight on the side of the data engineer is a success factor for ML systems that can hardly be overrated. In fact, the classical ML approach to speech recognition was for two decades relying on low-level acoustic features that had been hand-designed by insightful phonologists. The MP3 sound coding format is based on features that reflect characteristics of the human auditory system. Many of the first functional computer vision and optical character recognition systems relied heavily on visual feature hierarchies which grew from the joint efforts of signal processing engineers and cognitive neuroscience experts.

However, since hand-designing good features means good insight on the side of the engineer, and good engineers are rare and have little time, the practice of ML today relies much more on features that are obtained from learning algorithms. Numerous methods exist. In the following subsections we will inspect three such methods.

### 4.2 K-means clustering

K-means clustering is an algorithm which allows one to split a collection of training data points  $(\mathbf{x}_i)_{i=1,\dots,N} \in \mathbb{R}^n$  into K clusters  $C_1, \dots, C_K$ , such that points from the same cluster lie close to each other while being further away from points in other clusters. Each cluster  $C_j$  is represented by its codebook vector  $c_j$ , which is the vector pointing to the mean of all vectors in the cluster — that is, to the cluster's center of gravity.

The codebook vectors can be used in various ways to compress *n*-dimensional test data points  $\mathbf{x}^{\text{test}}$  into lower-dimensional formats. The classical method, which also explains the naming of the  $c_j$  as "codebook" vectors, is to represent  $\mathbf{x}^{\text{test}}$  simply by the index j of the codebook vector  $c_j$  which lies closest to  $\mathbf{x}^{\text{test}}$ , that is, which has the minimal distance  $\alpha_j = \|\mathbf{x}^{\text{test}} - c_j\|$ . The high-dimensional point  $\mathbf{x}^{\text{test}}$  becomes represented by a single number, thus here we would have m = 1. This method is clearly very economical. But it is also clear that relevant information contained in vectors  $\mathbf{x}^{\text{test}}$  may be lost — maybe too much information for many ML applications. In digital communication technologies, however, this simple codebook coding is widely used.

Another, less lossy, way to make use of the codebook vectors for dimension reduction is to compute the distances  $\alpha_j$  between  $\mathbf{x}^{\text{test}}$  and the codebook vectors, and reduce  $\mathbf{x}^{\text{test}}$  to the K-dimensional distance vector  $(\alpha_1, \ldots, \alpha_K)'$ . When  $K \ll n$ , the dimension reduction is substantial. The vector  $(\alpha_1, \ldots, \alpha_K)'$  can be considered a feature vector.

We can be brief when explaining the K-means clustering algorithm, because it is almost self-explaining. The rationale for defining clusters is that points within a cluster should have small metric distance to each other, points in different clusters should have large distance from each other. The procedure runs like this:



Figure 16: Clusters obtained from K-means clustering (schematic): For a training set of data points (light blue dots), a spatial grouping into clusters  $C_j$  is determined by the K-means algorithm. Each cluster becomes represented by a codebook vector (dark blue crosses). The figure shows three clusters. The light blue straight lines mark the cluster boundaries. A test data point  $\mathbf{x}^{\text{test}}$  (red cross) may then become coded in terms of the distances  $\alpha_j$  of that point to the codebook vectors. Since this  $\mathbf{x}^{\text{test}}$  falls into the second cluster  $C_2$ , it could also be compressed into the codebook index "2" of this cluster.

**Given:** a training data set  $(\mathbf{x}_i)_{i=1,\dots,N} \in \mathbb{R}^n$ , and a number K of clusters that one maximally wishes to obtain.

**Initialization:** randomly assign the training points to K sets  $S_j$  (j = 1, ..., K). **Repeat:** For each set  $S_j$ , compute the mean  $\mu_j = |S_j|^{-1} \sum_{\mathbf{x} \in S_j} \mathbf{x}$ . This mean vector  $\mu_j$  is the "center of gravity" of the vector cluster  $S_j$ . Create new sets  $S'_j$  by putting each data point  $\mathbf{x}_i$  into that set  $S'_j$  where  $\|\mathbf{x}_i - \mu_j\|$  is minimal. If some  $S'_j$  remains empty, dismiss it and reduce K to K' by subtractring the number of dismissed empty sets (this happens rarely). Put  $S_j = S'_j$  (for the nonempty sets) and K = K'.

**Termination:** Stop when in one iteration the sets remain unchanged.

It can be shown that at each iteration, the error quantity

$$J = \sum_{j=1}^{K} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mu_j\|^2$$
(26)

will not increase. The algorithm typically converges quickly and works well in practice. It finds a local minimum or saddle point of J. The final clusters  $S_j$  may depend on the random initialization. The clusters are bounded by planar boundary facets; each cluster forms a *Voronoi cell*. K-means cannot find clusters defined by curved boundaries. Figure 17 shows an example of a clustering run using K-means.



Figure 17: Running K-means with K = 3 on two-dimensional training points. Thick dots mark cluster means  $\mu_j$ , lines mark cluster boundaries. The algorithm terminates after three iterations, whose boundaries are shown in light gray, dark gray, red. (Picture taken from Chapter 10 of the textbook Duda, P. E. Hart, and Stork 2001).

K-means clustering and other clustering methods have many uses besides dimension reduction. Clustering can also be seen as a stand-alone technique of unsupervised learning. The detected clusters and their corresponding codebook vectors are of interest in their own right. They reveal a basic structuring of a set of patterns  $\{\mathbf{x}_i\}$  into subsets of mutually similar patterns. These clusters may be further analyzed individually, given meaningful names and helping a human data analyst to make useful sense of the original unstructured data cloud. For instance, when the patterns  $\{\mathbf{x}_i\}$  are customer profiles, finding a good grouping into subgroups may help to design targetted marketing strategies.

#### 4.3 Principal component analysis

Like clustering, principal component analysis (PCA) is a basic data analysis technique that has many uses besides dimension reduction. But here we will focus on that use.

Generally speaking, a good dimension reduction method, that is, a good feature map  $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ , should preserve much information contained in the highdimensional patterns  $\mathbf{x} \in \mathbb{R}^n$  and encode it robustly in the feature vectors  $y = \mathbf{f}(\mathbf{x})$ . But, what does it mean to "preserve information"? A clever answer is this: a feature representation  $\mathbf{f}(\mathbf{x})$  preserves information about  $\mathbf{x}$  to the extent that  $\mathbf{x}$ can be *reconstructed* from  $\mathbf{f}(\mathbf{x})$ . That is, we wish to have a *decoding function*  $\mathbf{d} : \mathbb{R}^m \to \mathbb{R}^n$  which leads back from the feature vector encoding  $\mathbf{f}(\mathbf{x})$  to  $\mathbf{x}$ , that is we wish to achieve

$$\mathbf{x} \approx \mathbf{d} \circ \mathbf{f}(\mathbf{x})$$

And here comes a fact of empowerment: when  $\mathbf{f}$  and  $\mathbf{d}$  are confined to *linear* functions and when the similarity  $\mathbf{x} \approx \mathbf{d} \circ \mathbf{f}(\mathbf{x})$  is measured by mean square error, the optimal solution for  $\mathbf{f}$  and  $\mathbf{d}$  can be easily and cheaply computed by a method that is known since the early days of statistics, *principal component analysis* (PCA). It was first found, in 1901, by Karl Pearson, one of the fathers of modern mathematical statistics. The same idea has been independently rediscovered under many other names in other fields and for a variety of purposes (check out https://en.wikipedia.org/wiki/Principal\_component\_analysis for the history). Because of its simplicity, analytical transparency, modest computational cost, and numerical robustness PCA is widely used — it is the first-choice default method for dimension reduction that is tried almost by reflex, before more elaborate methods are maybe considered.

PCA is best explained alongside with a visualization (Figure 18). Assume the patterns are 3-dimensional vectors, and assume we are given a sample of N = 200 raw patterns  $\mathbf{x}_1, \ldots, \mathbf{x}_{200}$ . We will go through the steps of computing a PCA for this demo dataset.

The first step in PCA is to *center* the training patterns  $\mathbf{x}_i$ , that is, subtract their mean  $\mu = 1/N \sum_i \mathbf{x}_i$  from each pattern, obtaining centered patterns  $\bar{\mathbf{x}}_i = \mathbf{x}_i - \mu$ . The centered patterns form a point cloud in  $\mathbb{R}^n$  whose center of gravity is the origin (see Figure 18A).

This point cloud will usually not be perfectly spherically shaped, but instead extend in some directions more than in others. "Directions" in  $\mathbb{R}^n$  are characterized by unit-norm "direction" vectors  $u \in \mathbb{R}^n$ . The distance of a point  $\bar{\mathbf{x}}_i$  from the origin *in the direction of* u is given by the projection of  $\bar{\mathbf{x}}_i$  on u, that is, the inner product  $u' \bar{\mathbf{x}}_i$  (see Figure 19).

The "extension" of a centered point cloud  $\{\bar{\mathbf{x}}_i\}$  in a direction u is defined to be the mean squared distance to the origin of the points  $\bar{\mathbf{x}}_i$  in the direction of u. The direction of the largest extension of the point cloud is hence the direction vector given by

$$u_1 = \underset{u, \|u\|=1}{\operatorname{argmax}} 1/N \sum_i (u' \,\bar{\mathbf{x}}_i)^2.$$
(27)

Notice that since the cloud  $\bar{\mathbf{x}}_i$  is centered, the mean of all  $u' \bar{\mathbf{x}}_i$  is zero, and hence the number  $1/N \sum_i (u' \bar{\mathbf{x}}_i)^2$  is the variance of the numbers  $u' \bar{\mathbf{x}}_i$ .

Inspecting Figure 18**A**, one sees how  $u_1$  points in the "longest" direction of the pattern cloud. The vector  $u_1$  is called the first *principal component* (PC) of the centered point cloud.

Next step: project patterns on the (n-1)-dimensional linear subspace of  $\mathbb{R}^n$ that is orthogonal to  $u_1$  (Figure 18B). That is, map pattern points  $\bar{\mathbf{x}}$  to  $\bar{\mathbf{x}}^* = \bar{\mathbf{x}} - (u'_1 \bar{\mathbf{x}}) \cdot u_1$ . Within this "flattened" pattern cloud, again find the direction



Figure 18: Visualization of PCA. A. Centered data points and the first principal component vector  $u_1$  (blue). The origin of  $\mathbb{R}^3$  is marked by a red cross. B. Projecting all points to the orthogonal subspace of  $u_1$  and computing the second PC  $u_2$  (green). C. Situation after all three PCs have been determined. D. Summary visualization: the original data cloud with the three PCs and an ellipsoid aligned with the PCs whose main axes are scaled to the standard deviations of the data points in the respective axis direction. E. A new dimension-reduced coordinate system obtained by the projection of data on the subspace  $U_m$  spanned by the m first PCs (here: the first two).

vector of greatest variance

$$u_2 = \underset{u, \|u\|=1}{\operatorname{argmax}} 1/N \sum_i (u' \, \bar{\mathbf{x}}_i^*)^2$$

and call it the second PC of the centered pattern sample. From this procedure it is clear that  $u_1$  and  $u_2$  are orthogonal, because  $u_2$  lies in the orthogonal subspace of  $u_1$ .

Now repeat this procedure: In iteration k, the k-th PC  $u_k$  is constructed by projecting pattern points to the linear subspace that is orthogonal to the already computed PCs  $u_1, \ldots, u_{k-1}$ , and  $u_k$  is obtained as the unit-length vector pointing in the "longest" direction of the current (n - k + 1)-dimensional pattern point distribution. This can be repeated until n PCs  $u_1, \ldots, u_n$  have been determined. They form an orthonormal coordinate system of  $\mathbb{R}^n$ . Figure 18**C** shows this situation, and Figure 18**D** visualizes the PCs plotted into the original data cloud.


Figure 19: Projecting a point  $\bar{\mathbf{x}}_i$  on a direction vector u: the inner product  $u' \bar{\mathbf{x}}_i$  (length of the green vector) is the distance of  $\bar{\mathbf{x}}_i$  from the origin along the direction given by u.

Now define features  $f_k$  (where  $1 \le k \le n$ ) by

$$f_k : \mathbb{R}^n \to \mathbb{R}, \quad \mathbf{x} \mapsto u'_k \, \bar{\mathbf{x}},$$
 (28)

that is,  $f_k(\bar{\mathbf{x}})$  is the projection component of  $\bar{\mathbf{x}}$  on  $u_k$ . Since the *n* PCs form an orthonormal coordinate system, any point  $\mathbf{x} \in \mathbb{R}^n$  can be perfectly reconstructed from its feature values by

$$\mathbf{x} = \mu + \sum_{k=1,\dots,n} f_k(\mathbf{x}) \, u_k. \tag{29}$$

The PCs and the corresponding features  $f_k$  can be used for dimension reduction as follows. We select the first few ("leading") PCs  $u_1, \ldots, u_m$  up to some index m. Then we obtain a feature map

$$\mathbf{f}: \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{x} \mapsto (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))'.$$
 (30)

At the beginning of this section I spoke of a decoding function  $\mathbf{d} : \mathbb{R}^m \to \mathbb{R}^n$ which should recover the original patterns  $\mathbf{x}$  from their feature vectors  $\mathbf{f}(\mathbf{x})$ . In our PCA story, this decoding function is given by

$$\mathbf{d}: (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))' \mapsto \mu + \sum_{k=1}^m f_k(\mathbf{x}) \, u_k.$$
(31)

How "good" is this dimension reduction, that is, how similar are the original patterns  $\mathbf{x}_i$  to their reconstructions  $\mathbf{d} \circ \mathbf{f}(\mathbf{x}_i)$ ? When dissimilarity of two patterns  $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$  is measured in the square error sense by

$$\delta(\mathbf{x}_1, \mathbf{x}_2) := \|\mathbf{x}_1 - \mathbf{x}_2\|^2,$$

a full answer can be given. Let

$$\sigma_k^2 = 1/N \sum_i f_k(\mathbf{x}_i)^2$$

denote the variance of the feature values  $f_k(\mathbf{x}_i)$  (notice that the mean of the  $f_k(\mathbf{x}_i)$ , taken over all patterns, is zero, so  $\sigma_k^2$  is indeed their variance). Then the mean square distance between patterns and their reconstructions is

$$1/N \sum_{i} \|\mathbf{x}_{i} - \mathbf{d} \circ \mathbf{f}(\mathbf{x}_{i})\|^{2} = \sum_{k=m+1}^{n} \sigma_{k}^{2}.$$
 (32)

A derivation of this result is given in Appendix E.

Equation (32) gives an absolute value for dissimilarity. For applications, however, the relative amount of dissimilarity compared to the mean variance of patterns is more instructive. It is given by

$$\frac{1/N \sum_{i} \|\mathbf{x}_{i} - \mathbf{d} \circ \mathbf{f}(\mathbf{x}_{i})\|^{2}}{1/N \sum_{i} \|\bar{\mathbf{x}}_{i}\|^{2}} = \frac{\sum_{k=m+1}^{n} \sigma_{k}^{2}}{\sum_{k=1}^{n} \sigma_{k}^{2}}.$$
(33)

Real-world sets of patterns often exhibit a rapid (roughly exponential) decay of the feature variances as their index k grows. The ratio (33) then is very small compared to the mean size  $E[\|\bar{X}\|^2]$  of patterns, that is, only very little information is lost by reducing the dimension from n to m via PCA. Our visualization from Figure 18 is not doing justice to the amount of compression savings that is often possible. Typical real-world, high-dimensional, centered data clouds in  $\mathbb{R}^n$  are often very "flat" in the vast majority of directions in  $\mathbb{R}^n$  and these directions can all be zeroed without much damage by the PCA projection.

*Note.* Using the word "variance" above is not mathematically correct. Variance is the *expected* squared deviation from a *population* mean, a statistical property which is different from the *averaged* squared deviation from a *sample* mean. It would have been more proper to speak of an "empirical variance" above, or an "estimated variance". Check out Appendix D!

## 4.4 Mathematical properties of PCA and an algorithm to compute PCs

The key to analysing and computing PCA is the  $(n \times n)$ -dimensional covariance matrix  $C = 1/N \sum_i \bar{\mathbf{x}}_i \bar{\mathbf{x}}'_i$ , which can be easily obtained from the centered training data matrix  $\bar{X} = [\bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_N]$  by  $C = 1/N \bar{X} \bar{X}'$ . C is very directly related to PCA by the following facts:

- 1. The PCs  $u_1, \ldots, u_n$  form a set of orthonormal, real eigenvectors of C.
- 2. The feature variances  $\sigma_1^2, \ldots, \sigma_n^2$  are the eigenvalues of these eigenvectors.

A derivation of these facts can be found in my legacy ML lecture notes, Section 4.4.1 (https://www.ai.rug.nl/minds/uploads/LN\_ML\_Fall11.pdf). Thus, the principal component vectors  $u_k$  and their associated data variances  $\sigma_k^2$  can be directly gleaned from C.

Computing a set of unit-norm eigenvectors and eigenvalues from C can be most conveniently done by computing the *singular value decomposition* (SVD) of C. Algorithms for computing SVDs of arbitrary matrices are shipped with all numerical or statistical mathematics software packages, like Matlab, R, or Python with numpy. At this point suffice it to say that every covariance matrix C is a so-called *positive semi-definite* matrix. These matrices have many nice properties. Specifically, their eigenvectors are orthogonal and real, and their eigenvalues are real and nonnegative.

In general, when an SVD algorithm is run on an n-dimensional positive semidefinite matrix C, it returns a factorization

$$C = U \Sigma U',$$

where U is an  $n \times n$  matrix whose columns are the normed orthogonal eigenvectors  $u_1, \ldots, u_n$  of C and where  $\Sigma$  is an  $n \times n$  diagonal matrix which has the eigenvalues  $\lambda_1, \ldots, \lambda_n$  on its diagonal. They are usually arranged in descending order. Thus, computing the SVD of  $C = U \Sigma U'$  directly gives us the desired PC vectors  $u_k$ , lined up in the columns of U, and the variances  $\sigma_k^2$ , which appear as the eigenvalues of C, collected in  $\Sigma$ .

This enables a convenient control of the goodness of similarity that one wants to ensure. For example, if one wishes to preserve 98% of the variance information from the original patterns, one can use the r.h.s. of (33) to determine the "cutoff" m such that the ratio in this equation is about 0.02.

## 4.5 Summary of PCA based dimension reduction procedure

**Data.** A set  $(\mathbf{x}_i)_{i=1,\dots,N}$  of *n*-dimensional pattern vectors.

**Result.** An *n* dimensional mean pattern vector  $\mu$  and *m* principal component vectors arranged column-wise in an  $n \times m$  sized matrix  $U_m$ .

Procedure.

Step 1. Compute the pattern mean  $\mu$  and center the patterns to obtain a centered pattern matrix  $\bar{X} = [\bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_N].$ 

**Step 2.** Compute the SVD  $U \Sigma U'$  of  $C = 1/N \bar{X}\bar{X}'$  and keep from U only the first m columns, making for a  $n \times m$  sized matrix  $U_m$ .

Usage for compression. In order to compress a new *n*-dimensional pattern  $\mathbf{x}$  to a *m*-dimensional feature vector  $\mathbf{f}(\mathbf{x})$ , compute  $\mathbf{f}(\mathbf{x}) = U'_m \bar{\mathbf{x}}$ .

Usage for uncompression (decoding). In order to approximately restore  $\mathbf{x}$  from its feature vector  $\mathbf{f}(\mathbf{x})$ , compute  $\mathbf{x}_{restored} = \mu + U_m \mathbf{f}(\mathbf{x})$ .

#### 4.6 Eigendigits

For a demonstration of dimension reduction by PCA, consider the "3" digit images. After reshaping the images into 240-dimensional grayscale vectors and centering and computing the PCA on the basis of N = 100 training examples, we obtain 240 PCs  $u_k$  associated with variances  $\sigma_k^2$ . Only the first 99 of these variances are nonzero (because the 100 image vectors  $\mathbf{x}_i$  span a 100-dimensional subspace in  $\mathbb{R}^{240}$ ; after centering the  $\bar{\mathbf{x}}_i$  however span only a 99-dimensional subspace – why? homework exercise! – thus the matrix  $C = 1/N \bar{X} \bar{X}'$  has rank at most the rank of  $\bar{X}$ , which is 99), thus only the first 99 PCs are useable. Figure 20 A shows some of these eigenvectors  $u_i$  rendered as  $15 \times 16$  grayscale images. It is customary to call such PC re-visualizations eigenimages, in our case "eigendigits". (If you have some spare time, do a Google image search for "eigenfaces" and you will find weird-looking visualizations of PC vectors obtained from PCA carried out on face pictures.)

Figure 20 **B** shows the log variances  $\log_{10} \sigma_i^2$  of the first 99 PCs. You can see the rapid (roughly exponential) decay. Aiming for a dissimilarity ratio (Equation 33) of 0.1 gives a value of m = 32. Figure 20 **C** shows the reconstructions of some "3" patterns from the first m PC features using (31).



Figure 20: A. Visualization of a PCA computed from the "3" training images. Top left panel shows the mean  $\mu$ , the next 7 panels (row-wise) show the first 7 PCs. Third row shows PCs 20–23, last row PCs 96-99. Grayscale values have been automatically scaled per panel such that they spread from pure white to pure black; they do not indicate absolute values of the components of PC vectors. B. The (log10 of) variances of the PC features on the "3" training examples. C. Reconstructions of digit "3" images from the first m = 32 features, corresponding to a re-constitution of 90% of the original image dataset variance. First row: 4 original images from the training set. Second row: their reconstructions. Third row: 4 original images from the test set. Last row: their reconstruction.

#### 4.7 Self-organizing maps

Nature herself has been facing the problem of dimension reduction when She was using evolution as a learning algorithm to obtain well-performing brains. Specifically, in human visual processing the input dimension n is in the order of a million (judging from the number of axons in the optical nerve), but the "space" where the incoming n-dimensional raw visual signal is processed has only m = 2 dimensions: it is the walnut-kernel-curly surface of the brain, the *cortical sheet*. Now this is certainly an over-simplified view of biological neural information processing, but at least in some brain areas and for some kinds of input projecting into those areas, it seems fairly established in neuroscience that an incoming high-dimensional signal becomes mapped on an m = 2 dimensional representation on the cortical sheet.

In the 1980 decade, Teuvo Kohonen developed an artificial neural network model which (i) could explain how this neural dimension reduction could be learnt in biological brains; and which (ii) could be used as a practical machine learning algorithm for dimension reduction. His model is today known as *Kohonen Network* or *Self-Organizing Map (SOM)*. The SOM model is one the very few and precious instances of a neural information processing model which is anchored in, and can bridge between, both neuroscience and machine learning. It made a strong impact in both communities. In the machine learning / data analysis world SOMs have been superseded in later years by other methods, and in my perception they are only rarely used today. But this may again change when non-digital neuromorphic microchip technologies will mature (my own research field!). In neuroscience research, SOM-inspired models continue to be explored. Furthermore, the SOM model is simple and intuitive. I think all of this is reason enough to include it in this lecture.

The learning task for SOMs is defined as follows (for the case of m = 2):

- **Given:** a pattern collection  $\mathcal{P} = (\mathbf{x}_i)_{i=1,\dots,N}$  of points in  $\mathbb{R}^n$ .
- Also given: a 2-dimensional grid of "neurons"  $\mathcal{V} = (v_{kl})_{1 \leq k \leq K, 1 \leq l \leq L}$ , where k, l are the grid coordinates of neuron  $v_{kl}$  (you might think of this as a rectangular segment in the cortical surface sheet).

**Learning objective:** find a map  $\kappa : \mathcal{P} \to \mathcal{V}$  which

- 1. preserves input space topology (the metric topology of  $\mathbb{R}^n$ ), that is, points  $\mathbf{x}_j$  in the neighborhood of a point  $\mathbf{x}_i$  should be mapped to  $\kappa(\mathbf{x}_i)$  or to grid neighbors of  $\kappa(\mathbf{x}_i)$ , and which
- 2. distributes the patterns from the collection  $\mathcal{P}$  evenly over the grid cells, that is, all grid neurons  $v_{kl}$  should have approximately the same number of  $\kappa$ -preimages.

Four comments: (i) In classical SOM models, input patterns  $\mathbf{x}$  are always assumed to have unit norm. (ii) Here I use a grid with a rectangular neighborhood

structure. Classical SOM papers and many applications of SOMs use a hexagonal neighborhood structure instead, where each neuron has 6 neighbors, all at the same distance. (iii) The "learning objective" sounds vague. It is. At the time when Kohonen introduced SOMs, tailoring learning algorithms along well-defined loss functions was not standard. Kohonen's modeling attitude was oriented toward biology and a mathematical analysis of the SOM algorithm was not part of his agenda. In fact, the problem to find a loss function which is minimized by the original SOM algorithm was still unsolved in the year 2008 (Yin 2008). I don't know what the current state of research in this respect is — I would guess not much has happened since. (iv) The learning task is impossible to solve. One cannot map the *n*-dimensional pattern space  $\mathbb{R}^n$  to the lower-dimensional (even just 2-dimensional) space  $\mathbb{R}^m$  while preserving neighborhood relations. For a graphical illustration of this impossibility, consider the case where the patterns are 3-dimensional and are uniformly spread in the unit cube  $[0,1] \times [0,1] \times [0,1]$ . Then, in order to let every grid neuron have about the same number of pattern points which are mapped on it (condition 2), the SOM learning task would require that the 2-dimensional neuron grid — think of it as a large square sheet of paper with the gridlines printed on — becomes "wrinkled up" in the 3-dimensional cube such that in every place in the cube the surrounding "neural sheet density" is about the same (Figure 21). When this condition 2 is met (as in the figure), condition 1 is necessarily violated: there will be points in the high-dimensional space which are close to each other but which will become mapped to grid neurons that are far from each other on the grid. Thus, SOM training always means finding a compromise.

In a trained SOM, each grid neuron  $v_{kl}$  is associated with a weight vector  $\mathbf{w}(v_{kl}) \in \mathbb{R}^n$ . This weight vector represents a point in the space  $\mathbb{R}^n$  from which the training patterns  $\mathbf{x}_i$  came. One could say that  $\mathbf{w}(v_{kl}) \in \mathbb{R}^n$  "locates" the grid neuron  $v_{kl}$  in the input pattern space (in Figure 21, think of every gridline crossing point as a location of a grid neuron  $v_{kl}$ ; the 3-dimensional position of this crossing point in the cube volume gives the weight vector  $\mathbf{w}(v_{kl})$ ).

The weight vectors associated with the grid neurons are the basis for defining the function  $\kappa : \mathcal{P} \to \mathcal{V}$  which assigns a grid neuron to every pattern  $\mathbf{x} \in \mathcal{P}$ , and also to any new test pattern. If some pattern  $\mathbf{x} \in \mathbb{R}^n$  is presented to the trained SOM, its  $\kappa$ -image is computed by

$$\kappa(\mathbf{x}) = \operatorname*{argmax}_{v_{kl} \in \mathcal{V}} \mathbf{w}'(v_{kl}) \mathbf{x}.$$
(34)

In words, the neuron v whose weight vector  $\mathbf{w}(v)$  best matches the input pattern  $\mathbf{x}$  is chosen. "Best matching" here means: maximal inner product  $\mathbf{w}' \mathbf{x}$ . In other variants of an SOM, one uses minimal metric distance as "best matching" criteriom, i.e. instead of  $\mathbf{w}'(v_{kl})\mathbf{x}$  one would write  $\|\mathbf{w}'(v_{kl}) - \mathbf{x}\|$  in Equation (34). Kohonen used the "best matching" criterion (34) in his classical papers. In the SOM literature the neuron determined by (34) is called the *best matching unit* (BMU) for pattern  $\mathbf{x}$ . Clearly the map  $\kappa$  is determined by the weight vectors



Figure 21: Trying to uniformly fill a cube volume with a 2-dimensional grid sheet will invariably lead to some points in the cube which are close to two (or more) "folds" of the sheet. That is, points that are far away from each other in the 2-dimensional sheet (here for instance: points on the red gridline vs. points on the blue gridline segment) will be close to each other in the 3-dimensional cube space; or stated the other way round: some points that are close in the 3D cube will become separated to distant spots on the neuron grid. "Crumpled grid" graphic taken from Obermayer, Ritter, and Schulten 1990.

 $\mathbf{w}(v)$ . In order to train them on the basis of the training data set  $\mathcal{P}$ , the basic SOM learning algorithm works as follows:

**Initialization:** The weights  $\mathbf{w}(v_{kl})$  are set to small random values.

Iterate until convergence: 1. Randomly select a training pattern  $\mathbf{x} \in \mathcal{P}$ .

- 2. Determine the BMU  $v_{\mathsf{BMU}}$  for this pattern, based on the current weight vectors  $\mathbf{w}(v)$ .
- 3. Update all the grid's weight vectors  $\mathbf{w}(v_{kl})$  according to the formula

$$\mathbf{w}(v_{kl}) \leftarrow \mathbf{w}(v_{kl}) + \lambda \ f_r(d(v_{kl}, v_{\mathsf{BMU}})) \ (\mathbf{x} - \mathbf{w}(v_{kl})). \tag{35}$$

4. Make r a little smaller before entering the next iteration.

In this sketch of an algorithm,  $\lambda$  is a *learning rate* — a small number, for instance  $\lambda = 0.01$ , plugged in for numerical stability. The function d is the Euclidean distance (between two neurons positions on the grid).  $f_r : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ is a non-negative, monotonously decreasing function defined on the nonnegative reals which satisfies  $f_r(0) = 1$  and which goes to zero as its argument grows. The function is parametrized by a radius parameter r > 0, where greater values of r



Figure 22: Three Gaussian-like distance-depending weighting functions  $f_r(x) = \exp(-x^2/r^2)$  with values r = 0.5, 1, 2 (red, green, blue).

spread out the range of  $f_r$ . A common choice is to use a Gaussian-like function  $f_r(x) = \exp(-x^2/r^2)$  (see Figure 22).

Here is an intuitive explanation of this learning algorithm. When a new training pattern  $\mathbf{x}$  has been presented and the BMU has been determined, each weight vector  $\mathbf{w}(v_{kl})$  is adapted a little bit. The weight update formula (35) adds to  $\mathbf{w}(v_{kl})$  a small multiple of  $\mathbf{x} - \mathbf{w}(v_{kl})$ . This pulls  $\mathbf{w}(v_{kl})$  into the direction of  $\mathbf{x}$ . How strong this pull is depends on how close  $v_{kl}$  is to  $v_{\mathsf{BMU}}$  — the closer, the stronger. This is regulated by the distance-dependent factor  $f_r(d(v_{kl}, v_{\mathsf{BMU}}))$ . Obviously the adaptation is strongest for the BMU. When (in the early phases of the algorithm) r is large, neurons in the wider vicinity of the BMU will likewise receive rather strong weight adjustments, while only far-away neurons remain more or less unaffected.

This mechanics has the effect that after convergence, the training dataset  $\mathcal{P}$ will be covered by grid neurons rather evenly (see objective nr. 2 stated above). To get an intuition why this is so, let us consider a specific scenario. Assume that the pattern set  $\mathcal{P}$  contains a dense cluster of mutually quite similar patterns  $\mathbf{x}$ , besides a number of other, dissimilar patterns. Furthermore assume that we are in an early stage of the learning process, where the radius r is still rather large, and also assume that at this early stadium of learning, each pattern from the cluster yields the same BMU  $v_0$ . Due to the large number of members in the cluster, patterns from that cluster will be drawn for the learning algorithm rather often. With r large, this will have the effect that neurons in the wider neighborhood of the BMU  $v_0$  will grow their weight vectors toward  $\mathbf{w}(v_0)$ . After some time,  $v_0$ will be surrounded by grid neurons whose weight vectors are all similar to  $\mathbf{w}(v_0)$ , and  $\mathbf{w}(v_0)$  will roughly be the mean of all patterns  $\mathbf{x}$  in the cluster. Now, some patterns  $\mathbf{x}'$  in the cluster will start to have as their BMU not  $v_0$  any longer, but some of its surrounding neighbors (why?). As a consequence, increasingly many patterns in the cluster will best-match the weight vectors of an increasing number of neighbors of  $v_0$ : the subpopulation of grid neurons which respond to cluster patterns has grown from the singleton population  $\{v_0\}$  to a larger one. This growth will continue until the population of grid neurons responding to cluster patterns has become so large that each member's BMU-response-rate has become

too low to further drive this population's expansion.

The radius r is set to large values initially in order to let all (or most) patterns in  $\mathcal{P}$  compete with each other, leading to a coarse global organization of the emerging map  $\kappa$ . In later iterations, increasingly smaller r leads to a fine-balancing of the BMU responses of patterns that are similar to each other.

SOM learning algorithms come in many variations. I sketched an arbitrary exemplar. The core idea is always the same. Setting up a SOM learning algorithm and tuning it is not always easy – the weight initialization, the decrease schedule for r, the learning rate, the random sampling strategy of training patterns from  $\mathcal{P}$ , the grid dimension (2 or 3 or even more... 2 is the most common choice), or the pattern preprocessing (for instance, normalizing all patterns to the same norm) are all design decisions that can have a strong impact on the convergence properties and final result quality of the learning process.

Yin 2008 includes a brief survey of SOM algorithm variants. I mention in passing an algorithm which has its roots in SOMs but is significantly different: The *Neural Gas* algorithm (good brief intro in https://en.wikipedia.org/wiki/Neural\_gas), like SOMs, leads to a collection of neurons which respond to patterns from a training set through trainable weight vectors. The main difference is that the neurons are not spatially arranged on a grid but are spatially uncoupled from each other (hence, neural "gas"). The spatially defined distance d appearing in the adaptation efficacy term  $f_r(d(v_{kl}, v_{BMU}))$  is replaced by a rank ordering: the neuron with the best response to training pattern  $\mathbf{x}$  (i.e., the BMU) is adapted most, the unit v with the second best response (i.e., second largest value of  $\mathbf{w}(v) \mathbf{x}$ ) is adapted second most strongly, etc.

For a quick SOM demo I used a version of a legacy Matlab toolbox published by Kohonen's own research group (I downloaded it 20 years ago). As a pattern dataset  $\mathcal{P}$  I used the Digits dataset that I also used before in this section. I used 100 examples from each digit class. Figure 23 shows the result of training an 8 × 8 neuron grid on this dataset. As expected, the ten digit classes become represented each by approximately the same number of grid neurons, reflecting the fact that the classes were represented in  $\mathcal{P}$  in equal shares.

I also generated an unbalanced pattern set that had 180 examples of class "5" and 20 examples of every other class. The result is shown in Figure 24. As it should be, roughly half of the SOM neurons are covering the "5" class examples, while the other SOM neurons reach out their weight vectors into the remaining classes. This is a desirable effect: SOM training on the basis of unbalanced datasets will lead to a higher resolution for the pattern sorts that occur more often in the dataset (more grid neurons covering the more often found sorts of data points).

Practical uses of SOMs appear to have been mostly 2-dimensional visualizations for exploring (labelled) high-dimensional datasets in two-dimensional graphics. Specifically, such SOM visualizations can give insight into metric similarities between pattern classes. For instance, inspecting the bottom right  $3 \times 3$  panels in Figure 23, one finds "morphable" similarities between certain versions of "5" and



Figure 23: An 8 × 8 SOM representation of the Digits training dataset. The grayscale images show the weight vectors  $\mathbf{w}(v_{kl})$ , reshaped back to the rectangular pixel image format. The numbers in the small green insets give the most frequent class label of all training patterns which had the respective grid neuron as BMU. If this number is missing, the grid neuron never fired as BMU for any training pattern.

"9" patterns. For the fun of it, in Figure 25 you find how a 1-dimensional SOM curls up in a 2-dimensional pattern space.

What I find more relevant and interesting about SOMs is their use in neuroscience. Learning mechanisms similar to the SOM learning algorithm have been (and are being) invoked to explain the 2-dimensional spatial organization of *cortical maps*. They can explain how neurons on a surface patch of the cortical sheet align their specific responsiveness to high-dimensional input (from sensors or other brain areas) with their local 2-dimensional metric neighborhood. Pictures which put synthetic SOM grids side to side with images recorded from small patches of cortical surface have variously been published. Figure 26 gives an example from a study by Swindale and Bauer 1998.

If you are interested in such themes, I can recommend the recent review of Bednar and Wilson 2016 on cortical maps as a starting point.



Figure 24: Similar to previous figure, but with an unbalanced pattern set where the number of "5" examples was the same as the total number of all other classes.

## 4.8 Summary discussion. Model reduction, data compression, dimension reduction

Reducing the dimensionality of patterns which are represented by vectors is important not only in machine learning but everywhere in science and engineering, where you find it explored under different headline names and for different purposes:

- **Dimension reduction** is a term used in machine learning and statistics. The core motivation is first, that statistical and machine learning models typically depend on a estimate of a probability distribution over some "pattern" space (this distribution estimate may remain implicit); second, that the curse of dimensionality makes it intrinsically hard to estimate distributions from training data points in high-dimensional vector spaces, such that third, a key to success is to reduce the dimension of the original "raw" patterns. Typically, the low-dimensional target format is again real-valued vectors ("feature vectors").
- **Data compression** is a term used in signal processing and communication technology. The practical and economical importance of being able to compress and uncompress bulky data is obvious. The compressed data may have a different type than the raw data. For instance, in vector quantization com-



Figure 25: A 1-dim SOM trying to be a true representation of a 2-dim image. Figure taken from https://eintrittverboten.wordpress.com/2012/10/04/kohonen-art/.

pression, a real-valued vector will become encoded in a natural number, the index of its associated codebook vector. Again there is a close connection with probability, established through Shannon information theory.

Model reduction is a term used when it comes to trim down not just the dimension of static data points but the dimension of entire dynamical system models. All branches of science and engineering today deal with models of complex physical dynamical systems which are instantiated as systems of coupled differential equations — often millions, sometimes billions ... or more ... of them. One obtains such gargantuan systems of coupled ODEs almost immediately when one discretizes system models expressed in partial differential equations (PDEs) or so-called finite-element models (used by engineers to model complex mechanical systems, like skyscrapers or airplane mainframes). Such systems cannot be numerically solved on today's computing hardware and need to be dramatically shrunk before a simulation can be attempted. That is, one must replace a system model made from (say) millions of ODE's by another sytem model made only of a few hundred ODE's. The reduced-size system, when simulated on a computer, should still give very much the same dynamical responses to external stimulation as the full-scale system. I once had an office across the corridor from the office of Anastasios Antoulas. He is a model reduction expert whose expertise was sought by actual skyscraper building companies who needed mathematical simulations of their planned supertowers, in particular to assess how they would respond to earthquake and storm (obviously, they should be demon-



Figure 26: The SOM algorithm reproducing biological cortical response patterns. The scenario: an anaesthesized but eye-open ferret is shown moving images of a bar with varying orientation and direction of movement, while response activity from neurons on a patch (about 10 square mm) of visual cortex is recorded. **A.** A color-coded recording of neural response activity depending on the *orientation* of the visually presented bar. For instance, if the bar was shown in horizontal orientation, the neurons who responded to this orientation with maximal activity are rendered in red. **B.** Like in panel A., but for the motion *direction* of the bar (same cortical patch). **C.**, **D.** Artificial similies of A., B. generated with a SOM algorithm. Figures taken from Swindale and Bauer 1998, who in turn took the panels A. and B from Weliky, Bosking, and Fitzpatrick 1996.

strated not to break, and that in turn boils down to show that they will not start swinging / oscillating / vibrating when externally "perturbed"). I am not familiar with the mathematics of this field, but what I understood from Anastasios' explanations is that these model reductions ultimately boil down to a linear algebra exercise whose core is again the hero of practical linear algebra, the singular value decomposition (SVD). I mention this field for completeness and because the name "model reduction" invites analogies with dimension reduction, although the latter is for static data points and model reduction is for entire dynamical models. But there is a common mathematical core. Antoulas and Sorensen 2001 give a tutorial overview with instructive examples.

In this section we took a look at three methods for dimension reduction of highdimensional "raw" data vectors, namely K-means clustering, PCA, and SOMs. While at first sight these methods appear quite different from each other, there is a unifying view which connects them. In all three of them, the reduction was done by introducing a comparatively simple kind of geometric object in the original high-dimensional pattern space  $\mathbb{R}^n$ , which was then used to re-express raw data points in a lightweight encoding:

- 1. In K-means-clustering, this object is the set of codebook vectors, which can be used to compress a test data point to the mere natural number index of its associated codebook vector; or which can be used to give a reduced K-dimensional vector comprised of the distances  $\alpha_i$  to the codebook vectors.
- 2. In PCA, this object is the *m*-dimensional (affine) linear subspace spanned by the first m eigenvectors of the data covariance matrix. An *n*-dimensional test point is represented by its m coordinates in this subspace.
- 3. In SOMs, this object is the "crumpled-up" grid of SOM neurons v, including their associated weight vectors  $\mathbf{w}(v)$ . A new test data point can be compressed to the natural number index of its BMU. This is entirely analog to how the codebook vectors in K-means clustering can be used. Furthermore, if an *m*-dimensional neuron grid is used (we considered only m = 2above), the grid plane is nonlinearly "folded" into the original dataspace  $\mathbb{R}^n$ , in that every grid point v becomes projected to  $\mathbf{w}(v)$ . It thus becomes an *m*dimensional manifold embedded in  $\mathbb{R}^n$ , which is characterized by codebook vectors. This can be seen as a nonlinear generalization of the *m*-dimensional linear affine hyperplanes embedded in  $\mathbb{R}^n$  in PCA.

Thus, the SOM shares properties with both K-means clustering and PCA. In fact, one can systematically explore a whole spectrum of dimension reduction / data compression algorithms which are located between K-means clustering and PCA, in the sense that they describe m-dimensional manifolds of different degrees of nonlinearity through codebook vectors. K-means clustering is the extreme case

that uses only codebook vectors and no manifolds; PCA is the other extreme with only manifolds and no codebook vectors; and SOMs are somewhere in between. The extensive preface to the collection volume *Principal Manifolds for Data Visualization and Dimension Reduction* (Gorban et al. 2008) gives a readable intro to this interesting field.

In today's deep learning practice one often ignores the traditional methods treated in this section. Instead one immediately fires the big cannon, training a deep neural network wired up in an *auto-encoder* architecture. An autoencoder network is a multilayer feedforward network whose ouput layer has the same large dimension n as the input layer. It is trained in a supervised way, using training data  $(\mathbf{x}_i, \mathbf{y}_i)$  to approximate the identity function: the training output data  $\mathbf{y}_i$  are identical to the input patterns  $\mathbf{x}_i$  (possibly up to some noise added to the inputs). The trick is to insert a "bottleneck" layer with only  $m \ll n$  neurons into the layer sequence of the network. In order to achieve a good approximation of the *n*-dimensional identity map on the training data, the network has to discover an  $n \rightarrow m$ -dimensional compression mapping which preserves most of the information that is needed to describe the training data points. I will not give an introduction to autoencoder networks in this course (it's a topic for the "Deep Learning" course given by Mathia Sabatelli). The Deep Learning standard reference (now already a classic) of I. Goodfellow, Bengio, and Courville 2016 has an entire section on autoencoders.

# 5 Discrete symbolic versus continuous real-valued

I hope this section will be as useful as it will be short and simple. Underneath it, however, lurks a mysterious riddle of mathematics, philosophy and the neurosciences.

Some data are given in symbolic form, for instance

- texts of all sorts,
- yes/no or rating scale questionnaire items,
- DNA and protein sequence data,
- records of chess or Go matches,
- large parts in public administration databases.

Others are real-valued (scalar, vector, matrix or array formatted) in principle, notwithstanding the circumstance that on digital machines real numbers must be approximated by finite-length bitstrings. Examples of data that are best understood as "continuous" are

- images, video, speech,
- measurement data from technical systems, e.g. in production process control or engine monitoring,
- financial data,
- environmental and weather data,
- signals used in robot motion control.

Many mathematical formalisms can be sorted in two categories: "discrete" and "continuous". Mathematicians, in fact, come in two kinds – ask one, and he/she will be able to tell you whether he/she feels more like a discrete or continuous mathematician. Discrete sorts of mathematics and discrete mathematicians typically operate in areas of number theory, set theory, logic, algebra, graph theory, automata theory. Continuous maths / mathematicians use linear algebra, functional analysis and calculus.

Computer scientists are almost always discrete-math-minded: 0 and 1 and algorithms and Boolean circuits and AI knowledge representation formalisms are eminently discrete.

Physicists are almost always continuous-math minded, because their world is made of continuous space, time, matter, forces and fields.

There are some mathematical domains that are neither, or open to both, especially topology and probability. Some of the most advanced and ingenious modern fields of mathematics arise from crossover formalisms between the Discrete and the Continuous. The fundamental difference between the two is not dissolved in these theories, but the tension between the Discrete and the Continuous sets free new forms of mathematical energy. Sadly, these lines of research are beyond what I understand and what I can explain (or even name), and certainly beyond what is currently used in machine learning.

The hiatus (an educated word of latin origin, meaning "dividing gap") between the Discrete and the Continuous is also the source of one of the great unresolved riddles in the neurosciences, cognitive science and AI: how can symbolic reasoning (utterly discrete) emerge from the continuous matter and signal processing in our material brains (very physical, very continuous)? This question has kept AI researchers and philosophers busy (and sometimes aggressively angry with one another) for 5 decades now and is not resolved; if you are interested, you can get a first flavor in the Wikipedia article on "Physical symbol system" or by reading up on the overview articles listed in http://www.neural-symbolic.org/.

Back to our down-to-earth business. Machine learning formalisms and algorithms likewise are often either discrete-flavored or continuous-flavored. The former feed on symbolic data and create symbolic results, using tools like decision trees, Bayesian networks and graphical models (including hidden Markov models), inductive logic, and certain sorts of neural networks where neurons have 0-1-valued activations (Hopfield networks, Boltzmann machines). The latter digest vector data and generate vector output, like neural networks, support vector machines and various sorts of regression learning "machines".

The great built-in advantage of discrete formalisms is that they often lend themselves well to *explainable*, human-understandable solutions. Their typical disadvantage is that learning or inference algorithms are often based on combinatorial search, which quickly lets computing times explode. In contrast, continuous formalisms typically lead to results that cannot be intuitively interpreted – vectors don't talk – but lend themselves to nicely, smoothly converging optimization algorithms.

When one speaks of "machine learning" today, one mostly has vector processing methods in mind. Also this RUG course is focussing on vector data. The discrete strands of machine learning are more associated with what one often calls "data mining". This terminology is however not clearly defined (see Section 1.5).

Sometimes one has vector data but wants to exploit benefits that come with discrete methods, or conversely, one has symbolic data and wants to use a neural network (because everybody else seems to be using them, or because one doesn't want to fight with combinatorial explosions). Furthermore, many an interesting dataset comes as a mix of symbolic-discrete and numerical-continuous data – for instance, data originating from questionnaires or financial/business/admin data often are mixed-sort.

Then, one way to go is to *convert* discrete data to vectors or vice versa. It is a highly empowering professional skill to know about basic methods of discrete  $\leftrightarrow$ 

continuous conversions.

Here are some discrete-to-continuous transformations:

- **One-hot encodings.** Given: data points  $a_{\nu}$  that are symbols from a finite "alphabet"  $A = \{a_1, \ldots, a_k\}$ . Examples: yes/no answers in a questionnaire; words from a vocabulary; nucleic acids A, C, G, T occuring in DNA. Turn each  $a_{\nu}$  into the k-dimensional binary vector  $v_{\nu} \in \{0, 1\}^k$  which is zero everywhere except at position  $\nu$ . This is a very common way to present symbolic input to neural networks. On the output side of a neural network (or any other regression learning machine), one-hot encodings are almost always used to give vector teacher data in classification tasks: if  $(\mathbf{x}_i, c_i)$  is a classification-task training dataset, where  $c_i$  is a symbolic class label from a class set  $C = \{c_1 \ldots, c_k\}$ , transform each  $c_i$  to its k-dimensional one-hot vector  $v_i$  and get a purely vector-type training dataset  $(\mathbf{x}_i, v_i)$ .
- **Binary pattern encodings.** If the symbol alphabet A is of a large size k, one might shy away from one-hot encoding because it gives large vectors. Instead, encode each  $a_{\nu}$  into a binary vector of length  $\lceil \log_2 k \rceil$ , where  $\lceil \log_2 k \rceil$  is the smallest integer larger or equal to  $\log_2 k$ . For example, the alphabet  $\{a, b, c, d\}$  would be encoded in the vectors  $\{[0, 0]', [0, 1]', [1, 0]', [1, 1]'\}$  Advantage: small vectors. Disadvantage: subsequent vector-processing algorithms must invest substantial nonlinear effort into decoding this essentially arbitrary encoding. This will often be crippling, and binary pattern encodings should only be considered if there is some intuitive logic in the encoding.
- **Linear scale encoding.** If there is some natural ordering in the symbols  $a_{\nu} \in A$ , encode every  $a_{\nu}$  by the index number  $\nu$ . Makes sense, for instance, when the  $a_{\nu}$  come from a Likert-scale questionnaire item, as in
  - $A = \{$ certainly not, rather not, don't know, rather yes, certainly yes $\},\$

which could be turned into a sequence of real numbers like  $\{-1.0, -0.5, 0, 0.5, 1.0\}$  in a way that allows neural networks to 'feel' the semantics of the ordering in the questionnaire answer options.

Semantic word vectors. This is a technique which enabled breakthroughs in deep learning for text processing (I mentioned it in the Introduction; it is used in the TICS demo). By a nontrivial machine learning algorithm pioneered by Mikolov et al. 2013, convert words  $a_{\nu}$  from a (large) vocabulary A into vectors of some reasonably large size, say 300-dimensional, such that semantically related words become mapped to vectors that are metrically close to each other.

For instance, code vectors v, v' for words w = airplane and w' = aircraftshould have small distance, wheras the code vector v'' for w'' = rose should lie at a great distance to both v and v'. So the goal is to find vectors v, v', v'' in this example that have small distance ||v-v'|| and large distances ||v-v''||, ||v'-v''||. This is achieved by measuring the semantic similarity of two words w, w' through counting how often they occur in similar locations in training texts. A large collection of English texts is processed, collecting statistics about similar sub-phrases in those texts that differed only in the two words whose similarity one wished to assess (plus, there was another trick: can you think of an important improvement of this basic idea?). The current de-facto standard algorithm for learning a vector embedding for words is documented in Devlin et al. 2018.

And here are some continuous-to-discrete transformations:

**One-dimensional discretization, also known as binning.** Given: a real number dataset  $\{x_i\}$  where  $x_i \in \mathbb{R}$ . Divide the real line into k segments

$$A_1, A_2, \dots, A_{k-1}, A_k = (-\infty, a_1], (a_1, a_2], \dots, (a_{k-2}, a_{k-1}], (a_{k-1}, \infty).$$

Then transform each  $x_i$  to that symbol  $A_{\nu}$  for which  $x_i \in A_{\nu}$ . Comments:

- The segments  $(a_{\nu}, a_{\nu+1}]$  are called *bins*.
- In the simplest case, the range  $[\min\{x_i\}, \max\{x_i\}]$  of the data points is split into k bins of equal length.
- Alternatively, another way of splitting the range  $[\min\{x_i\}, \max\{x_i\}]$  is to define the bins such that each bin will contain an approximately equal number of data points from the set  $\{x_i\}$ .
- Often the best way of splitting the data range into bins depends on the task. This can turn out to be a delicate optimization problem. For instance, there is a sizeable literature in deep learning which is concerned with the question of how one can reduce the numerical precision of weight parameters in neural networks. One first trains the neural network using standard high-precision learning algorithms, obtaining weight matrices in floating-point precision. In order to speed up computing, or to use small, limited-precision microchips on handheld devices, or to reduce energy consumption, one wishes to reduce the bit precision of the weight matrices. In extreme cases, one wants to end up with only three weight values  $\{-1, 0, 1\}$ , which would lead to very cheap neural network computations at use-time. This amounts to the splitting the range of floating-point precision weights obtained after training into just three bins. The goal is to ensure that the reducedprecision network performs approximately as well as the high-precision original. This is a nontrivial problem. Check out the (randomly chosen) reference Indiveri 2015 if you want to get a taste.
- Discretizing a continuous value range into bins with adaptive bin boundaries is the key for using decision trees (which are discrete models) for continuous attribute ranges.

- Multi-dimensional discretization by hierarchical refinement. If one wants to discretize a set  $\{\mathbf{x}_i\}$  of n-dimensional vectors, one has to split the ndimensional volume which contains the points  $\{\mathbf{x}_i\}$  into a finite set of discrete regions  $R_{\nu}$ . A common approach is to let these regions be *n*-dimensional hypercubes. By a process of hierarchical refinement one constructs these regions  $R_{\nu}$  such that in areas where there is a higher point density, or in areas where there is much fine-grained information encoded in the local point distribution, one uses smaller hypercubes to increase resolution. This leads to a tree structure, which in the case n = 2 is called a *quadtree* (because every non-leaf node has four children) and in the case n = 3 an octree of hierarchically nested hypercubes. This tree structure enables a computationally efficient indexing of the hypercubes. The left panel in Figure 27 shows an example. The Wikipedia article on quadtrees is quite instructive. One may also opt for regions  $R_{\nu}$  of other polygonal shape (see right panel in Figure 27). There are many such *mesh refinement* methods, with task-specific optimization criteria. They are not typically used in machine learning but rather in methods for simulating spatiotemporal (fluid) dynamics by numerically solving PDEs. Still, it is good to know that such methods exist.
- **Vector quantization.** Using K-means or other clustering algorithms, the vector set  $\{\mathbf{x}_i\}$  is partitioned into k cells whose center of gravity vectors are indexed and the indices are used as symbolic encodings of the  $\{\mathbf{x}_i\}$ . We have seen this in Section 4.2. This *is* a typical machine learning method for discretization.
- **Turning neural dynamics into symbol sequences.** When we (I really mean "we", = us humans!) speak or write, the continuous-time, continuous-valued neural brain dynamics leads to a discrete sequence of words. Somehow, this symbolic sequence is encoded in the "subsymbolic", continuous brain dynamics. It is unknown how, exactly, this encoding is realized. Numerous proposals based on nonlinear dynamical systems theory have been made. This is an area of research in which I am personally engaged. If you are interested: some approaches are listed in Durstewitz, Seamans, and Sejnowski 2000, Pascanu and Jaeger 2011, Fusi and Wang 2016. In machine learning, the problem of transforming continuous-valued neural state sequences to sequences of words (or letters) arises in applications like speech recognition ("speech-to-text") or gesture recognition. Here the most common solution (which is not necessarily biologically plausible) is to use the core neural network to generate a sequence of continuous-valued hypothesis output vectors with as many components as there are possible target symbols. At each point in time, the numerical value of each vector component reflects a current "degree of belief" about which symbol should be generated. With some postprocessing mechanism (not easy to set up), this hypothesis stream is denoised and turned into a symbol sequence, for instance by selecting at each point in time the symbol that has the largest degree of belief.



Figure 27: Left: A hierarchical hypercube mesh for a 2-dimensional dataset consisting of points homogeneously distributed inside a circle. Right: adaptive non-orthogonal meshing, here used in an airflow simulation. Sources: left panel: http://www.questinygroup.com/tag/quad-tree/; right panel: Luchinsky and al. 2012.

# 6 The bias-variance dilemma and how to cope with it

Reality is rich in detail and surprise. Measuring reality gives a finite number of data points – and the reality between and beyond these data points remains unmeasured, a source of unforseeable surprises. Furthermore, measuring is almost always "noisy" – imprecise or prone to errors. And finally, each space-time event point of reality can be measured in innumerable ways, of which only a small choice is actually measured. For instance, the textbook measurement of "tossing a coin" events is just to record "heads" or "tail". But one *could* also measure the weight and color of coin, the wind speed, the falling altitude, the surface structure of the ground where the coin falls, and ... everything else. In summary, any dataset will capture only a supremely scarce coverage of space-time events whose unfathomable qualitative richness has been reduced to a ridiculously small number of "observables". From this empoverished, punctuated information basis, called "training data", a machine learning algorithm will generate a "model" of a part of reality. This model will then be used to predict properties of *new* space-time events, called "test inputs". How should it be ever possible for a model to "know" anything about those parts of reality that lie between the pinpoints hit by the training data?

This would be a good point to give up.

But ... quoting Einstein: "Subtle is the Lord, but malicious He is not" (https://en.wikiquote.org/wiki/Albert\_Einstein). Reality is co-operative. Between measurement points, reality changes not arbitrarily but in a lawful manner. The



Figure 28: He gave the reason why machine learning works (from https://commons.wikimedia.org/wiki/File:Albert\_Einstein\_-\_Colorized.jpg)

question is, *which* law. In machine learning, this question is known as the problem of *overfitting*, or in more educated terms, the *bias-variance dilemma*. Welcome to this chapter which is all about this question. For all your practical exploits of machine learning in your professional future, this is the most empowering chapter in this course.

### 6.1 Training and testing errors

Let us take a closer look at the archetypical machine learning task of classifying handwritten digits. We use the simple Digits benchmark data that you know from Section 4.1. The task is specified as follows:

- **Training data.**  $(\mathbf{x}_i^{\text{train}}, c_i^{\text{train}})_{i=1,...,1000}$  where the  $\mathbf{x}_i^{\text{train}} \in [0, 1]^{240}$  are 240-dimensional image vectors, whose components correspond to the 15 × 16 pixels which have normalized grayscale values ranging in [0, 1]; and where the  $c_i^{\text{train}}$  are class labels from the set  $\{0, 1, \ldots, 9\}$ . There are 100 examples from each class in the training data set.
- **Test data.** Another set  $(\mathbf{x}_i^{\text{test}}, c_i^{\text{test}})_{i=1,\dots,1000}$ , also having 100 exemplars of each digit.
- **Task specification.** This is a benchmark dataset with a standardized task specification: train a classifier using (only) the training data, then test it on the test data and report the rate of misclassifications. The loss function is thus the count loss given in Equation 4.

An elementary yet professionally structured learning pipeline, which uses methods that we have already described in this course, goes as follows (it may be a welcome rehearsal to give a step-by-step instruction): First stage: dimension reduction by PCA: 1. Center the set of image vectors  $(\mathbf{x}_i^{\text{train}})_{i=1,\dots,1000}$  by subtracting the mean vector  $\mu$ , obtaining  $(\bar{\mathbf{x}}_i^{\text{train}})_{i=1,\dots,1000}$ .

- 2. Assemble the centered image vectors column-wise in a  $240 \times 1000$  matrix  $\overline{X}$ . Compute the correlation matrix C = 1/1000 X X' and factorize C into its SVD  $C = U\Sigma U'$ . The columns of U are the 240 principal component vectors of C.
- 3. Decide how strongly you want to reduce the dimension, shrinking it from n = 240 to m < n. Let  $U_m$  be the matrix made from the first m columns from U.
- 4. Project the centered patterns  $\bar{\mathbf{x}}_i^{\text{train}}$  on the *m* first principal components, obtaining *m*-dimensional feature vectors  $\mathbf{f}_i^{\text{train}} = U'_m \bar{\mathbf{x}}_i^{\text{train}}$ .
- Vectorize the class labels by one-hot encoding: Each  $c_i^{\text{train}}$  is re-written as a binary 10-dimensional vector  $v_i^{\text{train}}$  which has a 1 entry in the corresponding class  $c_i$ . Assemble these vectors column-wise in a 10 × 1000 matrix V.
- **Compute a linear regression classifier:** Assemble the 1000 *m*-dimensional feature vectors  $\mathbf{f}_i^{\text{train}}$  into a  $m \times 1000$  dimensional matrix F and obtain a  $10 \times m$  dimensional regression weight matrix W by

$$W' = (F F')^{-1} F V'.$$

**Compute the training MSE and training error rate:** The training mean square error (MSE) is given by

$$\text{MSE}^{\text{train}} = 1/1000 \ \sum_{i=1}^{1000} \ \|v_i^{\text{train}} - W(\mathbf{f}_i^{\text{train}})\|^2.$$

The training misclassification rate is

$$\varrho^{\text{train}} = 1/1000 |\{i | \text{maxInd} (W \mathbf{f}_i^{\text{train}}) - 1 \neq c_i^{\text{train}}\}|,$$

where maxInd picks the index of the maximal element in a vector. (Note: the "-1" is owed to the fact that the vector indices range from 1 to 10, while the class labels go from 0 to 9).

Similarly, compute the test MSE and error rates by

$$\text{MSE}^{\text{test}} = 1/1000 \ \sum_{i=1}^{1000} \ \|v_i^{\text{test}} - W(\mathbf{f}_i^{\text{test}})\|^2$$

and

$$\varrho^{\text{test}} = 1/1000 |\{i \mid \text{maxInd} (W \mathbf{f}_i^{\text{test}}) - 1 \neq c_i^{\text{test}}\}|,$$

where  $\mathbf{f}_i^{\text{test}} = U'_m (\mathbf{x}_i^{\text{test}} - \mu)$ . Note: for centering the test data, use the mean  $\mu$  obtained from the training data!

This is a procedure made from basic linear operations that you should master even when sleepwalking; with some practice the entire thing should not take you more than 30 minutes for programming and running. Altogether a handy quick-and-not-very-dirty routine that you should consider carrying out in every classification task, in order to get a baseline before you start exploring more sophisticated methods.

And now - let us draw what is probably the most helpful graphics in these lecture notes, worth to be burnt into your subconscious. Figure 29 shows these diagnostics for all possible choices of the number  $m = 1, \ldots, 240$  of PC features used.



Figure 29: Dashed: Train (blue) and test (red) MSE obtained for m = 1, ..., 240 PCs. Solid: Train (blue) and test (red) misclassification rates. The *y*-axis is logarithmic base 10.

This plot visualizes one of the most important issues in (supervised) machine learning and deserves a number of comments.

• As m increases (the x-axis in the plot), the number of parameters in the corresponding linear regression weight matrices W grows by  $10 \cdot m$ . More model parameters means more degree of freedoms, more "flexible" models. With greater m, models can increasingly better solve the linear regression learning equation (20). This is evident from the monontonous decrease of the train MSE curve. The training misclassification rate also decreases persistently except for a jitter that is due to the fact that we optimized models only indirectly for low misclassification.

- The analog performance curves for the testing MSE and misclassification first exhibit a decrease, followed by an increasing tail. The testing misclassification rate is minimal for m = 34.
- This "first decrease, then increase" behavior of testing MSE (or classification rate) is *always* observed in supervised learning tasks when models are compared which have growing degrees of data fitting flexibility. In our digit example, this increase in flexibility was afforded by growing numbers of PC features, which in turn gave the final linear regression a richer repertoire of feature values to combine into the hypothesis vectors.
- The increasing tail of testing MSE (or classification rate) is the hallmark of *overfitting*. When the learning algorithm admits too much flexibility, the resulting model can fit itself not only to what is "lawful" in the training data, but also to the random fluctuations in the training data. Intuitively and geometrically speaking, a learning algorithm that can shape many degrees of freedom in its learnt models allows the models to fold in curls and wiggles to accomodate the random whims of the training data. But then, the random curls and wiggles of the learnt model will be at odds with fresh testing data.

#### 6.2 The menace of overfitting – it's real, it's everywhere

ML research has invented a great variety of model types for an even larger variety of supervised and unsupervised learning tasks. How exactly overfitting (mal-)functions in each of these model types will need a separate geometrical analysis in each case. Because overfitting is such a fundamental challenge in machine learning, I illustrate its geometrical manifestations with four examples.

#### 6.2.1 Example 1: polynomial curve-fitting

This example is the standard textbook example for demonstrating overfitting. Let us consider a one-dimensional input, one-dimensional output regression task of the kind where the training data are of form  $(x_i, y_i) \in \mathbb{R} \times \mathbb{R}$ . Assume that there is some systematic relationship y = h(x) that we want to recover from the training data. We consider a simple artificial case where the  $x_i$  range in [0, 1] and the to-be-discovered true functional relationship is  $y = \sin(2\pi x)$ . The training data, however, contain a noise component, that is,  $y_i = \sin(2\pi x_i) + \nu_i$ , where  $\nu_i$  is drawn from a normal distribution with zero mean and standard deviation  $\sigma$ . Figure 30 shows a training sample  $(x_i, y_i)_{i=1,\dots,11}$ , where N = 11 training points  $x_i$  are chosen equidistantly.

We now want to solve the task of learning a good approximation for h from the training data  $(x_i, y_i)$  by applying polynomial curve fitting, an elementary technique you might be surprised to meet here as a case of machine learning. Consider an



Figure 30: An example of training data (red squares) obtained from a noisy observation of an underlying "correct" function  $sin(2\pi x)$  (dashed blue line).

m-th order polynomial

$$p(x) = w_0 + w_1 x + \dots + w_m x^m.$$
(36)

We want to approximate the function given to us via the training sample by a polynomial, that is, we want to find ("learn") a polynomial p(x) such that  $p(x_i) \approx y_i$ . More precisely, we want to minimize the mean square error on the training data

MSE<sup>train</sup> = 
$$\frac{1}{N} \sum_{i=1}^{N} (p(x_i) - y_i)^2$$
.

Here we don't bother how this task is solved computationally. We simply rely on the Matlab function *polyfit* which does exactly this job for us: given data points  $(x_i, y_i)$  and polynomial order m, find the coefficients  $w_j$  which minimize this MSE. Figure 31 shows the polynomials found in this way for m = 1, 3, 10.



Figure 31: Fitting polynomials (green lines) for polynomial orders 1, 3, 10 (from left to right).

If we compute the MSE's for the three orders m = 1, 3, 10, we get  $MSE^{train} = 0.4852, 0.0703, 0.0000$  respectively. Some observations:

- If we increase the order m, we get increasingly lower  $MSE^{train}$ .
- For m = 1, we get a linear polynomial, which apparently does not represent our original sine function well (underfitting).
- For m = 3, we get a polynomial that hits our target sine apparently quite well.
- For m = 10, we get a polynomial that perfectly matches the training data, but apparently misses the target sine function (overfitting).

The modelling flexibility is here defined through the polynomial order m. If it is too small, the models are too inflexible and underfit; if it is too large, we see overfitting.

However, please switch now into your most critical thinking mode and reconsider what I have just said. *Why*, indeed, should we judge the linear fit "underfitting", the order-3 fit "seems ok", and the order-10 fit "overfitting"? There is no other ground for justifying these judgements than our visual intuition and our secret mastermind knowledge that the underlying pure function was this sinewave! In fact, the order-10 fit might be the right one if the data contain no noise! the order-1 fit might be the best one if the data contain a lot of noise! If we didn't know beforehand that the true noiseless function was the sinewave, then *we could not know* which of the three cases from Figure 31 gives us the best model!

#### 6.2.2 Example 2: pdf estimation

Let us consider the task of estimating a 2-dimensional pdf over the unit square from 6 given training data points  $\{x_i\}_{i=1,\dots,6}$ , where each  $x_i$  is in  $[0,1] \times [0,1]$ . This is an elementary unsupervised learning task, the likes of which frequently occur as a subtask in more involved learning tasks, but which is of interest in its own right too. Figure 32 shows three pdfs which were obtained from three different learning runs with models of increasing flexibility (I don't explain the modeling method here — for the ones who know about it: simple Gaussian Parzen-window models where the degree of admitted flexibility was tuned by kernel width — will be explained in Section 7). Again we encounter the fingerprints of under/overfitting: the low-flexibility model seems too "unbending" to resolve any structure in the training point cloud (underfitting), the high-flexibility model is so volatile that it can accomodate each individual training point (presumably overfitting).

But again, we don't really know...

#### 6.2.3 Example 3: learning a decision boundary

Figure 33 shows a schematic of a classification learning task where the training patterns are points in  $\mathbb{R}^2$  and come in two classes. When the trained model is too inflexible (left panel), the decision boundary is confined to be a straight line,



Figure 32: Estimating a pdf from 6 data points. Model flexibility grows from left to right. Note the different scalings of the z-axis: the integral of the pdf is 1 in each of the three cases.

presumably underfitting. When the flexibility is too large, each individual training point can be "lasso-ed" by a sling of the decision boundary, presumably overfitting.

Do I need to repeat that while these graphics *seem* to indicate under- or overfitting, we do not actually *know*?



Figure 33: Learning a decision boundary for a 2-class classification task of 2dimensional patterns (marked by black "x" and red "o").

#### 6.2.4 Example 4: furniture design

Overfitting can also hit you in your sleep, see Figure 34.

Again, while this looks like drastic overfitting, it would be just right if all humans sleep in the same folded way as the person whose sleep shape was used for training the matrass model.

#### 6.2.5 Interim summary

The four examples stem from different learning tasks (function approximation, pdf learning, classification learning, furniture optimization), and correspondingly the



Figure 34: A nightmare case of overfitting. Picture spotted by Yasin Cibuk (2018 ML course participant) on http://dominicwilcox.com/portfolio/bed/ – now no longer accessible –, designed and crafted by artist Dominic Wilcox, 1999. Quote from the artist's description of this object: "I used my own body as a template for the mattress". From a ML point of view, this means a size N = 1 training data set.

overfitting problem manifests itself in different geometrical ways. But the flavor is the same in all cases. The model flexibility determines how "wiggly" the geometry of the learnt model can become. Very large flexibility ultimately admits to fold the model snugly around each individual training point. This leads to small, even zero, training error; but it is likely disastrous for generalization to new test data points. Very low flexibility can hardly adapt to the structure of the training data at all, likewise leading to poor test performance (and poor training performance too). Some intermediate flexibility will strike the best balance.

Properly speaking, flexibility is not a characteristic of the final model that one obtains after learning, but of the learning algorithm. If one uses the term precisely, one should speak, for example, of "the flexibility of the procedure to train a third-order polynomial function", or of "the flexibility of the PCA-based linear regression learning scheme which uses m PCA features".

I introduced this way of speaking about "flexibility" ad hoc. In statistics and machine learning textbooks you will hardly find this term. Instead, specific methods to measure and tune the flexibility of a learning algorithm have their specific names, and it is these names that you will find in the literature. The most famous among them is *model capacity*. This concept has been developed in a field now called *statistical learning theory*, and (not only) I consider it a highlight of modern ML theory. We will however not treat the concept of model capacity in this course, since it is not an easy concept, and it needs to be spelled out in different versions for different types of learning tasks and algorithms. Check out en.wikipedia.org/wiki/Vapnik-Chervonenkis\_theory if you want to get an impression. Instead, in Sections 6.4.1 and 6.4.2 I will present two simpler methods for handling modeling flexibility which, while they lack the analytical beauty and depth of the model capacity concept, are immensely useful in practice.

I emphasize that finding the right flexibility for a learning algorithm is ab-so-lute-ly crucial for good performance of ML algorithms. Our little visual examples do not do justice to the dismal effects that overfitting may have in real-life learning tasks where a high dimension of patterns is combined with a small number of training examples — which is a situation faced very often by ML engineers in practical applications.

#### 6.3 An abstract view on supervised learning

Before I continue with the discussion of modeling flexibility, it is helpful to introduce some standard theoretical concepts and terminology. Before you start reading this section, make sure that you understand the difference between the *expectation* of a random variable and the *sample mean* (explained in Appendix D).

In supervised learning scenarios, one starts from a training sample of the form  $(x_i, y_i)_{i=1,...,N}$ , which is drawn from a joint distribution  $P_{X,Y}$  of two random variables X and Y. So far, we have focussed on tasks where the  $x_i$  were vectors and the  $y_i$  were class labels or numbers or vectors, but supervised learning tasks can be defined for any kind of variables. In this subsection we will take an abstract view and just consider any kind of supervised learning based on a training sample  $(x_i, y_i)_{i=1,...,N}$ . We then call the  $x_i$  the arguments and the  $y_i$  the targets of the learning task. The target RV Y is also sometimes referred to as the teacher variable.

The argument and target variables each come with their specific data value space – a set of possible values that the RVs X and Y may take. For instance, in our digit classification example, the data value space for X was  $\mathbb{R}^{240}$  (or more constrained,  $[0, 1]^{240}$ ) and the data value space for Y was the set of class labels  $\{0, 1, \ldots, 9\}$ . After the extraction of m features and turning the class labels to class indicator vectors, the original picture–label pairs  $(x_i, y_i)$  turned into pairs  $(\mathbf{f}_i, v_i)$  with data value spaces  $\mathbb{R}^m$ ,  $\{0, 1\}^k$  respectively. In English-French sentence translation tasks, the data value spaces of the random variables X and Y would be a (mathematical representation of a) set of English and French sentences. We now abstract away from such concrete data value spaces and write  $\mathcal{E}_X, \mathcal{E}_Y$  for the data value spaces of X and Y.

Generally speaking, the aim of a supervised learning task is to derive from the training sample a function  $h : \mathcal{E}_X \to \mathcal{E}_Y$ . We called this function a *decision* function earlier in these lecture notes, and indeed that is the term that is used in statistical learning theory.

Some of the following I already explained before (Section 2.3) but it will be

helpful if I repeat this here with more detail.

The decision function h obtained by a learning algorithm should be optimized toward some objective. One introduces the concept of a *loss function*. A loss function is a function

$$L: \mathcal{E}_Y \times \mathcal{E}_Y \to \mathbb{R}^{\ge 0}. \tag{37}$$

The idea is that a loss function measures the "cost" of a mismatch between the target values y and the values h(x) returned by a decision function. Higher cost means lower quality of h. We have met two concrete loss functions so far:

• A loss that counts misclassifications in pattern classification: when the decision function returns a class label, define

$$L(h(x), y) = \begin{cases} 0, & \text{if } h(x) = y \\ 1, & \text{if } h(x) \neq y \end{cases}$$
(38)

• A loss that penalizes quadratic errors of vector-valued targets:

$$L(h(x), y) = ||h(x) - y||^2.$$
(39)

This loss is often just called "quadratic loss". We used it as a basis for deriving the algorithm for linear regression in Section 3.1.

The decision function h is the outcome of a learning algorithm, which in turn is informed by a sample  $(x_i, y_i)_{i=1,...,N}$ . Learning algorithms should minimize the *expected* loss, that is, a good learning algorithm should yield a decision function h whose *risk* 

$$R(h) = E[L(h(X), Y)]$$
(40)

is small. The expectation here is taken with respect to the true underlying joint distribution  $P_{X,Y}$ . For example, in a case where X and Y are numerical RVs and their joint distribution is described by a pdf f, the risk of a decision function h would be given by

$$R(h) = \int_{\mathcal{E}_X \times \mathcal{E}_Y} L(h(x), y) f(x, y) d(x, y)$$

However, the true distribution f is unknown. The mission to find a decision function h which minimizes (40) is, in fact, hopeless. The only access to  $P_{X,Y}$  that the learning algorithm affords is the scattered reflection of  $P_{X,Y}$  in the training sample  $(x_i, y_i)_{i=1,...,N}$ .

A natural escape from this impasse is to set up a learning algorithm such that instead of attempting to minimize the risk (40) it tries to minimize the *empirical risk* 

$$R^{\rm emp}(h) = 1/N \sum_{i=1}^{N} L(h(x_i), y_i), \qquad (41)$$

which is just the mean loss calculated over the training examples. Minimizing this empirical risk is an achievable goal, and a host of optimization algorithms for all kinds of supervised learning tasks exist which do exactly this, that is, they find

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \quad 1/N \sum_{i=1}^{N} L(h(x_i), y_i).$$

$$(42)$$

The set  $\mathcal{H}$  is the *hypothesis space* – the search space within which a learning algorithm may look for an optimal h.

It is important to realize that every learning algorithm comes with a specific hypothesis space. For instance, in decision tree learning  $\mathcal{H}$  is the set of all decision trees that use a given set of properties and attributes. Or, in linear regression,  $\mathcal{H}$  is the set of all affine linear functions from  $\mathbb{R}^n$  to  $\mathbb{R}^k$ . Or, if one sets up a neural network learning algorithm,  $\mathcal{H}$  is typically the set of all neural networks that have a specific connection structure (number of neuron layers, number of neurons per layer); the networks in  $\mathcal{H}$  then differ from each other by the weights associated with the synaptic connections.

The empirical risk is often – especially in numerical function approximation tasks – also referred to as *training error*.

While minimizing the empirical loss is a natural way of coping with the impossibility of minimizing the risk, it may lead to decision functions that combine a low empirical risk with a high risk. This is the ugly face of overfitting which I highlighted in the previous subsection. In extreme cases, one may learn a decision function which has zero empirical risk and yet has a extremely large expected testing error, which makes it absolutely useless.

There is no easy or general solution for this conundrum. It has spurred statisticians and mathematicians to develop a rich body of theories which analyze the relationships between risk and empirical risk, and suggest insightful strategies to manage as well as one can in order to keep the risk within provable bounds. These mathematical theories, sometimes referred to as *statistical learning theory* (or better, theor*ies*), are beyond the scope of this lecture.

If you are in a hardcore mood and if you have some background in probability theory, you can inspect Section 19 of lecture notes of my legacy "Principles of Statistical Modeling" course (online at https://www.ai.rug.nl/minds/uploads/ LN\_PSM.pdf). You will find that the definitions of loss and risk given there in the spirit of mathematical statistics are a bit more involved than what I presented above, but the definitions of loss and risk that I gave here are used in textbooks on machine learning.

#### 6.4 Tuning model flexibility

In order to deal with the overfitting problem, one must have ways to tune the flexibility of the learning algorithm and set it to the "right" value.

Let us briefly return to the overfitting diagram in Figure 29. In that demo, the flexibility regulation (moving left or right on the horizontal axis) was effected by moving in a model class inclusion hierarchy — more flexible learning algorithms were more flexible because they searched larger hypothesis spaces  $\mathcal{H}$ . But similar diagrams would be obtained in any other ML exercise where other methods for navigating on the flexibility axis might be used. Because the essential message of Figure 29 is universal across all machine learning tasks, I redraw that figure and annotate it generically (Figure 35).



Figure 35: The generic, universal, core challenge of machine learning: finding the right model flexibility which gives the minimal risk.

Let us use the arbitrary symbol m for the degree of flexibility — we used m before in the special cases in the Digits example (Figure 29), where m meant the number of extracted principal components; and and the polynomial fit example (Equation 36), where m meant the highest used polynomial order. Abstracting from these special cases, we use m as a general modeling flexibility indicator - the larger m, the more flexibly is a learning algorithm wired up to nestle its models around training data points.

Summarizing and emphasizing the main messages of this figure:

- Increasing the model flexibility leads to monotonous decrease of the empirical risk because training data can be fitted better and better.
- Increasing the model flexibility from very low to very high will lead to a risk that first decreases (less and less underfitting) and then rises again (more and more overfitting).
- The best model flexibility is where the risk curve reaches its minimum. The problem is that one does not know this precious risk curve it is defined on the distribution of all future "test" data, which one does not have at training time.

There are many, quite different, ways of tuning flexibility of a learning algorithm. Note that the word "flexibility" is only intuitive; how this is concretely formalized, implemented and measured differs between the methods of adjusting this "flexibility". I proceed to outline the most common ones.

#### 6.4.1 Tuning learning flexibility through model class size

In the Digits classification demo from Section 6.1, we tuned flexibility by changing the dimension of the PCA features. In the polynomial curve fitting demo in Section 6.2.1 we changed the order of polynomials. Let us re-consider these two examples to get a clearer picture:

- In the Digits example (Figure 29) we found that the number m of principal component features that were extracted from the raw image vectors was decisive for the testing error. When m was too small, the resulting models were too simple to distinguish properly between different digit image classes (underfitting). When m was too large, overfitting resulted. Fixing a particular m determines the class H of candidate decision functions within which the empirical risk (42) is minimized. Specifically, using a fixed m meant that the optimal decision function h<sub>opt</sub> was selected from the set H<sub>m</sub> which contains all decision functions which first extract m principal component features before carrying out the linear regression. It is clear that H<sub>m-1</sub> is contained in H<sub>m</sub>, because decision functions that only combine the first m − 1 principal component features into the hypothesis vector can be regarded as special cases of decision functions that combine m principal component features into the hypothesis vector, namely those whose linear combination weight for the m-th feature is zero.
- In the polynomial curve fitting example from Section 6.2.1, the model parameters were the monomial coefficients  $w_0, \ldots, w_m$  (compare Equation 36). After fixing the polynomial order m, the optimal decision function p(x) was selected from the set  $\mathcal{H}_m = \{p : \mathbb{R} \to \mathbb{R} \mid p(x) = \sum_{j=0}^m w_j x^j\}$ . Again it is clear that  $\mathcal{H}_{m-1}$  is contained in  $\mathcal{H}_m$ .

A note on terminology: I use the words "decision function" and "model" as synonyms, meaning the (classification) algorithm h which results from a learning procedure. The word "decision function" is standard in statistics, the word "model" is more common in machine learning.

Generalizing from these two examples, we are now in a position to draw a precise picture of what it may mean to consider learning algorithms of "increasing flexibility". A model class inclusion sequence is a sequence  $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \ldots \subset \mathcal{H}_L$  of sets of candidate models. Since there are more candidate models in classes that appear later in the sequence, "higher" model classes have more possibilities to fit the training data, thus the optimal model within class  $\mathcal{H}_{m+1}$  can achieve an empirical risk that is at least as low as the optimal model in class  $\mathcal{H}_m$ , — but may put you closer to overfitting.

There are many ways how one can set up a sequence of learning algorithms that pick their respective optimal models from such a model class inclusion sequence. In most cases – such as in our two examples – this will just mean to admit larger models with more tuneable parameters.

#### 6.4.2 Using regularization for tuning modeling flexibility

The flexibility tuning mechanism explained in this subsection is simple, practical, and almost always used. It is called *model regularization*.

When one uses regularization to vary the modeling flexibility, one does not vary the model class  $\mathcal{H}$  at all. Instead, one varies the optimization objective (42) for minimizing the training error.

The basic geometric intuition behind modeling flexibility is that low-flexibility models should be "smooth", "more linear", "flatter", admitting only "soft curvatures" in fitting data; whereas high-flexibility models can yield "peaky", "rugged", "sharply twisted" curves (see again Figures 31, 32, 33).

When one uses model regularization, one fixes a single model structure and size with a fixed number of trainable parameters, that is, one fixes  $\mathcal{H}$ . Structure and size of the considered model should be rich and large enough to be able to overfit (!) the available training data. Thus one can be sure that the "right" model is contained in the search space  $\mathcal{H}$ . For instance, in our evergreen digit classification task one would altogether dismiss the dimension reduction through PCA (which has the same effect as using the maximum number m = 240 of PC components) and directly use the raw picture vectors (padded by a constant 1 component to enable affine linear maps) as argument vectors for a training a linear regression decision function. Or, in polynomial curve-fitting, one would fix a polynomial order that clearly is too large for the expected kind of true curve.

The models in  $\mathcal{H}$  are typically characterized by a set of trainable parameters. In our example of digit classification through linear regression from raw images, these trainable parameters are the elements of the regression weight matrix; in polynomial curve fitting these parameters are the monomial coefficients. Following the traditional notation in the machine learning literature we denote this collection of trainable parameters by  $\theta$ . This is a vector that has as many components as there are trainable parameters in the chosen kind of model. We assume that we have M tuneable parameters, that is  $\theta \in \mathbb{R}^M$ .

Such a high-flexibility model type would inevitably lead to overfitting when an "optimal" model would be learnt using the basic learning equation (42) which I repeat here for convenience:

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \ 1/N \ \sum_{i=1}^{N} L(h(x_i), y_i).$$

In order to dampen the exaggerated flexibility of this baseline learning algorithm, one adds a *regularization term* (also known as *penalty term*, or simply *regularizer*) to the loss function. A regularization term is a cost function
$R: \mathbb{R}^M \to \mathbb{R}^{\geq 0}$  which penalizes model parameters  $\theta$  that code models with a high degree of geometrical "wiggliness".

The learning algorithm then is constructed such that it solves, instead of (42), the regularized optimization problem

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \ 1/N \ \sum_{i=1}^{N} L(h(x_i), y_i) + \alpha^2 \ R(\theta_h).$$

$$(43)$$

where  $\theta_h$  is the collection of parameter values in the candidate models  $h \in \mathcal{H}$ . The nonnegative number  $\alpha^2$  determines how strongly we want to let the regularizer affect the final model — the larger  $\alpha^2$ , the more strongly we prefer "soft" models.

The design of a useful penalty term is up to your ingenuity. A good penalty term should, of course, assign high penalty values to parameter vectors  $\theta$  which represent "wiggly" models; but furthermore it should be easy to compute and blend well with the algorithm used for empirical risk minimization.

Two examples of such regularizers:

1. In the polynomial fit task from Section 6.2.1 one might consider for  $\mathcal{H}$  all 10th order polynomials, but penalize the "oscillations" seen in the right panel of Figure 31, that is, penalize such 10th order polynomials that exhibit strong oscillations. The degree of "oscillativity" can be measured, for instance, by the integral over the (square of the) second derivative of the polynomial p,

$$R(\theta) = R((w_0, \dots, w_{10})) = \int_0^1 \left(\frac{d^2 p(x)}{dx^2}\right)^2 dx.$$

Investing a little calculus (good exercise! not too difficult), it can be seen that this integral resolves to a quadratic form  $R(\theta) = \theta' C \theta$  where C is an  $11 \times 11$  sized positive semi-definite matrix. That format is more convenient to use than the original integral version.

2. A popular regularizer that often works well is just the squared sum of all model parameters,

$$R(\theta) = \sum_{w \in \theta} w^2.$$

This regularizer favors models with small absolute parameters, which often amounts to "geometrically soft" models. This regularizer is popular among other reasons because it supports simple algorithmic solutions for minimizing risk functions that contain it. It is called the  $L_2$ -norm regularizer because it measures the (squared)  $L_2$ -norm of the parameter vector  $\theta$ .

Computing a solution to the minimization task (43) means to find a set of parameters which simultaneously minimizes the original risk and the penalty term. The factor  $\alpha^2$  in (43) controls how strongly one wishes the regularization to "soften" the solution. Increasing  $\alpha$  means downregulating the model flexibility. For  $\alpha^2 = 0$  one returns to the original un-regularized empirical risk (which would likely mean overfitting). For  $\alpha^2 \to \infty$  the regularization term entirely dominates the model optimization and one gets a model which does not care anymore about the training data but instead only is tuned to have minimal regularization penalty. In case of the  $L_2$  norm regularizer this means that all model parameters are zero – the ultimate wiggle-free model; one should indeed say the model is dead.

When regularization is used to steer the degree of model flexibility, the x-axis in Figure 35 would be labelled by  $\alpha^2$  (highest  $\alpha^2$  on the left, lowest at the right end of the x-axis).

Using regularizers to vary model flexibility is often computationally more convenient than using different model sizes, because one does not have to tamper with differently structured models. One selects a model type with a very large (unregularized) flexibility, which typically means to select a big model with many parameters (maybe hundreds of thousands). Then all one has to do (and one *must* do it – it is a *most crucial* part of any professionally done machine learning project) is to find the optimal degree of flexibility which minimizes the risk. At this moment we don't know how to do that – the secret will be revealed later in this section.

I introduced the word "regularization" here for the specific flexibility-tuning method of adding a penalty term to the loss function. The same word is however often used in a generalized fashion to denote *any* method used for steering model flexibility.

#### 6.4.3 Ridge regression

Let us briefly take a fresh look at linear regression, now in the light of general supervised learning and regularization. Linear regression should always be used in conjunction with regularization. Because this is such a helpful thing to know, I devote this separate subsection to this trick. Recall from Section 3.1 that the learning task solved by linear regression is to find

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^{N} (\mathbf{w} \, \mathbf{x}_i - y_i)^2, \tag{44}$$

where  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  is a set of training data with  $\mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathbb{R}$ . Like any other supervised learning algorithm, linear regression may lead to overfitting solutions  $\mathbf{w}_{opt}$ . It is *always* advisable to control the flexibility of linear regression with an  $L_2$  norm regularizer, that is, instead of solving (44) go for

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} (\mathbf{w} \, \mathbf{x}_i - y_i)^2 + \alpha^2 \, \|\mathbf{w}\|^2$$
(45)

and find the best regularization coefficient  $\alpha^2$ . The optimization problem (45) admits a closed-form solution, namely the *ridge regression* formula that we have

already met in Equation 21. Rewriting it a little to make it match with the current general scenario, here it is again:

$$\mathbf{w}_{\text{opt}}' = \left(\frac{1}{N} X X' + \alpha^2 I_{n \times n}\right)^{-1} \frac{1}{N} X Y = \left(X X' + \alpha^2 I_{n \times n}\right)^{-1} X Y,$$
(46)

where  $X = [x_1, ..., x_N]$  and  $Y = (y_1, ..., y_N)'$ .

In Section 3.1 I motivated to use the ridge regression formula because it warrants numerical stability. Now we see that a more fundamental reason to prefer ridge regression over the basic kind of regression (44) is that it implements  $L_2$  norm regularization. The usefulness of ridge regression as an allround simple baseline tool for supervised learning tasks can hardly be overrated.

### 6.4.4 Tuning model flexibility through adding noise

Another way to tune model flexibility is to add noise. Noise can be added at different places. Here I discuss three scenarios.

- Adding noise to the training input data. If we have a supervised training dataset  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  with vector patterns  $\mathbf{x}_i$ , one can enlarge the training dataset by adding more patterns obtained from the original patterns  $(\mathbf{x}_i, y_i)_{i=1,...,N}$  by adding some noise vectors to each input pattern  $\mathbf{x}_i$ : for each  $\mathbf{x}_i$ , add l variants  $\mathbf{x}_i + \nu_{i1}, \ldots, \mathbf{x}_i + \nu_{il}$  of this pattern to the training data, where the  $\nu_{ij}$  are i.i.d. random vectors (for instance, uniform or Gaussian noise). This increases the number of training patterns from N to (l+1)N. The more such noisy variants are added and the stronger the noise, the more difficult will it be for the learning algorithm to fit all data points, and the smoother the optimal solution becomes that is, the more one steers to the left (underfitting) side of Figure 35. Adding noisy (or otherwise artificially varied) examples to the training dataset is a very common strategy, called *data augmentation*.
- Adding noise while the optimization algorithm runs. If the optimization algorithm used for minimizing the empirical risk is iterative, one can "noisify" it by jittering the intermediate results with additive noise. We have not seen an iterative algorithm in this course so far, therefore I cannot demonstrate this by example here. The "dropout regularization" trick, which is widely used in deep learning, is of this kind. The effect is that the stronger the algorithm is noisified, the stronger the regularization, that is the further one steers to the left end of Figure 35.
- Use stochastic ensemble learning. We have seen this strategy in the presentation of random forests. A stochastic learning algorithm is repeatedly executed with different random seeds (called  $\Theta$  in Section 2.7). The stronger the randomness of the stochastic learning algorithm, and the more members are included in the ensemble, the stronger the regularization effect.

### 6.4.5 Interim summary: what we have when we can tune model flexibility

We have seen that there are many ways to set up a learning algorithm such that it has a specific degree of 'flexibility' m. Once a particular such degree of flexibility m is set (for instance by fixing model class size, regularization coefficient  $\alpha^2$ , or inserting a specific amount of noise in the data or algorithm), the learning algorithm is run on the training data and it finds the mini (or should find) the minimal-training-error model

$$h_{\text{opt }m} = \underset{h \in \mathcal{H}_m}{\operatorname{argmin}} \quad 1/N \sum_{i=1}^N L(h(x_i), y_i).$$

$$(47)$$

But for every setting of m, we get a different  $h_{\text{opt }m}$ . Which one is the best, that is, which one has the smallest expected test error (risk)  $R(h_{\text{opt }m})$ ? We have to solve the overarching optimization problem

$$m_{\text{opt}} = \underset{m}{\operatorname{argmin}} R(h_{\text{opt }m})?$$
 (48)

The visual intuition that you should have is the one from Figure 35 - you have to find the sweet spot on the *m*-axis where the test error curve has its minimum. The next subsection explains how this spot is practically determined by crossvalidation. Here I want to spell out the big take-home for this entire section in yet another wording:

A (supervised) machine learning project typically involves two nested optimization procedures:

- 1. the inner loop of minimizing the empirical risk / training error for a given degree of flexibility m,
- 2. the outer loop of optimizing for best (estimated) risk / testing error across a search range of degrees m of flexibility.

I mention that in professional modeling one usually applies several flexibilitychanging mechanisms at the same time. For instance, in deep learning one typically combines  $L_2$ -norm regularization (called 'weight decay' in that field) with several sorts of adding noise (the most famous among them known as the *dropout* trick, but also setting minibatch sizes and learning rates, as well as *data augmentation* can be regarded as setting different noiselevels). If one has several flexibilityshaping control knobs, the single-line *m*-axis in our Figure 35 because a multidimensional search space for modeling *hyperparameters*. While in the single-line, one-dimensional flexibility tuning case, one often can do a comprehensive sweep through the relevant range of *m*, this quickly becomes infeasible when there are many tuneable hyperparameters m. The outer search loop then becomes a steep computational and algorithmical challenge in itself. This amounts to develop search methods for minima in few-dimensional search spaces (as many dimensions as one has hyperparameters) that need only very few trials (evaluation of the objective function at candidate solutions, here: at certain combinations of hyperparameter values) to come close to a minimum — very few, because each such trial means a very expensive training of a model in the inner loop. The current best practice appears to use for this outer loop a search method called *Bayesian optimization* with *Gaussian processes*. This lies beyond the scope of our course. If you are interested, check out https://en.wikipedia.org/wiki/Bayesian\_optimization.

# 6.5 Finding the right modeling flexibility by cross-validation

Statistical learning theory has come up with a few analytical methods to approximately solve (48). But these methods are based on additional assumptions, which are neither easy to verify nor always granted. By far the most widely used method to determine an (almost) optimal model flexibility (that is, determine the position of the green line in Figure 35) is a rather simple scheme called *cross-validation*. Cross-validation is a generic method which does not need analytical insight into the particulars of the learning task at hand. Its main disadvantage is that it may be computationally expensive.

Here is the basic idea of cross-validation.

In order to determine whether a given model is under- or overfitting, one would need to run it on test data that are "new" and not contained in the training data. This would allow one to get a hold on the red curve in Figure 35.

However, at training time only the training data are available.

The idea of cross-validation is to artificially split the training data set  $D = (x_i, y_i)_{i=1,...,N}$  into two subsets  $T = (x_i, y_i)_{i \in I}$  and  $V = (x'_i, y'_i)_{i \in I'}$ . These two subsets are then pretended to be a "training" and a "testing" dataset. In the context of cross-validation, the second set is called a *validation* set.

A bit more formally, let the flexibility axis in Figure 35 be parametrized by m, where small m means "strong regularization", that is, "go left" on the flexibility axis. Let the range of m be  $m = 1, \ldots, M$ .

For each setting of regularization strength m, the data in T is used to train an optimal model  $h_{\text{opt }m}$ . The test generalization performance on "new" data is then tested on the validation set, for each  $m = 1, \ldots, M$ . It is determined which model  $h_{\text{opt }m}$  performs best on the validation data. Its regularization strength m is then taken to be the sought solution  $m_{\text{opt}}$  to (48). After this screening of regularization strengths for the best test performance, a final model is then trained, with that optimal regularization strength, on the original complete training data set D.

This whole procedure is called cross-validation. Notice that nothing has been said so far about how to split D into T and V. This is not a trivial question: how should D be best partitioned?

A clever way to answer this question is to split D into K subsets  $D_j$  of equal size (j = 1, ..., K). Then, for every regularization strength m, run K model learning procedures, where in the j-th run the subset  $D_j$  is withheld as a validation set, and the remaining K - 1 sets joined together make for a training set. After these K runs, average the validation errors in order to find a good estimate for the expected test error at this regularization strength m. After this K-fold cross-validation has been done for all regularization strengths  $m = 1, \ldots, M$ , pick that  $m_{opt}$  which had the lowest validation error. Here is the procedure in detail (where I use class inclusion as the method for navigating through the flexibilities m):

**Given:** A set  $(x_i, y_i)_{i=1,\dots,N}$  of training data, and a loss function L.

Also given: Some method which allows one to steer the model flexibility along a regularization strength parameter m = 1, ..., M. The weakest regularization should be weak enough to allow overfitting.

**Step 1.** Split the training data into K disjoint subsets  $D_j = (x_i, y_i)_{i \in I_j}$  of roughly equal size N' = N/K.

Step 2. Repeat for  $m = 1, \ldots, M$ :

**Step 2.1** Repeat for j = 1, ..., K:

**Step 2.2.1** Designate  $D_j$  as validation set  $V_j$  and the union of the other  $D_{j'}$  as training set  $T_j$ .

**Step 2.2.1** Compute the model with minimal training error on  $T_i$ 

$$h_{\text{opt}\,m\,j} = \underset{h \in \mathcal{H}_m}{\operatorname{argmin}} 1/|T_j| \sum_{(x_i, y_i) \in T_j} L(h(x_i), y_i)$$

where  $\mathcal{H}_m$  is the model search space for the regularization strength m.

**Step 2.2.2** Test  $h_{opt mj}$  on the current validation set  $V_j$  by computing the validation risk

$$R_{mj}^{\text{val}} = 1/|V_j| \sum_{(x_i, y_i) \in V_j} L(h_{\text{opt}\,m\,j}(x_i), y_i).$$

**Step 2.2** Average the K validation risks  $R_{mj}^{\text{val}}$  obtained from the "folds" carried out for this m, obtaining

$$R_m^{\rm val} = 1/K \sum_{j=1,\dots,K} R_{m\,j}^{\rm val}$$

Step 3. Find the optimal class by looking for that m which minimizes the averaged validation risk:

$$m_{\text{opt}} = \underset{m}{\operatorname{argmin}} R_m^{\text{val}}.$$

**Step 4.** Compute  $h_{m_{opt}}$  using the complete original training data set:

$$h_{m_{\text{opt}}} = \underset{h \in \mathcal{H}_{m_{\text{opt}}}}{\operatorname{argmin}} 1/N \sum_{i=1,\dots,N} L(h(x_i), y_i).$$

This procedure contains two nested loops and looks expensive. For economy, one starts with the low-end m and increases it stepwise, assessing the generalization quality through cross-validation for each regularization strength m, until the

validation risk starts to rise. The strength  $m_{\text{opt}}$  reached at that point is likely to be about the right one.

The best assessment of the optimal class is achieved when the original training data set is split into singleton subsets – that is, each  $D_j$  contains just a single training example. This is called *leave-one-out cross-validation*. It looks like a horribly expensive procedure, but yet it may be advisable when one has only a small training data set, which incurs a particularly large danger of ending up with poorly generalizing models when a wrong model flexibility was used.

K-fold cross validation is widely used – it is a factual standard procedure in supervised learning tasks when the computational cost of learning a model is affordable.

### 6.6 Why it is called the bias-variance dilemma

We have seen that a careful adjustment of the flexibility of a supervised learning algorithm is needed to find the sweet spot between underfitting and overfitting. A more educated way to express this condition is to speak of the *bias-variance tradeoff*, also known as *bias-variance dilemma*. In this subsection I want to unravel the root cause of the under/overfitting phenomenon in a little more mathematical detail. We will find that it can be explained in terms of a bias and a variance term in the expected error of estimated models.

Again I start from a training data set  $(x_i, y_i)_{i=1,...,N}$  drawn from a joint distribution  $P_{X,Y}$ , with  $x_i \in \mathbb{R}^n, y_i \in \mathbb{R}$ . Based on these training data I consider the learning task to find a decision function  $h : \mathbb{R}^n \to \mathbb{R}$  which has a low quadratic risk

$$R(h) = E_{X,Y}[(h(X) - Y)^2],$$
(49)

where the notation  $E_{X,Y}$  indicates that the expectation is taken with respect to the joint distribution of X and Y. We assume that we are using some fixed learning algorithm  $\mathcal{A}$  which, if it is given a training sample  $(x_i, y_i)_{i=1,...,N}$ , estimates a model  $\hat{h}$ . The learning algorithm  $\mathcal{A}$  can be anything, good or bad, clever or stupid, overfitting or underfitting; it may be close to perfect or just be always returning the same model without taking the training sample into account – we don't make any assumptions about it.

The next consideration leads us to the heart of the matter, but it is not trivial. In mathematical terms, the learning algorithm  $\mathcal{A}$  is just a *function* which takes a training sample  $(x_i, y_i)_{i=1,...,N}$  as input and returns a model  $\hat{h}$ . Importantly, if we would run  $\mathcal{A}$  repeatedly, but using freshly sampled training data  $(x_i, y_i)_{i=1,...,N}$ in each run, then the returned models  $\hat{h}$  would be varying from trial to trial – because the input samples  $(x_i, y_i)_{i=1,...,N}$  are different in different trials. Applying these varying  $\hat{h}$  to some fixed pattern  $x \in \mathbb{R}^n$ , the resulting values  $\hat{h}(x)$  would show a random behavior too. The distribution of these variable values  $\hat{h}(x)$  is a distribution over  $\mathbb{R}$ . This distribution is determined by the distribution  $P_{X,Y}$  and the chosen learning algorithm  $\mathcal{A}$  and the training sample size. A good statistician could give us a formal derivation of the distribution of  $\hat{h}(x)$  from  $P_{X,Y}$  and knowledge of  $\mathcal{A}$ , but we don't need that for our purposes here. The only insight that we need to take home at this point is that for a fixed x,  $\hat{h}(x)$  is a random variable whose value is determined by the drawn training sample  $(x_i, y_i)_{i=1,...,N}$ , and which has an expectation which we write as  $E_{\text{retrain}}[\hat{h}(x)]$  to indicate that the expectation is taken over all possible training runs with freshly drawn training data.

Understanding this point is the key to understanding the inner nature of under/overfitting.

If you feel that you have made friends with this  $E_{\text{retrain}}[\hat{h}(x)]$  object, we can proceed. The rest is easy compared to this first conceptual clarification.

Without proof I note the following, intuitively plausible fact. Among *all* decision functions (from any candidate space  $\mathcal{H}$ ), the quadratic risk (49) is minimized by the function

$$\Delta(x) = E_{Y|X=x}[Y],\tag{50}$$

that is, by the expectation of Y given x. This function  $\Delta : \mathbb{R}^n \to \mathbb{R}, x \mapsto E[Y|X = x]$  is the gold standard for minimizing the quadratic risk; no learning algorithm can give a better result than this. Unfortunately, of course,  $\Delta$  remains unknown because the underlying true distribution  $P_{X,Y}$  cannot be exactly known.

Now fix some x and ask by how much h(x) deviates, on average and in the squared error sense, from the optimal value  $\Delta(x)$ . This expected squared error is

$$E_{\mathsf{retrain}}[(\hat{h}(x) - \Delta(x))^2]$$

We can learn more about this error if we re-write  $(\hat{h}(x) - \Delta(x))^2$  as follows:

$$(\hat{h}(x) - \Delta(x))^2 = (\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)] + E_{\text{retrain}}[\hat{h}(x)] - \Delta(x))^2 = (\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)])^2 + (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x))^2 + 2(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)]) (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)).$$
(51)

Now observe that

$$E_{\text{retrain}}\left[\left(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)]\right) \left(E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)\right)\right] = 0, \tag{52}$$

because the second factor  $(E_{\mathsf{retrain}}[\hat{h}(x)] - \Delta(x))$  is a constant, hence

$$\begin{split} E_{\mathsf{retrain}} & \left[ (\hat{h}(x) - E_{\mathsf{retrain}}[\hat{h}(x)]) \ (E_{\mathsf{retrain}}[\hat{h}(x)] - \Delta(x)) \right] = \\ & = E \left[ \hat{h}(x) - E_{\mathsf{retrain}}[\hat{h}(x)] \right] \ (E_{\mathsf{retrain}}[\hat{h}(x)] - \Delta(x)), \end{split}$$

and

$$\begin{aligned} E_{\mathsf{retrain}} \left[ \hat{h}(x) - E_{\mathsf{retrain}} [\hat{h}(x)] \right] &= E_{\mathsf{retrain}} [\hat{h}(x)] - E_{\mathsf{retrain}} [E_{\mathsf{retrain}} [\hat{h}(x)]] \\ &= E_{\mathsf{retrain}} [\hat{h}(x)] - E_{\mathsf{retrain}} [\hat{h}(x)] \\ &= 0. \end{aligned}$$

Inserting (52) into (51) and taking the expectation on both sides of (51) of finally gives us

$$E_{\text{retrain}}[(\hat{h}(x) - \Delta(x))^{2}] = \underbrace{\left(E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)\right)^{2}}_{(\text{squared}) \text{ bias}} + \underbrace{E_{\text{retrain}}\left[\left(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)]\right)^{2}\right]}_{\text{variance}}.$$
 (53)

The two components of this error are conventionally named the *bias* and the *variance* contribution to the expected squared mismatch  $E_{\mathsf{retrain}}[(\hat{h}(x) - \Delta(x))^2]$  between the learnt-model decision values  $\hat{h}(x)$  and the optimal decision value  $\Delta(x)$ . The bias measures how strongly the average learning result deviates from the optimal value; thus it indicates a systematic error component. The variance measures how strongly the learning results  $\hat{h}(x)$  vary around their expected value  $E_{\mathsf{retrain}}[\hat{h}(x)]$ ; this is an indication of how strongly the particular training data sets induce variations on the learning result.



Figure 36: Illustrating the bias and variance contributions to expected modeling error. Some x from the input space is fixed, and the model outputs here are twodimensional vectors  $\hat{h}(x) = (y_1, y_2)'$ . The black dot gives  $\Delta(x)$ . Each color group of dots shows outputs of models trained with the same learning algorithm but with freshly sampled training sets. The green dots show model outputs  $\hat{h}(x)$  of models trained with an algorithm  $\mathcal{A}_{green}$  that has high variance but zero bias (that is, the expected average value of the green dots would be centered on the black dot). The red dots show outcomes obtained from a learning algorithm  $\mathcal{A}_{red}$  with smaller variance and again, zero bias. The blue dots show how outcomes from a learning algorithm with very small variance but a nonzero bias would spread.

When the modeling flexibility is too low (underfitting), the bias term dominates the expected modeling error. In the most extreme case, the modeling flexibility is zero – the learning algorithm always returns models that 'compute' the same value for  $\hat{h}(x)$ , independent of the training set: zero variance, but likely a high bias (unless the designer of this 'learning' algorithm knew the right model beforehand). When the modeling flexibility is too high (overfitting), the variance term is the main source of mismatch, because the learnt models essentially just replicate the training data — which means, it replicates the variance across different training samples. This is why the underfitting versus overfitting challenge is also called the bias-variance tradeoff (or dilemma). Figure 36 illustrates the look-and-feel of different combinations of bias and variance.

# 7 Representing and learning distributions

Almost all machine learning tasks are based on training data that have some random component. Having completely noise-free data from deterministic sources observed with high-precision measurements is a rare exception. Thus, machine learning algorithms are almost all designed to cope with stochastic data. Their ultimate functionality (classification, prediction, control, ...) will be served well or poorly to the extent that the probability distribution of the training data has been properly accounted for. We have seen this in the previous section. If one (wrongly) believes that there is little randomness in the training data, one will take the given training points as almost correct – and end up with an overfitted model. Conversely, if one (wrongly) thinks the training data are almost completely "white noise randomness", the learnt model will under-exploit the information in the training data and underfit. Altogether it is fair to say that machine learning is the art of probability distribution modeling (plus the subsequent use of that learnt distribution for classification etc.)

In many machine learning algorithms, the distribution model remains implicit – there is no place or data structure within the learnt model which explicitly models the data distribution. In some algorithms however, and for some tasks, the probability distribution of the training data is explicitly modeled. We will encounter some of these algorithms later in this course (hidden Markov models, Bayesian networks, Boltzmann machines are of that kind). It is part of the standard knowledge of a machine learning professional to know some ways to represent probability distributions and to estimate these representations from data.

# 7.1 Optimal classification

The grand role which is played by modeling a data distribution accurately can be demonstrated very nicely in classification learning. We have met this basic task several times now – and in your professional future you will meet it again, dozens of times if you will seriously work in machine learning. We consider again a scenario where the input patterns are vectors  $\mathbf{x} \in \mathbf{P} \subseteq \mathbb{R}^n$ , which belong to one of k classes which we represent by their class labels  $C = \{c_1, \ldots, c_k\}$ . The set  $\mathbf{P}$  is the "possible pattern space", usually a bounded subset of  $\mathbb{R}^n$  (for instance, when the patterns  $\mathbf{x}$  are photographic pixel vectors and the pixel color values are normalized to a range between 0 and 1,  $\mathbf{P}$  would be the *n*-dimensional unit hypercube). We furthermore introduce a random variable (RV) X for the patterns and a RV Y for the class labels.

Many different decision functions  $h : \mathbf{P} \to C$  exist, some of which can be learnt from training data using some known learning algorithm. We will not deal with the learning problem in this subsection, but ask (and answer!) the fundamental question:

Among all possible decision functions  $h : \mathbf{P} \to C$ , which one has the lowest risk in the sense of giving the lowest possible rate of misclassifications on test data? Or, expressing the same question in terms of a loss function, which decision function minimizes the risk connected to the counting loss

$$L(y,c) = \begin{cases} 1 & \text{if } y \neq c, \\ 0 & \text{if } y = c \end{cases}$$

It turns out that the minimal-risk decision function is in fact well-defined and unique, and it can (and must) be expressed in terms of the distribution of our data-generating RVs X and Y.

Our starting point is the true joint distribution  $P_{X,Y}$  of patterns and labels. This joint distribution is given by all the probabilities of the kind

$$P(X \in A, Y = c), \tag{54}$$

where A is some subvolume of **P** and  $c \in \mathbf{C}$ . The subvolumes A can be ndimensional hypercubes within **P**, but they also can be arbitrarily shaped "volume bodies", for instance balls or donuts or whatever. Note that the probabilities  $P(X \in A, Y = c)$  are numbers between 0 and 1, while the distribution  $P_{X,Y}$  is the *function* which assigns to every choice of  $A \subseteq \mathbf{P}, c \in C$  the number  $P(X \in$ A, Y = c) (probability theory gives us a rigorous formal way to define and handle this strange object,  $P_{X,Y}$  — it is explained with loving care in my lecture notes for "Principles of Statistical Modeling").

The joint distribution  $P_{X,Y}$  is the "ground truth" – it is the real-world statistical distribution of pattern-label pairs of the kind we are interested in. In our Digits example, it would be the distribution of pairs made of (i) a handwritten digit and (ii) a human-expert provided class label. Test digit images and their class labels would be randomly "drawn" from this distribution.

A decision function  $h : \mathbf{P} \to \{c_1, \dots, c_k\}$  partitions the pattern space  $\mathbf{P}$  into k disjoint decision regions  $R_1, \dots, R_k$  by

$$R_i = \{ \mathbf{x} \in \mathbf{P} \,|\, h(\mathbf{x}) = c_i \}. \tag{55}$$

A test pattern  $\mathbf{x}^{\text{test}}$  is classified by h as class i if and only if it falls into the decision region  $R_i$ .

Now we are prepared to analyze and answer our ambitious question, namely which decision functions yield the lowest possible rate of misclassifications. Since two decision functions yield identical classifications if and only if their decision regions are the same, we will focus our attention on these regions and reformulate our question: which decision regions yield the lowest rate of misclassifications, or expressed in its mirror version, which decision regions give the highest probability of correct classifications?

Let  $f_i$  be the pdf for the conditional distribution  $P_{X|Y=c_i}$ . It is called the *class-conditional* distribution.

The probability to obtain a correct classification for a random test pattern, when the decision regions are  $R_i$ , is equal to  $\sum_{i=1}^{k} P(X \in R_i, Y = c_i)$ . Rewriting this expression using the pdfs of the class conditional distributions gives

$$\sum_{i=1}^{k} P(X \in R_i, Y = c_i) =$$

$$= \sum_{i=1}^{k} P(X \in R_i | Y = c_i) P(Y = c_i)$$

$$= \sum_{i=1}^{k} P(Y = c_i) \int_{R_i} f_i(\mathbf{x}) d\mathbf{x}$$

$$= \sum_{i=1}^{k} \int_{R_i} P(Y = c_i) f_i(\mathbf{x}) d\mathbf{x}.$$
(56)

Note that the integral is taken over a region that possibly has curved boundaries, and the integration variable  $\mathbf{x}$  is a vector. The boundaries between the decision regions are called *decision boundaries*. For patterns  $\mathbf{x}$  that lie exactly on such boundaries, two or more classifications are equally probable. For instance, the digit pattern shown in the last but third column in the second row in Figure 15 would likely be classified by humans as a "1" or "4" class pattern with roughly the same probability; this pattern would lie close to a decision boundary.

The expression (56) obviously becomes maximal if the decision regions are given by

$$R_i = \{ \mathbf{x} \in \mathbf{P} \,|\, i = \operatorname*{argmax}_j P(Y = c_j) \,f_j(\mathbf{x}) \}.$$
(57)

Thus we have found the decision function which is optimal in the sense that it maximizes the probability of correct classifications: namely

$$h_{\text{opt}} : \mathbf{P} \to C, \ \mathbf{x} \mapsto c_{\underset{j}{\operatorname{argmax}}P(Y=c_j) f_j(\mathbf{x})}.$$
 (58)

A learning algorithm that finds the optimal decision function (or some function approximating it) must learn (implicitly or explicitly) estimates of the class-conditional distributions  $P_{X|Y=c_i}$  and the class probabilities  $P(Y=c_i)$ .

The class probabilities are also called the class *priors*. Figure 37 visualizes optimal decision regions and decision boundaries. In higher dimensions, the geometric shapes of decision regions can become exceedingly complex, fragmented and "folded into one another" — disentangling them during a learning process is one of the eternal challenges of ML.

## 7.2 Representing and learning distributions

The optimal classifier described in the previous subsection is optimal because it is shaped along the true distribution of pattern and class label random variables.



Figure 37: Optimal decision regions  $R_i$ . A case with a one-dimensional pattern space **P** and k = 3 classes is shown. Broken lines indicate decision boundaries. Decision regions need not be connected!

Quite generally, *any* machine learning task can be solved "optimally" (in terms of minimizing some risk) only if the solution takes the true distribution of all task-relevant RVs into account. As I mentioned before, many learning algorithms estimate a model of the underlying data distribution only implicitly. But some ML algorithms generate explicit models of probability distributions, and in the wider fields of statistical modeling, explicit models of probability distributions are often the final modeling target.

Representing a probability distribution in mathematical formalism or by an algorithm is not always easy. Real-world probability distributions can be utterly complex and high-dimensional objects which one cannot just "write down" in a formula. Over the last 60 or so years, ML and statistics research has developed a wide range of formalisms and algorithms for representing and estimating ("learning") probability distributions. A machine learning professional should know about some basic kinds of such formalisms and algorithms. In this section I present a choice, ranging from elementary to supremely complex, powerful — and computationally costly.

#### 7.2.1 Some classical probability distributions

In this section we describe a few distributions which arise commonly in application scenarios. They have standard names and one should just know them, and under which conditions they arise. Most of you will know then already anyway from an introductory course in statistics.

We start with distributions that are defined over discrete sample spaces  $S = \{s_1, \ldots, s_k\}$  (finite sample space) or  $S = \{s_1, s_2, s_3 \ldots\}$  (countably infinite sample space). These distributions can all be represented by their *probability mass function* (pmf):

**Definition 7.1** Given a discrete sample space S (finite or countably infinite), a probability mass function on S is a function  $p: S \to [0,1]$  whose total mass is 1, that is, which satisfies  $\sum_{s \in S} p(s) = 1$ .

**Bernoulli distribution.** The Bernoulli distribution arises when one deals with observations that have only two possible outcomes, like tail – head, female – male, pass – fail, 0 – 1. We have a two-element sample space  $S = \{s_1, s_2\}$ , on which a Bernoulli distribution is defined by its pmf, which has its own standard terminology:

**Definition 7.2** The distribution of a random variable which takes values in a binary value space  $S = \{s_1, s_2\}$  with probability mass function

$$p(s_i) = \begin{cases} 1 - q & \text{for } i = 1\\ q & \text{for } i = 2 \end{cases}$$

is called a Bernoulli distribution with success parameter q, where  $0 \le q \le 1$ .

A Bernoulli distribution is thus specified by a single parameter, q. Learning / estimation: Given a set of training data  $\{x_1, \ldots, x_N\}$  where  $x_i \in \{s_1, s_2\}$ , the parameter q can be estimated by  $\hat{q} = (1/N) |\{i \mid x_i = s_2\}|$ .

**Binomial distribution.** The binomial distribution describes the counts of successes if a binary-outcome "Bernoulli experiment" is repeated N times. For a simple example, consider a gamble where you toss a coin N times, and every time the head comes up, you earn a Dollar (not Euro; gambling is done in Las Vegas). What is the distribution of Dollar earnings from such N-repetition games, if the coin comes up with head (=  $s_2$ ; outcome tail is  $s_1$ ) with a success probability of q? Clearly the range of possible earnings goes from 0 to N Dollars. These earnings are distributed according to the *binomial distribution*:

**Definition 7.3** Let N be the number of trials of independent Bernoulli experiments with success probability q in each trial. The distribution of the number of successes is called the binomial distribution with parameters N and q and its pmf is given by

$$p(s) = \binom{N}{s} q^s (1-q)^{N-s} = \frac{N!}{s! (N-s)!} q^s (1-q)^{N-s}, \quad s = 0, 1, 2, ..., N.$$

We write Bi(N,q) to denote the binomial distribution with parameters N and q.

The factor  $\binom{N}{s}$  is called *binomial coefficient*. Figure 38 shows some binomial pmf's. A note on notation: it is customary to write

$$X \sim Bi(10, 0.25)$$



Figure 38: The pmf's of Bi(20, 0.1) (blue), Bi(20, 0.5) (red), and Bi(20, 0.9) (green). Figure taken from www.boost.org.

as a shorthand for the statement "X is distributed according to Bi(10, 0.25)". Learning / estimation: Depends on the format of available training data. In most cases the data will be of the same form as in the Bernoulli distribution, which gives rise to an estimate  $\hat{q}$  of the success parameter. The number N of repetitions will usually be given by the context in which the modeling task arises and need not be estimated.

**Poisson distribution.** This distribution is defined for  $S = \{0, 1, 2, ...\}$ . p(s) is the probability that a particular kind of event occurs k times within a given time interval. Examples (except the last one taken from https://en.wikipedia.org/wiki/Poisson\_distribution): p(k) might be the probability that

- k meteorites impact on the earth within 100 years,
- a call center receives k calls in an hour,
- a block of uranium emits k alpha particles in a second,
- k patients arrive in an emergency ward between 10 and 11 pm,
- a piece of brain tissue emits k neural spikes within a second.

Similarly, instead of referring to a time interval, k may count spatially or otherwise circumscribed events, for instance

- the number of dust particles found in a milliliter of air,
- the number of diamonds found in a ton of ore.

The Poisson distribution is also called the "distribution of rare events", which I find somewhat a misnomer, because many of the examples above describe anything but rare events — for instance the number of spikes emitted from a blob of brain tissue within a second could be in the order of millions. The important two conditions that let a piece of reality create Poisson-distributed events are (i) that the individual events are created in total independence from each other — when in a block of uranium one atom decays and a little later another atom, these two events have no causal influence on each other whatsover; and (ii) that there is a very large "reservoir" of event-creating objects or mechanisms. For instance, in the meteroite bombardment example, the event-generating resource would be the entire universe; in the call center example it would be the population of people in a country served by the call center. In mathematical rigor, the "event reservoir" should be infinitely sized, but in the real world of course no infinite sources of events exist, thus the Poisson distribution is an approximate model of such "rare event" generating systems.

The expected number of events E[X] is called the *rate* of the Poisson distribution, and is commonly denoted by  $\lambda$ . The pmf of a Poisson distribution with rate  $\lambda$  is given by

$$p(k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$
(59)

Figure 39 depicts the pmf's for three different rates.



Figure 39: The pmf of the Poisson distribution for various values of the parameter  $\lambda$ . The connecting lines better the dots are drawn only for better visual appearance. Note that the *k*-axis is infinite to the right but here capped at 20 because all shown curves will be so close to zero beyond k = 20 that plotting them will not show the difference (image source: https://commons.wikimedia.org/wiki/File:Poisson\_pmf.svg).

**Learning** / estimation: For concreteness consider the call center example. Training data would be N 1-hour protocols of incoming calls, where  $n_i$  (i =  $1, \ldots, N$  is the number of calls received in the respective hour. Then  $\lambda$  is estimated by  $\hat{\lambda} = (1/N) \sum_{i} n_{i}$ .

We continue with a few distributions on continuous sample spaces ( $\mathbb{R}^n$  or subsets thereof). These distributions are most often characterized by their *probability density functions* (pdf's).

**Definition 7.4** Let X be a RV which takes values in  $S \subseteq \mathbb{R}^n$ . A pdf for the distribution of X is a function  $f: S \to \mathbb{R}^{\geq 0}$  which satisfies the following condition: For every subvolume  $A \subseteq S$  of S, the probability  $P(X \in A)$  that X gives a value in A is equal to

$$P(X \in A) = \int_{A} f(\mathbf{x}) d\mathbf{x}.$$
 (60)

**Point distributions.** There exist continuous-valued distributions for which a description through a pdf is impossible. One kind of such "non-pdf-able" continuous distributions occurs quite often: *point distributions*. Here, the probability mass is concentrated in a few (finitely many or countably infinite many) points in  $\mathbb{R}^n$ . The simplest point distribution is defined on the real line  $S = \mathbb{R}$  and has the defining property that for any subset  $A \subseteq \mathbb{R}$  it holds that

$$P(X \in A) = \begin{cases} 1 & \text{if } 0 \in A \\ 0 & \text{if } 0 \notin A. \end{cases}$$

This is the formal way to express the fact that "X always returns an observation value 0".

There are several ways to write down such a point distribution in mathematical formalism. In the machine learning literature (and throughout the natural sciences, especially physics), the above point distribution would be represented by a weird kind of pdf-like function, called the *Dirac delta function*  $\delta$  (the Wikipedia article https://en.wikipedia.org/wiki/Dirac\_delta\_function is recommendable if you want to understand this function better). The Dirac delta function is used inside an integral just like a normal pdf is used in (60). Thus, for a subset  $A \subseteq \mathbb{R}$  one has

$$P(X \in A) = \int_A \delta(x) \, dx.$$

The Dirac delta is also used in  $\mathbb{R}^n$ , where likewise it is a "pdf" which describes a probability distribution concentrated in a single point, the origin.

If one wants to have a multi-point distribution one can combine Dirac deltas. For example, if you want to create a probability measure on the real line that places a probability of 1/2 on the point 1.0, and probabilities 1/4 each on the points 2.0 and 3.0, you can do this by a linear combination of shifted Dirac deltas:

$$P(X \in A) = \int_{A} 1/2 \,\delta(x-1) + 1/4 \,\delta(x-2) + 1/4 \,\delta(x-3) \,dx$$

Point distributions occur all over the place in Bayesian machine learning, where they are needed to express and compute certain "hyperdistributions". We will meet them later in this course.

We now take a look at a choice of common continuous distributions that can be characterized by pdfs.

The uniform distribution. We don't need to make a big fuzz about this. If  $I = [a_1, b_1] \times \ldots \times [a_n, b_n]$  is a *n*-dimensional interval in  $\mathbb{R}^n$ , the uniform distribution on *I* is given by the pdf

$$p(x) = \begin{cases} \frac{1}{(b_1 - a_1) \cdot \dots \cdot (b_n - a_n)} & \text{if } x \in I \\ 0 & \text{if } x \notin I. \end{cases}$$
(61)

**Learning / estimation:** This distribution is not normally "learnt". There is also nothing about it that could be learnt: the uniform distribution has no "shape" that would be specified by learnable parameters.

The exponential distribution. This distribution is defined for  $S = [0, \infty)$  and could be paraphrased as "the distribution of waiting times until the next of these rare things happens". Consider any of the kinds of temporal events listed for the Poisson distribution, for instance the event "meteorite hits earth". The exponential distribution characterizes how long you have to wait for the next impact, given that one impact has just happened. Like in the Poisson distribution, such random event processes have an average *rate* events / unit reference time interval. For instance, meteorites of a certain minimum size hit the earth with a rate of 2.34 per year (just guessing). This rate is again denoted by  $\lambda$ . The pdf of the exponential distribution is given by

$$p(x) = \lambda e^{-\lambda x}$$
 (note that  $x \ge 0$ ). (62)

The expectation of an exponential distribution is the reciprocal of the rate,  $E[X] = 1/\lambda$ . Figure 40 shows pdf's for some rates  $\lambda$ .

Modeling waiting times by the exponential distribution is based on the same assumptions as the Poisson distribution: the event-generating real-world sector must have an essentially infinite 'reservoir' of independent event-generating sources.

The exponential distribution plays a big role in modeling spiking neural networks (SNNs). Biological neurons communicate with each other by sending short "point-like" electrical pulses, called *spikes*. Many people believe that Nature invented communication by spikes to let the brain save energy – your head shouldn't heat up (like your PC does) when you do a hefty thinking job! For the same reason, microchip engineers have teamed up with deep learning researchers to design novel kinds of microchips for deep learning applications. These microchips contain neuron-like processing elements which communicate with each other by spikes.



Figure 40: The pdf of the exponential distribution for various values of the parameter  $\lambda$  (image source: https://commons.wikimedia.org/wiki/File: Exponential\_pdf.svg).

IBM and Intel have actually built such chips (check out "Tianjic chip" or "Intel Loihi" or "IBM NorthPole" if you want to learn more). Research about artificial SNNs for machine learning applications goes hand in hand with research in neuroscience. In both domains, one often uses models based on the assumption that the temporal pattern in a spike sequence sent by a neuron is a stochastic process called a *Poisson process*. In a Poisson process, the waiting times between two consecutive spikes are exponentially distributed. Recordings from real biological neurons often show a temporal randomness of spikes that can be almost perfectly modeled by a Poisson process.

**Learning** / estimation: The rate  $\lambda$  is the only parameter characterizing an exponential distribution. Training data: a sequence  $t_1, t_2, \ldots, t_N$  of time points where the event in question was observed. Then the rate can be estimated by  $\hat{\lambda} = 1/(N-1) \sum_{i=1,\ldots,N-1} t_{i+1} - t_i$ .

**The one-dimensional normal distribution.** Fanfare! Enter the queen of distributions! Bow in reverence! I am sure you know her from the tabloid press... For royal garments, she wears the pdf

$$p(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$
(63)

which is fully specified by its mean  $\mu$  and standard deviation (square root of variance)  $\sigma$ . It has the famous bell shape with  $\mu$  being the location of the maximum and  $\mu \pm \sigma$  being the locations of the zeros of the second derivative (Fig. 41).

The normal distribution with mean  $\mu$  and variance  $\sigma^2$  is denoted by  $\mathcal{N}(\mu, \sigma^2)$ . The normal distribution with zero mean and unit variance,  $\mathcal{N}(0, 1)$ , is called the *standard normal distribution*. The normal distribution is also called *Gaussian distribution* or simply *Gaussian*.



Figure 41: pdf of a normal distribution with mean 2 and standard deviation 1.

The normal distribution has a number of nice properties that very much facilitate calculations and theory. Specifically, linear combinations of normal distributed, independent normal RVs are again normal distributed:

**Proposition 7.1** Let  $X, Y : \Omega \to \mathbb{R}$  be two independent, normally distributed RVs with means  $\mu$  and  $\nu$  and variances  $\sigma^2$  and  $\tau^2$ . Then the weighted sum a X + b Y is normally distributed with mean  $a \mu + b \nu$  and variance  $a^2 \sigma^2 + b^2 \tau^2$ .

The majestic power of the normal distribution, which makes her reign almost universally over almost all natural phenomena, comes from one of the most central theorems of probability theory, the *central limit theorem*. It is stated in textbooks in a variety of (not always exactly equivalent) versions. It says, in brief, that one gets the normal distribution whenever random effects of many independent smallsized causes sum up to large-scale observable effects. The following definition makes this precise (you do not have to understand it — that would need a full course in mathematical probability theory — but you should at least have seen this definition):

**Definition 7.5** Let  $(X_i)_{i\in\mathbb{N}}$  be a sequence of independent, real-valued, square integrable random variables with nonzero variances  $Var(X_i) = E[(X_i - E[X_i])^2]$ . Then we say that the central limit theorem holds for  $(X_i)_{i\in\mathbb{N}}$  if the distributions  $P_{S_n}$  of the standardized sum variables

$$S_n = \frac{\sum_{i=1}^n (X_i - E[X_i])}{\sigma \left(\sum_{i=1}^n X_i\right)}$$
(64)

converge weakly to  $\mathcal{N}(0,1)$ .

Explanations (if you wish to give it a try — these concepts will not be queried in homeworks or exams):

- A real-valued random variable with pdf p is square integrable if its [uncentered] second moment, that is the integral  $E[X^2] = \int_{\mathbb{R}} x^2 p(x) dx$ , is finite.
- If  $(P_n)_{n \in \mathbb{N}}$  is a sequence of distributions over  $\mathbb{R}$ , and P a distribution (all over the same measure space  $(\mathbb{R}, \mathcal{B})$ ), then  $(P_n)_{n \in \mathbb{N}}$  is said to *converge weakly* to P if

$$\lim_{n \to \infty} \int f(x) P_n(dx) = \int f(x) P(dx)$$
(65)

for all continuous, bounded functions  $f : \mathbb{R} \to \mathbb{R}$ . You will find the notation of these integrals unfamiliar, and indeed you see here cases of *Lebesgue integrals* – a far-reaching generalization of the *Riemann integrals* that you know. Lebesgue integrals can deal with a far greater range of functions than the Riemann integral. Mathematical probability theory is formulated exclusively with the Lebesgue integral. We cannot give an introduction to Lebesgue integration theory in this course. Therefore, simply ignore the precise meaning of "weak convergence" and take home that sequences of distributions are required to converge to a target distribution in some subtly defined way.

Note that the central limit theorem is actually not a theorem — it is a property that a sequence of distributions may or may not have. There are many accompanying actual theorems of the kind, "A sequence  $(X_i)_{i\in\mathbb{N}}$  of random variables (or, equivalently, its associated sequence of distribution  $(P_{X_i})_{i\in\mathbb{N}}$ ) obeys the central limit theorem under these and those specific conditions".

The conditions under which the central limit theorem holds can be very general and weak. It is rather the common case, not the exception, that some sequence of RV's (or their distributions) has the central limit theorem property. A simple, important class of  $(X_i)$  for which the central limit theorem holds is obtained when the  $X_i$  are identically distributed (and, of course, are independent, square integrable and have nonzero variance). Notice that regardless of the shape of the distribution of each  $X_i$ , the distribution of the normalized sums converges to  $\mathcal{N}(0, 1)$ !

The classical demonstration of the central limit theorem is the *Galton board*, named after Sir Francis Galton (1822–1911), an English multi-scientist. The idea is to let little balls (or beans, hence this device is sometimes called "bean machine") trickle down a grid of obstacles which randomly deflect the ball left or right (Figure 42). It does not matter how, exactly, these deflections act — in the simplest case, the ball is just kicked right or left by one space grid unit with equal probability. The deeper the trickling grid, the closer will the resulting distribution be to a normal distribution. A nice video can be watched at https://www.youtube.com/watch?v=PM7z\_03o\_kk.

However, this simple case does not explain the far-reaching, general importance of the central limit theorem (rather, property). In textbooks one often finds statements like, "if the outcomes of some measurement procedure can be conceived to



Figure 42: The Dalton board. Compare text for explanation. Figure taken from https://janav.wordpress.com/2013/09/26/power-law/.

be the combined effect of many independent causal effects, then the outcomes will be approximately normal distributed". The "many independent causal effects" that are here referred to are the random variables  $(X_i)$ ; they will typically not be identically distributed at all. Still the central limit theorem holds under mild assumptions. Intuitively, all that one has to require is that none of the individual random variables  $X_i$  dominates all the others – the effects of any single  $X_i$  must asymptotically be "washed out" if an increasing number of other  $X_{i'}$  is entered into the sum variable  $S_n$ . In mathematical textbooks on probability you may find numerous mathematical conditions which amount to this "washing out". A special case that captures many real-life cases is the condition that the  $X_i$  are uniformly bounded, that is, there exists some b > 0 such that all possible values that  $X_i$ can take are in the interval [-b, b]. However, there exist much more general (nontrivial to state) conditions that likewise imply the central limit theorem. For our purposes, a good enough take-home message is

if  $(X_i)$  is a halfway reasonably behaved sequence of numerical RV's, then the normalized sums converge to the standard normal distribution.

The normal distribution plays an overwhelming role in applied statistics. One

often has to actually compute integrals of the pdf (63):

**Task:** compute the numerical value of 
$$\int_{a}^{b} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^{2}}{2\sigma^{2}}} dx$$
.

There is no closed-form solution formula for this task. Instead, the solution is found in a two-step procedure:

1. Transform the problem from its original version  $\mathcal{N}(\mu, \sigma^2)$  to the standard normal distribution  $\mathcal{N}(0, 1)$ , by using

$$\int_{a}^{b} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^{2}}{2\sigma^{2}}} dx = \int_{\frac{a-\mu}{\sigma}}^{\frac{b-\mu}{\sigma}} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x)^{2}}{2}} dx.$$
 (66)

In terms of probability theory, this means to transform the original,  $\mathcal{N}(\mu, \sigma^2)$ distributed RV X to a  $\mathcal{N}(0, 1)$ -distributed variable  $Z = (X - \mu)/\sigma$ . (The symbol Z is often used in statistics for standard normal distributed RVs.).

2. Compute the numerical value of the r.h.s. in (66) by using the cumulative density function of  $\mathcal{N}(0, 1)$ , which is commonly denoted by  $\Phi$ :

$$\int_{\frac{a-\mu}{\sigma}}^{\frac{b-\mu}{\sigma}} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x)^2}{2}} dx = \Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma}).$$

Since there is no closed-form solution for calculating  $\Phi$ , in former times statisticians found the solution in books where  $\Phi$  was tabulated. Today, statistics software packages run fast iterative solvers for  $\Phi$ .

**Learning** / estimation: The one-dimensional normal distribution is characterized by two parameters,  $\mu$  and  $\sigma^2$ . Given a dataset  $(x_i)_{i=1,\dots,N}$  of points in  $\mathbb{R}$ , the estimation formulas for these two parameters are

$$\hat{\mu} = \frac{1}{N} \sum_{i} x_i,$$
$$\hat{\sigma}^2 = \frac{1}{N-1} \sum_{i} (x_i - \hat{\mu})^2.$$

The *n*-dimensional normal distribution. If data points are not just real numbers but vectors  $\mathbf{x} = (x_1, \ldots, x_n)' \in \mathbb{R}^n$ , whose component RVs  $X_i$  fulfill the central limit theorem, the joint distribution of the RVs  $X_1, \ldots, X_n$  is the multidimensional normal distribution. It has the pdf

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} \det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)'\Sigma^{-1}(\mathbf{x}-\mu)\right).$$
 (67)

Here  $\mu$  is the expectation  $E[(X_1, \ldots, X_n)']$  and  $\Sigma$  is the covariance matrix of the *n* component variables, that is  $\Sigma(i, j) = E[(X_i - E[X_i])(X_j - E[X_j])]$ . Figure 43 shows the pdf of a 2-dimensional normal distribution. In geometrical terms, a multidimensional normal distribution is shaped as an *ellipsoid*, whose main axes coincide with the eigenvectors  $\mathbf{u}_i$  of the covariance matrix  $\Sigma$ . Like in PCA one can obtain them from the SVD: if  $UDU' = \Sigma$  is the singular value decomposition of  $\Sigma$ , the eigenvectors  $\mathbf{u}_i$  are the columns of U.



Figure 43: A pdf (indicated by contour lines) of a 2-dimensional normal distribution with expectation  $\mu$ . The vectors  $\mathbf{u}_1, \mathbf{u}_2$  are the principal axes of the ellipses which characterize the geometry of this pdf.

Multidimensional normal distributions (or simply, "Gaussians") are major players in machine learning. On the elementary end, one can use weighted combinations of Gaussians to approximate complex distributions - we will see this later in today's lecture. At the advanced end, there is an entire modern branch of ML, *Gaussian processes*, where complex distributions (and hyperdistributions) are modeled by infinitely many, interacting Gaussians. This is beyond the scope of our course.

**Learning** / estimation: Given a set of *n*-dimensional training data points  $(\mathbf{x}_i)_{i=1,\dots,N}$ , the expectation  $\mu$  and the covariance matrix  $\Sigma$  can be estimated from the training data in the obvious way:

$$\hat{\mu} = 1/N \sum_{i} \mathbf{x}_{i} \text{ and } \hat{\Sigma} = 1/(N-1) \sum_{i} (\mathbf{x}_{i} - \hat{\mu})(\mathbf{x}_{i} - \hat{\mu})^{\prime}$$

... and many more! The few common, named distributions that I displayed in this section are only meant to be illustrative picks from a much, much larger reservoir of well-known, completely analyzed, tabulated, pre-computed, and individually named distributions. The online book "Field Guide to Continuous Probability Distributions" (Crooks 2017) attempts a systematic overview. You should take home the following message:

- In 100 years or so of research, statisticians have identified hundreds of basic mechanisms by which nature generates random observations. In this subsection we looked at only two of them (i) intermittent rare "impact events" coming from large numbers of independent sources which hit some target system with a mean frequency λ, giving rise to Poisson and exponential distributions; and (ii) stochastic physical measurables that can be understood as the additive effect of a large number of different causes, which leads to the normal distribution.
- One way of approaching a statistical modeling task for a target distribution  $P_X$  is to
  - 1. first analyze and identify the nature of the physical (or psychological or social or economical...) effects that give rise to this distributions,
  - 2. then do a literature search (e.g. check out what G. E. Crooks says) or ask a statistics expert friend which known and named distribution is available that was tailored to capture exactly these effects, – this will likely give you a distribution formula that is shaped by a small number of parameters  $\theta$ ,
  - 3. then estimate  $\theta$  from available observation data, getting a distribution estimate  $\hat{\theta}$ , and
  - 4. use the theory that statisticians have developed in order to calculate confidence intervals (or similar accuracy tolerance measures) for  $\hat{\theta}$ , which
  - 5. finally allows you to state something like, "given the observation data, with a probability of 0.95, the true distribution  $\theta^{true}$  differs from the estimate  $\hat{\theta}$  by less than 0.01 percent."
- In summary, a typical classical statistical modeling project starts from wellargued assumptions about the type of the true distribution, then estimates the parameters of the distribution, then reports the estimate together with a quantification of the error bound or confidence level (or the like) of the estimate.
- Professionally documented statistical analyses will *always* state not only the estimated model, but also in addition quantify how accurate the model estimate is. This can take many forms, like error bars drawn around estimated parameters or stating "significance levels".

# 7.3 Mixture of Gaussians; maximum-likelihood estimates by EM algorithms

Approximating a distribution of points  $\mathbf{x} \in \mathbb{R}^n$  by a single *n*-dimensional Gaussian will often be a too coarse approximation which wipes out most of the interesting structure in the data. Consider, for a demonstration, the probability distribution of points in a rectangular subset of  $\mathbb{R}^2$  visualized in Figure 44. The true distribution shown in panel **A** could not be adequately approximated by a single 2-dimensional Gaussian, but if one invests a mixture of twenty of them, the resulting model distribution (panel **D**) starts to look useful.



Figure 44: A. A richly structured 2-dimensional pdf (darker: larger values of pdf; white: zero value of pdf). B. Same, discretized to a pmf. C. A MoG coverage of the pmf with 20 Gaussians, showing the ellipsoids coresponding to standard deviations. D. Contour plot of the pdf represented by the mixture of Gaussians. (The picture I used here is an iconic photograph from the history of aviation, July 1909. It shows Latham departing from Calais, attempting to cross the Channel in an aeroplane for the first time in history. The motor of his Antoinette IV monoplane gave up shortly before he reached England and he had to water. A few days later his rival Blériot succeeded. https://en.wikipedia.org/wiki/Hubert\_Latham

Approximating a complex probability distribution by such *mixtures of Gaussians* (MoG's) is a widely used technique. It comes with a transparent and robust optimization algorithm which locates, scales and rotates the various Gaussians

such that all together they give an optimal approximation. This optimization algorithm is an *expectation-maximization* (EM) algorithm. EM algorithms appear in many variants in machine learning, but all of them are designed according to the same basic principle, which every machine learning professional should know. I will use this occasion of optimizing MoGs to introduce you to the world of EM algorithms as a side-effect.

Let us first fix notation for MoGs. We consider probability distributions in  $\mathbb{R}^n$ — like the 2-dimensional distributions shown in Figure 44. A Gaussian (multidimensional normal distribution) in  $\mathbb{R}^n$  is characterized by its mean  $\mu$  and its *n*-dimensional covariance matrix  $\Sigma$ . We lump all the parameters inside  $\mu$  and  $\Sigma$ together and denote the resulting parameter vector by  $\theta$ . We denote the pdf of the Gaussian with parameters  $\theta$  by  $p(\cdot|\theta)$ . Now we can write down the pdf p of a MoG that is assembled from m different Gaussians as

$$p: \mathbb{R}^n \to \mathbb{R}^{\geq 0}, \ \mathbf{x} \mapsto \sum_{j=1}^m p(\mathbf{x}|\theta_j) \ P(j),$$
 (68)

where  $\theta_j = (\mu_j, \Sigma_j)$  are the parameters of the *j*th Gaussian, and where the *mixture* coefficients P(j) satisfy

$$0 \le P(j)$$
 and  $\sum_{j=1}^{m} P(j) = 1.$ 

For notational simplification we will mostly write  $\sum_{j=1}^{m} p(\mathbf{x}|j) P(j)$  instead of  $\sum_{j=1}^{m} p(\mathbf{x}|\theta_j) P(j)$ .

MoG's are used in unsupervised learning situations: given a data set  $(\mathbf{x}_i)_{i=1,...,N}$  of points in  $\mathbb{R}^n$ , one wants to find a "good" MoG approximation of the distribution from which the  $\mathbf{x}_i$  have been sampled. But... what does "good" mean?

This is the point to introduce the loss function which is invariably used when the task is to find a "good" distribution model for a set of training data points. Because this kind of loss function and this way of thinking about optimizing distribution models is not limited to MoG models but is the universal way of handling distribution modeling throughout machine learning, I devote a separate subsection to it and discuss it in general terms.

### 7.3.1 Maximum likelihood estimation of distributions

We consider the following general modeling problem: Given a dataset  $D = (\mathbf{x}_i)_{i=1,...,N}$ of points in  $\mathbb{R}^n$  and a hypothesis set  $\mathcal{H}$  of pdf's p over  $\mathbb{R}^n$ , where each pdf  $p \in \mathcal{H}$ is parametrized by  $\theta_p$ , we want to select that  $p_{opt} \in \mathcal{H}$  which "best" models the distribution from which the data points  $\mathbf{x}_i$  were sampled.

In supervised learning scenarios, we formalized this modeling task around a loss function which compared model outputs with teacher outputs. But now we find ourselves in an unsupervised learning situation. How can a pdf p be a "good" or "bad" model for a given set of training points?

In order to answer this question, one turns the question around: assuming the true distribution is given by a pdf p, how "likely" is it to get the training dataset  $D = (\mathbf{x}_i)_{i=1,\dots,N}$ ? That is, how well can the training datapoints be "explained" by p?

To make this precise, one considers the probability that the dataset D was drawn *given* that the true distribution has the pdf p (which is characterized by parameters  $\theta$ ).

That exactly this sample D was drawn from p is obviously a random event if one would repeat the entire data sampling procedure, one would have obtained another sample D'. It thus makes sense to ask, "what is the probability to draw just exactly this sample D?" The probability of drawing D is described by a pdf  $f(D|\theta)$  over  $(\mathbb{R}^n)^N$  (don't read on before you fully understand this). At this point one assumes that the datapoints in D have been drawn *independently* (each of them from a distribution with pdf p), which leads to

$$f(D|\theta) = \prod_{i=1}^{N} p(\mathbf{x}_i | \theta).$$
(69)

This pdf  $f(D|\theta)$  value gives the probability density of getting the sample Dif the underlying distribution is the one modeled by  $\theta$ . Conversely, one says that this value  $f(D|\theta)$  is the *likelihood* of the distribution  $\theta$  given the data D, and writes  $L(\theta | D)$ . The words "probability" and "likelihood" are two different words which refer to the same mathematical object, namely  $f(D|\theta)$ , but this object is read from left to right when speaking of "probability":

• "the probability to get data D given the distribution is modeled by  $\theta$ ", written  $f(D|\theta)$ ,

and it is read from right to left when speaking of "likelihood":

• "the likelihood of the model  $\theta$  given a dataset D", written  $L(\theta \mid D)$ .

If you meet someone who is able to correctly use the words "probability" vs. "likelihood", you can trust that person — s/he has received a good education in machine learning.

The likelihood pdf (69) is not suitable for doing numerical computations, because a product of very many small numbers (the  $p(\mathbf{x}_i | \theta)$ ) will lead to numerical underflow and become squenched to zero on digital computers. To avoid this, one always works with the *log likelihood* 

$$\mathcal{L}(\theta|D) = \log \prod_{i=1}^{N} p(\mathbf{x}_i \mid \theta) = \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \theta)$$
(70)

instead. A model  $\theta$  is "good" if it has a high (log)likelihood. Machine learning algorithms for finding the best distribution model p in a hypothesis set  $\mathcal{H}$  are

designed to find the maximum likelihood model

$$\theta^{\mathsf{ML}} = \operatorname*{argmax}_{\theta \in \mathcal{H}} \mathcal{L}(\theta \mid D).$$
(71)

Machine learners like to think of their learning procedures as minimizing some cost or loss. Thus one also finds in the literature the sign-inverted version of (71)

$$\theta^{\mathsf{ML}} = \underset{\theta \in \mathcal{H}}{\operatorname{argmin}} - \mathcal{L}(\theta \mid D), \tag{72}$$

which obviously is equivalent. If one writes out the  $\mathcal{L}(\theta \mid D)$  one sees the structural similarity of this unsupervised learning task with the supervised loss-minimization task which we met earlier in this course (e.g. (42)):

$$\theta^{\mathsf{ML}} = \operatorname*{argmin}_{\theta \in \mathcal{H}} - \sum_{i=1}^{N} \log p(\mathbf{x}_i \,|\, \theta).$$

## 7.3.2 Maximum-likelihood estimation of MoG models by an EM algorithm

Back to our concrete business of modeling a distribution represented by a dataset  $D = (\mathbf{x}_i)_{i=1,\dots,N}$  by a MoG. In order to to find a maximum-likelihood solution for this MoG modeling task, one has to answer two questions:

- 1. What is the appropriate number m of mixture components? Setting m determines the class of candidate solutions in which an optimization algorithm can search for the maximum-likelihood solution. The set  $\mathcal{H}_m$  contains all mixtures of m Gaussians. We have here a case of a model class inclusion sequence as discussed in Section 6.4. For determining an appropriate model flexibility m the routine way would be to carry out a cross-validation.
- 2. If an appropriate m has been determined, how should the parameters  $\theta_j$  (where j = 1, ..., m) of the participating Gaussians be set in order to solve the optimization problem

$$\theta^{\mathsf{ML}} = \operatorname*{argmax}_{\theta \in \mathcal{H}_m} \mathcal{L}(\theta \mid D)?$$
(73)

The second question is answered by employing an *expectation-maximization* (EM) algorithm. EM algorithms can often be designed for maximizing the (log) likelihoods of parametrized distributions. There is a general scheme for designing such algorithms, and general theorems concerning their convergence properties. The EM principle had been introduced by Dempster, Laird, and Rubin 1977 — one of these classical landmark papers that will forever continue to be continuously cited (as of 2021, 65K Google cites).

EM algorithms are called to help in computational statistics when it comes to estimating model parameters  $\theta$  from *incomplete* or *partially observable* data. The principle of designing EM algorithms is a cornerstone of many important machine learning algorithms that are using "incomplete" training data. What "incomplete" means depends on the specific case. For instance, it might mean missing data points in time series measurements, or measurements of physical systems where the available sensors give only a partial account of the system state. The general idea is that the observed data have been caused or influenced by some "hidden" mechanism which by itself could not be observed. A solid and reliable model of the distribution of the observed measurable data should somehow include a model of these unobserved hidden mechanisms. This is mathematically done by creating models with two sorts of random variables: the *observables* (lumped together in a random variable X) and the *hiddens* (collectively written as Y). If one manages to estimate a good model of the joint distribution of both X and Y, one can use this "complete" model for many purposes, for instance

- for filling in "best guesses" for missing values in time series data where observations are unavailable for certain time points (like handwritten texts with unreadable words in them);
- for estimating *hidden Markov models* (HMMs) of stochastic timeseries; HMMs are the most widely used models for stochastic processes with memory;
- for estimating models of complex systems that are described by many interacting random variables, some of which are not observable — an example from human-machine interfacing: modeling the decisions expected from a user assuming that these decisions are influenced by emotional states that are not directly observable; such models are called *Bayesian networks* or more generally, *graphical models*;
- for a host of other practical modeling tasks, like our MoG optimization task.

Because of the widespread usefulness of EM algorithms I will now carry you through a general, abstract explanation of the underlying principle. This is not easy stuff but worth the effort. I follow the exposition given in Roweis and Ghahramani 1999, which also covers other versions of EM for other machine learning problems. I use however another notation than these authors and I supply more detail. In blue font I will insert how the abstract picture translates to our specific MoG case. The following treatment will involve a lot of probability formalism — I consider it a super exercise in becoming friends with random variables and products of random variables! try not to give up on the way! I suggest that in a first reading you skip the blue parts and concentrate on the abstract picture alone.

The general situation is the following. Let X be an observable random variable and let Y be a hidden random variable, which take values in value sets ("sample spaces")  $S_X$  and  $S_Y$ , respectively. X produces the visible training data D and Y produces the hidden data which have a causal impact on D but cannot be observed.

In the MoG case, X would be the random variable which generates the entire training sample  $(\mathbf{x}_i)_{i=1,\ldots,N}$ , thus here  $S_X = (\mathbb{R}^n)^N$ . Y would be a random variable that decides which of the *m* Gaussians is used to generate  $\mathbf{x}_i$ , thus  $S_Y = \{1, \ldots, m\}^N$ . The RV Y is a product  $Y = \bigotimes Y_i$  of i.i.d. RVs  $Y_i$ , where  $Y_i$  takes values  $j \in \{1, \ldots, m\}$ . Concretely,  $Y_i$  selects one of the *m* Gaussians by a random choice weighted by the probability vector  $(P(1), \ldots, P(m))$ . Thus the value of  $Y_i$  is  $j_i$ , an element of  $\{1, \ldots, m\}$ .

The RV X is a product  $X = \bigotimes X_i$  of i.i.d. RVs  $X_i$ , where  $X_i$  generates the *i*-th training data point  $\mathbf{x}_i$  by a random draw from the  $j_i$ -th Gaussian, that is from the normal distribution  $\mathcal{N}(\mu_{j_i}, \Sigma_{j_i})$ . Figure 45 illustrates what it means not to know the values of the hidden variable Y.



Figure 45: A point set sampled from a mixture of Gaussians. Left: what we (don't) know if we don't know from which Gaussian a point was drawn. Right: the "hidden" variable made visible. (Figure copied from an online presentation of Christopher M. Bishop, no longer accessible)

Let  $\theta$  be a vector of parameters describing the joint distribution  $P_{X,Y}$  of hidden and visible RVs.

In our MoG case,  $\theta = (\mu_1, \ldots, \mu_m, \Sigma_1, \ldots, \Sigma_m, P(1), \ldots, P(m))'$ .

Let D be the observed sample, that is, the result from a random draw with X. In the MoG scenario,  $D = (\mathbf{x}_i)_{i=1,\dots,N}$ .

The objective of an EM algorithm is to determine  $\theta$  such that the likelihood  $L(\theta) = P(X = D | \theta)$  becomes maximal, or equivalently, such that the log likelihood

$$\mathcal{L}(\theta \mid D) = \log P(X = D \mid \theta)$$

becomes maximal.

At this point we would have to start using different notations (pmf's versus pdf's) depending whether X and Y are discrete or continuous RVs. I will carry

out the formal description only for the case that both X and Y are continuous, which means that their joint distribution can be described by a pdf  $p_{X,Y}$  defined on  $S_X \times S_Y$ . The marginal distributions of X and Y are then given by parametrized pdf's  $p_X(x|\theta) = \int_Y p_{X,Y}(x, y|\theta) dy$  and  $p_Y(y|\theta) = \int_X p_{X,Y}(x, y|\theta) dx$ . We can then switch to a description of distributions using only pdfs, that is, the distribution of samples D is described by the pdf  $p_X(D|\theta)$ .

In the case of all-continuous distributions, the log likelihood of  $\theta$  becomes

$$\mathcal{L}(\theta \mid D) = \log p_X(D|\theta).$$
(74)

The MoG case has a mix of types: X gives values in the continuous space  $(\mathbb{R}^n)^N$ and Y gives values in the discrete space  $\{1, \ldots, m\}^N$ . This makes it necessary to use a mix of pdf's and pmf's when working out the MoG case in detail.

The difficulty we are facing is that the pdf value  $p_X(D|\theta)$  depends on the values that the hidden variables take, that is,

$$\mathcal{L}(\theta \mid D) = \log p_X(D|\theta) = \log \int_{S_Y} p_{X,Y}(D, y \mid \theta) \, dy$$

In the MoG case, the integral over  $S_Y$  becomes a sum over  $S_Y$ :

$$p_X(D|\theta) = \sum_{(j_1,\dots,j_N)\in\{1,\dots,m\}^N} P(Y = (j_1,\dots,j_N)) p_{X|Y=(j_1,\dots,j_N)}(D)$$
  
= 
$$\sum_{(j_1,\dots,j_N)\in\{1,\dots,m\}^N} \prod_{i=1,\dots,N} P(Y_i = j_i) p_{X_i|Y_i=j_i}(\mathbf{x}_i),$$

where  $p_{X|Y=(j_1,\ldots,j_N)}$  is the pdf of the conditional distribution of X given that the hidden variables take the value  $(j_1,\ldots,j_N)$  and  $p_{X_i|Y_i=j_i}$  is the pdf of the conditional distribution of  $X_i$  given that the hidden variable  $Y_i$  takes the value  $j_i$ .

Now let q be any pdf for the hidden variables Y. Then we can obtain a lower bound on log  $p_X(D|\theta)$  by

$$\mathcal{L}(\theta \mid D) = \log \int_{S_Y} p_{X,Y}(D, y \mid \theta) \, dy =$$

$$= \log \int_{S_Y} q(y) \frac{p_{X,Y}(D, y \mid \theta)}{q(y)} \, dy$$

$$\geq \int_{S_Y} q(y) \log \frac{p_{X,Y}(D, y \mid \theta)}{q(y)} \, dy$$

$$= \int_{S_Y} q(y) \log p_{X,Y}(D, y \mid \theta) \, dy - \int_{S_Y} q(y) \log q(y) \, dy \qquad (75)$$

$$=: \mathcal{F}(q, \theta),$$

(76)

where the inequality follows from a version of Jensen's inequality which states that

$$E[f \circ Y] \le f(E[Y])$$

for any concave function f (the log is concave).

EM algorithms maximize the lower bound  $\mathcal{F}(q,\theta)$  by alternatingly and iteratively maximizing  $\mathcal{F}(q,\theta)$  first with respect to q, then with respect to  $\theta$ , starting from an initial guess  $q^{(0)}, \theta^{(0)}$  which is then updated by:

Expectation step: 
$$q^{(k+1)} = \operatorname*{argmax}_{q} \mathcal{F}(q, \theta^{(k)}),$$
 (77)

Maximization step: 
$$\theta^{(k+1)} = \operatorname*{argmax}_{\theta} \mathcal{F}(q^{(k+1)}, \theta).$$
 (78)

The maximum in the E-step is obtained when q is the conditional pdf of Y given the data D,

$$q^{(k+1)} = p_{Y|X=D,\theta^{(k)}},\tag{79}$$

because then  $\mathcal{F}(q^{(k+1)}, \theta^{(k)}) = \mathcal{L}(\theta^{(k)} | D)$ :

$$\begin{split} \mathcal{F}(p_{Y|X=D,\theta^{(k)}}, \theta^{(k)}) &= \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_{X,Y}(D, y \mid \theta^{(k)}) \, dy - \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_{Y|X=D,\theta^{(k)}}(y) \, dy \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log \frac{p_{X,Y}(D, y \mid \theta^{(k)})}{p_{Y|X=D,\theta^{(k)}}(y)} \, dy \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_X(D \mid \theta^{(k)}) \, dy \\ &= \int_{S_Y} \frac{p_{X,Y}(D, y \mid \theta^{(k)})}{p_X(D \mid \theta^{(k)})} \log p_X(D \mid \theta^{(k)}) \, dy \\ &= \frac{\log p_X(D \mid \theta^{(k)})}{p_X(D \mid \theta^{(k)})} \int_{S_Y} p_{X,Y}(D, y \mid \theta^{(k)}) \, dy \\ &= \log p_X(D \mid \theta^{(k)}) \\ &= \mathcal{L}(\theta^{(k)} \mid D). \end{split}$$

In concrete algorithms, in order to compute the conditional distribution  $p_{Y|X=D,\theta^{(k)}}$ , one makes assumptions about the class of distributions from which  $p_{Y|X=D,\theta^{(k)}}$ comes (e.g. in our MoG example it would come from the class of pmf distributions over  $S_Y = \{1, \ldots, m\}^N$ ), and compute a set of parameters which together fully characterize  $p_{Y|X=D,\theta^{(k)}}$ . Such parameters are called a *sufficient statistic* for a distribution — for instance, the mean and variance are a sufficient statistic for a normal distribution (In our MoG example, a sufficient statistic for the distribution of  $p_{Y|X=D,\theta^{(k)}}$  is the set of all probabilities  $P(Y_i = j \mid X_i = \mathbf{x}_i, \theta^{(k)})$ , see Equation (82) below). Such parameters can typically be expressed as expectations (e.g. the mean and variance of a distribution are the expected numerical value and the expected squared distance from the mean), which gave this step its name, the Expectation step.

The maximum in the M-step is obtained when the first term in (75) is maximized, because the second term does not depend on  $\theta^{(k)}$ :

$$\theta^{(k+1)} = \operatorname{argmax}_{\theta} \int_{S_Y} q^{(k+1)}(y) \log p_{X,Y}(D, y \mid \theta) \, dy$$
$$= \operatorname{argmax}_{\theta} \int_{S_Y} p_{Y \mid X = D, \theta^{(k)}}(y) \log p_{X,Y}(D, y \mid \theta) \, dy$$
(80)

How the M-step is concretely computed (i.e. how the argmax problem (80) is solved) depends on the particular kind of model. Can be tricky and worth a publication to find an M-step algorithm for your new kind of model.

Because we have  $\mathcal{F} = \mathcal{L}(\theta^{(k)} | D)$  before each M-step, and the E-step does not change  $\theta^{(k)}$ , and  $\mathcal{F}$  cannot decrease in an EM-double-step, the sequence  $\mathcal{L}(\theta^{(0)} | D), \mathcal{L}(\theta^{(1)} | D), \ldots$  monotonously grows toward a supremum. The iterations are stopped when  $\mathcal{L}(\theta^{(k)} | D) = \mathcal{L}(\theta^{(k+1)} | D)$ , or more realistically, when a predefined number of iterations is reached or when the growth rate falls below a predetermined threshold. The last parameter set  $\theta^{(k)}$  that was computed is taken as the outcome of the EM algorithm.

It must be emphasized that EM algorithms steer toward a local maximum of the likelihood. If started from another initial guess, another final parameter set may be found. Here is a summary of the EM principle:

- **E-step:** Estimate the distribution  $p_{Y|X=D,\theta^{(k)}}(y)$  of the hidden variables, given data D and the parameters  $\theta^{(k)}$  of a preliminary model. This can be intuitively understood as inferring knowledge about the hidden variables, thereby *completing* the data.
- **M-step:** Use the (preliminary) "knowledge" of the complete data (given by D for the visibles and by  $p_{Y|X=D,\theta^{(k)}}(y)$  for the hidden variables) to compute a maximum likelihood model  $\theta^{(k+1)}$ .

There are other ways to compute maximum likelihood solutions in tasks that involve visible and hidden variables. Specifically, one can often invoke a gradient descent optimization. A big advantage of EM over gradient descent is that EM does not need a careful tuning of algorithm control parameters (like learning rates, an eternal bummer in gradient descent methods), simply because there are no tuning parameters. Furthermore, EM algorithms are typically numerically robust, which cannot be said about gradient descent algorithms.

Now let us put EM to practice for the MoG estimation. For better didactic transparency, I will restrict my treatment to a special case where we require all
Gaussians to be spheric, that is, their covariance matrices are of the form  $\Sigma = \sigma^2 I_n$  where  $\sigma^2$  is the variance of the Gaussian in every direction and  $I_n$  is the *n*-dimensional identity matrix. The general case of mixtures of Gaussians composed of member Gaussians with arbitrary  $\Sigma$  is described in textbooks, for instance Duda, P. E. Hart, and Stork 2001. And anyway, you would probably use a ready-made online tool for MoG estimation... Here we go.

A MoG pdf with *m* spherical components is given by the vector of parameters  $\theta = (\mu_1, \ldots, \mu_m, \sigma_1^2, \ldots, \sigma_m^2, P(1), \ldots, P(m))'$ . This gives the following concrete optimization problem:

$$\theta^{\mathsf{ML}} = \operatorname{argmax}_{\theta \in \mathcal{H}_m} \mathcal{L}(\theta \mid D)$$
  
=  $\operatorname{argmax}_{\theta \in \mathcal{H}_m} \sum_{i=1}^N \log \left( \sum_{j=1}^m \frac{1}{(2\pi\sigma_j^2)^{n/2}} \exp\left(-\frac{\|\mathbf{x}_i - \mu_j\|^2}{2\sigma_j^2}\right) P(j) \right)$ (81)

Assume that we are after iteration k and want to estimate  $\theta^{(k+1)}$ . In the E-step we have to compute the conditional distribution of the hidden variable Y, given data D and the preliminary model

$$\theta^{(k)} = (\mu_1^{(k)}, \dots, \mu_m^{(k)}, \sigma_1^{2(k)}, \dots, \sigma_m^{2(k)}, P^{(k)}(1), \dots, P^{(k)}(m))'.$$

Unlike in the treatment that I gave for the general case, where we assumed Y to be continuous, now Y is discrete. Its conditional distribution is given by the probabilities  $P(Y_i = j | X_i = \mathbf{x}_i, \theta^{(k)})$ , where i = 1, ..., N and j = 1, ..., m. These probabilities are

$$P(Y_i = j \mid X_i = \mathbf{x}_i, \theta^{(k)}) = \frac{p^{(k)}(\mathbf{x}_i \mid Y_i = j) \ P^{(k)}(j)}{p_X^{(k)}(\mathbf{x}_i)},$$
(82)

where  $p^{(k)}(\mathbf{x}_i | Y_i = j)$  is the pdf of the *j*-th Gaussian with parameters  $\mu_j^{(k)}, \sigma_j^{2(k)}$ and

$$p_X^{(k)}(\mathbf{x}_i) = \sum_{j=1}^m P^{(k)}(j) \ p^{(k)}(\mathbf{x}_i \mid Y_i = j).$$

In the M-step we have to find maximum likelihood estimates for all parameters in  $\theta$ . I do not give a derivation here but just report the results, which are intuitive enough:

$$\mu_{j}^{(k+1)} = \frac{\sum_{i=1}^{N} P(Y_{i} = j | X_{i} = \mathbf{x}_{i}, \theta^{(k)}) \mathbf{x}_{i}}{\sum_{i=1}^{N} P(Y_{i} = j | X_{i} = \mathbf{x}_{i}, \theta^{(k)})},$$
  

$$\sigma_{j}^{2(k+1)} = \frac{1}{n} \frac{\sum_{i=1}^{N} P(Y_{i} = j | X_{i} = \mathbf{x}_{i}, \theta^{(k)}) \|\mu_{j}^{(k+1)} - \mathbf{x}_{i}\|^{2}}{\sum_{i=1}^{N} P(Y_{i} = j | X_{i} = \mathbf{x}_{i}, \theta^{(k)})}, \text{ and }$$
  

$$P^{(k+1)}(j) = \frac{1}{N} \sum_{i=1}^{N} P(Y_{i} = j | X_{i} = \mathbf{x}_{i}, \theta^{(k)}).$$

I conclude this sudorific (look this up in a dictionary) subsection with a little EM-for-MoG demo that I copied from an online presentation of Christopher Bishop (now apparently no longer online). The sample data  $\mathbf{x}_i \in \mathbb{R}^2$  come from observations of the Old Faithful geysir in the Yellowstone National Park (Figure 46).



Figure 46: A two-dimensional dataset.

Figure 47 shows 20 EM iterations with m = 2 Gaussian components. The first panel shows the initial guess. Color codes are the estimated probabilities  $P(Y_i = j | X_i = \mathbf{x}_i, \theta^{(k)})$ . The MoG modeling of the aeroplane image in Figure 44 was also done with the EM algorithm, in a version that allowed for arbitrary covariance matrices  $\Sigma_j$ . I did it myself, programming it from scratch, just to prove to myself that all of this really works. It does!

### 7.4 Parzen windows

After the heavy-duty work of coming to grips with the EM algorithm, let us finish this section with something easy, for chilling out. Parzen windows!

Parzen windows are a simple representative of the larger class of *kernel-based* representations, providing an alternative to mixture models for representing pdf's. These representations are *non-parametric* — there is no parameter vector  $\theta$  for specifying a Parzen window approximation to a pdf. To introduce Parzen windows, consider a sample of 5 real-valued points shown in Figure 48. Centered at each sample point we place a unit square area on the x-axis. Weighing them each by 1/5 and summing them gives an intuitively plausible representation of a pdf.

To make this example more formal and general, consider *n*-dimensional data points  $\mathbf{x}_i$ . Instead of a unit-length square, we wish to make these points the centers of *n*-dimensional hypercubes of side length *d*. We need a function *H* that indicates which points around  $\mathbf{x}_i$  fall into the hypercube centered at  $\mathbf{x}_i$ . To this



Figure 47: The EM algorithm at work on the Old Faithful dataset.

end we introduce a kernel function, in this case also known as Parzen window,

$$H: \mathbb{R}^n \to \mathbb{R}^{\geq 0}, \quad (x_1, \dots, x_n)' \mapsto \begin{cases} 1, & \text{if } |x_j| < 1/2 \text{ for } j = 1, \dots, n \\ 0, & \text{else} \end{cases}$$
(83)

which makes H the indicator function of a unit hypercube centered at the origin. Using H, we get the *n*-dimensional analog of the staircase pdf in Figure 48 for a sample  $D = (\mathbf{x}_i)_{i=1,...,N}$  by

$$p^{(D)}(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{d^n} H\left(\frac{\mathbf{x} - \mathbf{x}_i}{d}\right), \tag{84}$$

observing that the volume of such a cube is  $d^n$ . The superscript (D) in  $p^{(D)}$  is meant to indicate that the pdf depends on the sample D.

Clearly, given some sample  $(\mathbf{x}_i)_{i=1,\dots,N}$ , we do not believe that such a rugged staircase reflects the true probability distribution the sample was drawn from. We would rather prefer a smoother version. This can be easily done if we use smoother kernel functions. A standard choice is to use multivariate Gaussians with diagonal covariance matrix and uniform standard deviations  $\sigma =: d$  for H. This turns (84)



Figure 48: Rectangular Parzen window representation of a distribution given by a sample of 5 real numbers. The sample points are marked by colored circles. Each data point lies in the middle of a square "Parzen window", that is, a rectangular pdf centered on the point. Weighted by 1/5 (colored rectangles) and summed (solid black staircase line) they give a pdf.

into

$$p^{(D)}(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{(2\pi d^2)^{n/2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2d^2}\right)$$
$$= \frac{1}{N} \sum_{i=1}^{N} \frac{1}{d^n} \frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2} \|\frac{\mathbf{x} - \mathbf{x}_i}{d}\|^2\right)$$
(85)

where the second line brings the expression to a format that is analog to (84).

It is clear that any nonnegative kernel function H which integrates to unity can be used in an equation of this sort such that the resulting  $p^{(D)}$  will be a pdf. The scaling factor d determines the width of the Parzen window and thereby the amount of smoothing. Figure 49 illustrates the effect of varying d.



Figure 49: The effect of choosing different widths *d* in representing a 5-point, 2dimensional sample by Gaussian windows. (Taken from the online set of figures of the book by Duda, Hart & Stork, ftp://ftp.wiley.com/public/sci\_tech\_ med/pattern/)

Comments:

- Parzen window representations of pdfs are "non-parametric" in the sense that the shape of such a pdf is determined by a sample (plus, of course, by the shape of the kernel function, which however mainly serves a smoothing purpose). This can render Parzen window representations computationally expensive, because if the sample size is large, a large number of data points have to be stored (and accessed if the pdf is going to be used).
- The basic Parzen windowing scheme, as introduced here, can be refined in many ways. A natural way to improve on it is to use different widths d for different sample points  $\mathbf{x}_i$ . One then makes d narrow in regions of the sample set which are densely populated, and wide in regions that are only thinly covered by sample points. One way of doing that (which I invented while I was writing this there are many ways to go) would be to (i) choose a reasonably small integer K; (ii) for each sample point  $\mathbf{x}_i$  determine its K nearest neighbors  $\mathbf{x}_{i1}, ..., \mathbf{x}_{iK}$ ; (iii) compute the mean squared distance  $\delta$  of  $\mathbf{x}_i$  from these neighbors, (iv) set d proportional to this  $\delta$  for this sample point  $\mathbf{x}_i$ .
- As Figure 49 demonstrates, the width d has a strong effect on the Parzen pdf. If d is too large, the smoothing becomes too strong, and information contained in the sample is smoothed away underfitting! In contrast, when d is too small, all that we see in the resulting Parzen pdf is the individual data points the pdf then models not a distribution, but just the sample. Overfitting! Here one should employ a cross-validation scheme to optimize d, minimizing the validation loss defined by  $-\sum_{\mathbf{v} \in V} \log p^{(T)}(\mathbf{v} \mid d)$ , where V is the validation set, T the training set, and  $p^{(T)}(\mathbf{v} \mid d)$  the Parzen pdf obtained from the training set using width d.
- The Parzen-window based distribution (85) will easily lead to numerical underflow problems for arguments  $\mathbf{x}$  which are not positioned close to a sample point, especially if d is set to small values. A partial solution is to use log probabilities instead, i.e. use

log 
$$p^{(D)}(\mathbf{x}) = \log\left(\frac{1}{N} \sum_{i=1}^{N} \frac{1}{(2\pi d^2)^{n/2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2d^2}\right)\right).$$
 (86)

Still this will not usually solve all underflow problems, because the sum-exp terms within the log still may underflow. Here is a trick to circumvent this problem, known as the "log-sum-exp" trick. Exploit the following:

$$log(exp(-A) + exp(-B)) = = log(exp(-A + C) exp(-C) + exp(-B + C) exp(-C)) = -C + log(exp(-A + C) + exp(-B + C)),$$

where you use  $C = \max(A, B)$ .

# 8 Bayesian model estimation

Machine learning is based on probability, and probability is ... what? — We don't know what probability is! After centuries of thinking, philosophers and mathematicians have not arrived at a general consensus of how to best understand randomness. The proposals that are on the table can be broadly grouped into *objectivist* and *subjectivist* interpretations of "probability".

According to the objectivist view, probability resides physically in the real world — it is a phenomenon of nature, as fundamentally a property of physical reality as for instance "time" or "energy".

According to the subjectivist view, probability describes an observer's subjective opinion on something observed — a degree of belief, of uncertainty, of plausibility of judgement, or missing information etc.

Calling the two views "objectivist" versus "subjectivist" is the philosophers' terminology. Statisticians and machine learners rather call it *frequentist statistics* versus *Bayesian statistics*.

This duality of understandings of "probability" has left a strong mark on machine learning. The two ways of thinking about probability lead to two different ways of designing learning algorithms. Both are important and both are in daily use. A machine learning professional should be aware of the fundamental intuitions behind the two sorts of algorithms, and know when to use which. In this section I explain the fundamental ideas behind the two understandings of probability, outline the general principles of Bayesian learning algorithms, and demonstrate them in a case study of great importance in bioinformatics, namely the classification of proteins.

### 8.1 The ideas behind frequentist statistics

When probability is regarded as a phenomenon of nature, there should be ways to *measure* it. The standard proposition of how one can measure probability is by *relative frequency counting*. For instance, in the textbook example of a loaded (unfair) die, the die may have the physical property that P(X = 6) = 1/5 (as opposed to P(X = 6) = 1/6 for a fair die), where X is the RV which gives the outcomes of throwing-the-die experiments. This property of the die could be experimentally measured by *repeating* the die-throwing act many times. This would give a sequence of outcomes  $x_1, x_2, x_3, \ldots$  where  $x_i \in \{1, \ldots, 6\}$ . After N such throws, an estimate of the quantity P(X = 6) is calculated by

$$\hat{P}_N(X=6) = \frac{\text{number of outcomes } x_i = 6}{N}, \tag{87}$$

where N is the number of throws that the experimentalist measuring this probability has carried out.

We generally use the hat symbol  $\hat{\cdot}$  on top of some variable to denote a numerical estimate based on a finite amount of observation data. The true probability P(X =

6) = 1/5 would then become "measurable in principle" by

(\*) 
$$P(X = 6) = \lim_{N \to \infty} \hat{P}_N(X = 6).$$
 (88)

A fundamental mathematical theorem in frequentist probability theory — actually, the fundamental law which justifies frequentist probability theory in the first place — is the law of large numbers. It states that if one carries out an infinite sequence of independent repetitions of the same numerical measurement experiment, obtaining a sequence of measurement values  $x_1, x_2, \ldots$ , where  $x_i \in \mathbb{R}$ , then the mean value  $\mu_N = (1/N) \sum_{i=1}^N x_i$  of initial sequences up to N will "almost always" converge to the same number  $\mu = E[X]$ , called the *expectation* of X. Stating and proving this theorem in formal rigor has taken mathematicians a long time and was achieved by Kolmogorov no earlier than in the early 1930's. A precise formulation needs the full apparatus of measure-theoretic probability theory (which is introduced to some useful extent in my lecture notes on Principles of Statistical Modeling).

More generally, considering some random variable X which takes values in a sample space S, one considers subsets  $A \subseteq S$  (called *events*) and then defines

The frequentist view of probability: The probability  $P(X \in A)$  that the outcome of sampling a datapoint with procedure X gives an outcome in the value range A is the relative frequency (in the limit of carrying out infinitely many independent and unbiased trials of carrying out a measurement procedure X) of obtaining a measurement value in A.

If one looks at this "definition" with a critical mind one will find that it is loaden with difficulties.

- First, it defines a "measurement" process that is not feasible in reality because one cannot physically carry out infinitely many measurement trials. This is maybe not really disturbing because *any* measurement in the sciences (say, of a voltage) is imprecise and one gets measurements of increasing precision by repeating the measurement, just as when one measures a probability.
- **Second**, it does not inform us about how, exactly, the "repeated trials" of executing X should be done in order to be "unbiased". What does that mean in terms of experimental procedures? This is a very critical issue. To appreciate its impact, consider the example of a bank who wishes to estimate the probability that a customer will fail to pay back a loan. In order to get an estimate of this probability, the bank can only use customer data collected in the *past*, but wants to base creditworthyness decisions for *future* customers on those data. Picking only past data points to base probability estimates on

hardly can qualify as an "absolutely unbiased" sampling of data, and in fact the bank may grossly miscalculate credit risks when the general customer body or their economical conditions change over time. These difficulties have of course been recognized in practical applications of statistics. Textbooks and statistics courses for psychologists and economists contain entire chapters with instructions on how to collect "unbiased samples".

**Third,** if one repeats the repeated measurement, say by carrying out one measurement sequence giving  $x_1, x_2, \ldots$  and then another one giving  $x'_1, x'_2, \ldots$ , the values  $\hat{P}_N$  from Equation (88) are bound to differ between the two series. The limit indicated in Equation (88) must somehow be robust against different versions of the  $\hat{P}_N$ . Mathematical probability theory offers several ways to rigorously define limits of series of probability quantities which we do not present here. Equation (88) is suggestive only and I marked it with a (\*) to indicate that it is not technically complete.

Among these three difficulties, only the second one is really problematic. The first one is just a warning that in order to measure a probability with increasing precision we need to invest an increasingly large effort — but that is the same for other measurables in the sciences. The third difficulty can be fully solved by a careful definition of suitable limit concepts in probability theory. But the second difficulty is fundamental and raises its ugly head whenever statistical assertions about reality are made. It is the reason for the grain of truth in the ugly saying, "all statistics lie".

In spite of these difficulties, the objectivist view on probability in general, and the frequentist account of how to measure it in particular, is widely shared among empirical scientists. It is also the view of probability which is commonly taught in courses of statistics for mathematicians. A student of mathematics may not even have heard of Bayesian statistics at the end of his/her studies. In machine learning, the frequentist view often leads to learning algorithms which are based on the principle of maximum likelihood estimation of distributions — as we have seen in Section 7.3.1. In fact, all what I wrote in these lecture notes up to now was implicitly based on a frequentist understanding of probability.

# 8.2 The ideas behind Bayesian statistics

Subjectivist conceptions of probability also have led to mathematical formalisms that can be used in statistical modeling. A hurdle for the student here is that a number of different formalisms exist which reflect different modeling goals and approaches, and tutorial texts are rare.

The common starting point for subjectivist theories of probability is to cast "probability" as a subjective degree of belief, of certainty of judgement, or plausibility of assertions, or similar — instead of as an objective property of real-world systems. Subjectivist theories of probability do not develop analytical tools to describe randomness in the world. Instead they provide formalisms that code how rational agents (you, I, and AI systems) should *think about* the world, in the face of various kinds of uncertainty in their knowledge and judgements. The formalisms developed by subjectivists can by and large be seen as generalizations of classical *logic*. Classical logic only knows two truth values: true or false. In subjectivist versions of logic formalisms, a proposition can be assigned graded degrees of "belief", "plausibility", etc. For a first impression, contrast a classical-logic syllogism like

> (i) We know: if A is true, then B is true (ii) We know: A is true

> > (iii) We conclude: B is true

with a "plausibility reasoning rule" like

(i) We know: if A is true, then B becomes more plausible
 (ii) We know: B is true

(iii) We conclude: A becomes more plausible

This example is taken from Jaynes 2003, where a situation is described in which a policeman sees a masked man running away from a juweler's shop whose window was just smashed in. The plausibility rule captures the policeman's inference that the runner is likely, as follows. Set for A: "the person is a thief", and for B: "the person runs away from a smashed-in shop window". Using the above reasoning rule, the policeman will argue: (i) I saw this person running away from the smashed window; and (ii) I know that when a person runs away from a smashed shop window, it's some evidence that he's the thief; hence (iii) there's some evidence that this person is the thief. From reasoning rules like this taken as starting points, a number of logic formalisms have been devised which enrich/modify classical twovalued logic in various ways. If you want to explore these areas a little further, the Wikipedia articles probabilistic logic, Dempster-Shafer theory, fuzzy logic, or Bayesian probability are good entry points. In some of these formalisms the Kolmogorov axioms of frequentist probability re-appear as part of the respective mathematical apparatus. Applications of such formalisms arise in classical, logicoriented AI (modeling reasoning under uncertainty), human-machine interfaces (supporting discourse generation), game theory and elsewhere.

The discipline of statistics has almost entirely been developed in an objectivist spirit, firmly rooted in the frequentist interpretation of probability. Machine learning also in large parts roots in this view. However, a certain subset of machine learning models and computational procedures have a subjectivist component. These techniques are referred to as *Bayesian model estimation* methods. Bayesian modeling is particularly effective and important when training datasets are small. I will explain the principle of Bayesian model estimation with a super-simple synthetic example. A more realistic example — of great practical importance — will be detailed in Section 8.3.

Consider the general statistical modeling task, here discussed for real-valued random variables. A measurement which yields real-valued outcomes (like measuring the speed of a diving falcon) is repeated N times, giving a measurement sequence  $x_1, \ldots, x_N$ . The *i*th measurement value is obtained from a RV  $X_i$ . These RVs  $X_i$ , which model the individual measurements, are i.i.d. We assume that the distribution of each  $X_i$  can be represented by a pdf  $p_{X_i} : \mathbb{R} \to \mathbb{R}^{\geq 0}$ . The i.i.d. property of the family  $(X_i)_{i=1,\ldots,N}$  implies that all these  $p_{X_i}$  are the same, and we call that pdf  $p_X$ . We furthermore assume that  $p_X$  is a parametric pdf, that is, it is a function which is parametrized by a parameter vector  $\theta$ , for which we often write  $p_X(\theta)$ . Then, the statistical modeling / machine learning task is to estimate  $\theta$  from the sample data  $x_1, \ldots, x_N$ . We have seen that this set-up naturally leads to maximum-likelihood estimation algorithms which need to be balanced with regards to bias-variance using some regularization and cross-validation scheme.

For concreteness let us consider a case where N = 2, that is two observations (only two!) have been collected, forming a training sample  $D = (x_1, x_2) =$ (0.9, 1.0). We assume that the pdf  $p_X$  is a normal distribution with unit standard deviation, that is, the pdf has the form  $p_X(x) = 1/\sqrt{2\pi} \exp(-(x-\mu)^2/2)$ . This leaves the expectation  $\mu$  as the only parameter that has to be estimated, thus  $\theta = \mu$ . The learning task is to estimate  $\mu$  from  $D = (x_1, x_2) = (0.9, 1.0)$ .

The classical frequentist answer to this question is to estimate  $\mu$  by the sample mean (which, by the way, is the maximum-likelihood estimator for the expectation  $\mu$ ). That is, one computes

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x_i,$$
(89)

which in our example gives  $\hat{\mu} = (0.9 + 1.0)/2 = 0.95$ .

This is the best a classical-frequentist modeler can do. In a certain well-defined sense which we will not investigate, the sample mean is the optimal estimator for the true mean of a real-valued distribution. But "best" is not "good": with only two data points, this estimate is quite shaky. It has a high variance: if one would repeat the observation experiment, getting a new sample  $x'_1, x'_2$ , chances are very high that one would obtain a quite different sample and thus an estimate  $\hat{\mu}'$  that is quite different from  $\hat{\mu}$ .

Bayesian model estimation shows a way how to do better. It is a systematic method to make use of *prior knowledge* that the modeler may have beforehand. This prior knowledge takes the form of "beliefs" about which parameter vectors  $\theta$ are more or less plausible. This belief is cast in the form of a probability distribution over the space  $\Theta$  of possible parameter vectors  $\theta$ . In our super-simple example let us assume that the modeler knows or believes that the true expectation  $\mu$  can't be negative and it can't be greater than 1. Furthermore, let us assume that the modeler is lacking any more specific insight — all values of  $\theta$  (here:  $\theta = \mu$ ) are equally plausible to the modeler. Thus he/she will encode his/her prior belief as the uniform distribution h over [0, 1]. This distribution is called the *Bayesian* prior distribution.

This kind of prior knowledge is often available, and it can be quite weak (as in our example). Abstracting from our example, this kind of knowledge means to fix a "belief profile" (in the mathematical format of a probability distribution) over the space  $\Theta$  of possible parameters  $\theta$  for a model family. In our mini-example the modeler felt confident to restrict the possible range of the single parameter  $\theta_1 = \mu$ to the interval [0, 1], with a uniform (= non-committing) distribution of "belief" over this interval.

Before I finish the treatment of our baby example, I present the general schema of Bayesian model estimation.

To start the Bayesian model estimation machinery, available prior knowledge about parameters  $\theta$  is cast into the form of a distribution over parameter space. For a K-parametric pdf  $p_X$ , the parameter space is  $\mathbb{R}^K$ . Two comments:

- The distribution for model parameters  $\theta$  is not a distribution in the classical sense. It is not connected to a random variable and does not model a realworld outcome of observations. Instead it captures subjective beliefs that the modeler has about how the true distribution  $P_{X_i}$  of data points *should* look like. It is here that subjectivist aspects of "probability" intrude into an otherwise classical-frequentist picture.
- Each parameter vector  $\theta \in \mathbb{R}^{K}$  corresponds to one specific pdf  $p_{X}(\theta)$  one of the many candidate distributions in  $\Theta$ . A distribution over parameter vectors  $\theta \in \Theta \subseteq \mathbb{R}^{K}$  is thus a distribution over distributions. It is called a *hyperdistribution*. We will use the symbol h to denote this hyperdistribution.

In order to proceed with our discussion of general principles, we need to lift our view from the pdf's  $p_{X_i}(\theta)$ , which model the distribution of single data points, to the the N-dimensional pdf  $p_{\bigotimes X_i} : \mathbb{R}^N \to \mathbb{R}^{\geq 0}$  for the distribution of the product RV  $\bigotimes_i X_i$  (where as usual N is the number of observed data points). Assuming independence of the  $X_i$ , it can be written as

$$p_{\bigotimes X_i}((x_1, \dots, x_N)) = p_{X_1}(x_1) \cdot \dots \cdot p_{X_N}(x_N).$$
(90)

or in another notation (observing that all pdfs  $p_{X_i}$  are identical, so we can use the generic RV X with  $p_X = p_{X_i}$  for all i), as

$$p_{\bigotimes_i X}((x_1, \dots, x_N)) = \prod_i p_X(x_i).$$
(91)

We write  $p_{\bigotimes_i X}(D \mid \theta)$  to denote the pdf value  $p_{\bigotimes_i X}(\theta)((x_1, \dots, x_N)) = p_{\bigotimes_i X}(\theta)(D)$ of  $p_{\bigotimes_i X}(\theta)$  on a particular training data sample  $D = (x_1, \dots, x_N)$ .

Summarizing:

- The pdf  $h(\theta)$  encodes the modeler's prior beliefs about how the parametrized distribution  $p_X(\theta)$  should look like. Parameters  $\theta$  where  $h(\theta)$  is large correspond to data distributions that the modeler a priori finds more plausible. The distribution represented by  $h(\theta)$  is a hyperdistribution; it is often called the (*Bayesian*) prior.
- If  $\theta$  is fixed,  $p_{\bigotimes_i X}(D \mid \theta)$  can be seen as a function of data vectors D. This function is a pdf over the training sample data space. For each possible training sample  $D = (x_1, \ldots, x_N)$  it describes how probable this particular outcome is, assuming the true distribution of X is  $p_X(\theta)$ .

A note on notation: normally the vertical bar denotes conditional probabilities, as in  $P(X = a \mid Y = b)$ . When however a parameter vector  $\theta$  appears right to the bar, such expressions like  $p_{\bigotimes_i X}(D \mid \theta)$  do not denote conditional probabilities. It is just the customary (and admittedly confusing) notation to state that the probability measure in question (here: the pdf  $p_{\bigotimes_i X}$ ) is parametrized by  $\theta$ . We could also equivalently write  $p_{\bigotimes_i X}(\theta)(D)$  as I did earlier, but the notation  $p_{\bigotimes_i X}(D \mid \theta)$  is more common.

If, conversely, D is fixed, then p<sub>⊗i</sub>X(D | θ) can be seen as a function of θ. Seen as a function of θ, p<sub>⊗i</sub>X(D | θ) is not something like a pdf over θ-space. Its integral over θ will not usually be one. Seen as a function of θ, p<sub>⊗i</sub>X(D | θ) is called a *likelihood function* — given data D, it reveals certain models θ as being more likely than others. A model (characterized by θ) "explains" given data D better if p<sub>⊗i</sub>X(D | θ) is higher. We have met the concept of a likelihood function before in Section 7.3.1.

We thus have two sources of information about the sought-after, unknown true distribution  $p_X(\theta)$ : the likelihood  $p_{\bigotimes_i X}(D \mid \theta)$  of  $\theta$  given data D, and the prior plausibility encoded in  $h(\theta)$ . These two sources of information are independent of each other: the prior plausibility is settled by the modeler *before* data have been observed, and should not be informed by data. Because the two sources of information come from "independent" sources of information (belief and data), it makes sense to combine them by multiplication and consider the product

$$p_{\bigotimes_i X}(D \mid \theta) h(\theta).$$

This product combines the two available sources of information about the sought-after true distribution  $p_X(\theta)$ . When data D are given, this product is a function of model candidates  $\theta$ . High values of this product mean that a candidate model  $\theta$  is a good estimate, low values mean it's bad — in the combined light of both observed data and prior beliefs.

With fixed D, the product  $p_{\bigotimes_i X}(D \mid \theta) h(\theta)$  is a non-negative function on the *K*-dimensional parameter space  $\theta \in \mathbb{R}^K$ . It will not in general integrate to unity and thus is not a pdf. Dividing this product by its integral however gives a pdf, which we denote by  $h(\theta \mid D)$ :

$$h(\theta \mid D) = \frac{p_{\bigotimes_{i} X}(D \mid \theta) \ h(\theta)}{\int_{\mathbb{R}^{K}} p_{\bigotimes_{i} X}(D \mid \theta) \ h(\theta) \ d\theta}$$
(92)

The distribution on model parameter space represented by the pdf  $h(\theta|D)$ is called the *posterior distribution* or simply the *posterior*. The formula (92) shows how Bayesians combine the subjective prior  $h(\theta)$  with empirical information  $p_{\bigotimes_i X}(D|\theta)$  to get a posterior distribution over candidate models. Comments:

- The posterior distribution  $h(\theta \mid D)$  is the final result of a Bayesian model estimation procedure. It is a *probability distribution over candidate models*, which is a richer and often a more useful thing than the single model that is the result of a classical frequentist model estimation (like the sample mean from Equation 89).
- Here I have considered real-valued distributions that can be represented by pdfs throughout. If some of the concerned distributions are discrete or cannot be represented by pdfs for some reason, one gets different versions of (92).
- If one wishes to obtain a single, definite model estimate from a Bayesian modeling exercise, a typical procedure is to compute the mean value of the posterior. The resulting model  $\hat{\theta}$  is called the *posterior mean estimate*

$$\hat{\theta} = \theta^{\mathsf{PME}} = \int_{\mathbb{R}^K} \theta \ h(\theta \mid D) \ d\theta.$$

- Compared to classical-frequentist model estimation, generally Bayesian model estimation procedures are computationally more expensive and also more difficult to design properly, because one has to invest some thinking into good priors. With diligently chosen priors, Bayesian model estimates may give *far* better models than classical-frequentist ones, especially when sample sizes are small (really: *far!!* better models, as we will see in the worked-out protein modeling example later).
- If one abbreviates the normalization term  $\int_{\mathbb{R}^K} p_{\bigotimes_i X}(D \mid \theta) h(\theta) d\theta$  in (92) as p(D) ("probability density of seeing data D, averaged over candidate model parameters  $\theta$ "), the formula (92) looks like a version of Bayes' rule:

$$h(\theta \mid D) = \frac{p_{\bigotimes_i X}(D \mid \theta) \ h(\theta)}{p(D)},\tag{93}$$

which is why the this entire approach is called "Bayesian". Note that while the classical textbook Bayes' rule is a *theorem* that can be proven from the axioms of classical probability theory, (93) is a *definition* (of  $h(\theta \mid D)$ ). • A note on terminology. There exists a family of models in machine learning named *Bayesian networks*. This name does not imply that Bayesian networks are estimated from data using "subjectivist" Bayesian statistics. Bayesian networks are parametrized models and their parameter vectors can be estimated with classical frequentist *or* with Bayesian methods.



Figure 50: Bayesian model estimation. Compare text.

Let us conclude this section with a workout of our simple demo example. For the given sample D = (0.9, 1.0), the likelihood function becomes

$$p_{\bigotimes_{i} X}(D \mid \mu) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-(0.9 - \mu)^{2}}{2}\right) \cdot \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-(1.0 - \mu)^{2}}{2}\right)$$
$$= \frac{1}{2\pi} \exp\left(-0.905 + 1.9 \,\mu - \mu^{2}\right)$$

The green line in Figure 50 gives a plot of  $p_{\bigotimes_i X}(D \mid \mu)$ , and the red broken line a plot of the posterior distribution  $h(\mu \mid D)$ . A numerical integration of  $\int_{\mathbb{R}} \mu h(\mu \mid D) d\mu$  yields a posterior mean estimate of  $\hat{\theta} \approx 0.565$ . This is quite different from the sample mean 0.95, revealing the strong influence of the prior distribution on possible models.

### 8.3 Case study: modeling proteins

I will now discuss in some detail an elementary modeling task that is easily stated and met in practical applications all over the place — yet it needs a surprisingly subtle treatment if training data are scarce.

In a general formulation, the task is to estimate a probability mass function for a finite, discrete distribution, given a histogram from a sample. For example, one might want to estimate the probabilities that a coin comes up with head or tail after tossing (formally modeled by a RV X with possible outcome values H and T). Tossing the coin 100 times gives a histogram over the two outcomes H and T, say, (55, 45). Based on these training data, one wants to estimate the probabilities P(X = H) and P(X = T). A maximum likelihood estimate is the pmf  $\hat{p} = (P(X = H), P(X = T))' = (0.55, 0.45)'$ . Easy and obvious.

But things become not-so-easy and non-obvious when the number of symbolic categories is large and the number of available observations is small. I will highlight this with an example taken from a textbook of bioinformatics (Durbin et al. 2000).

Proteins are sequences of amino acids, of which 20 different ones can occur in proteins. They are standardly tagged by 20 symbols  $S = \{A, C, D, ..., Y\}$ , an alphabet that is intimately familiar to biologists. Proteins come in classes. Some protein in one animal or plant species has typically close relatives in other species. Related proteins differ in detail but generally can be *aligned*, that is, corresponding sites in the amino acid string can be detected, put side by side, and compared. For instance, consider the following short section of an alignment of 7 amino acids, all from the class of "globuline" proteins:

- ...GHGK...
- . . . AHGK . . .
- ...KHGV...
- ...THAN...
- ...WHAE...
- ... AHAG...
- ...ALGA...

A basic task in bioinformatics is to estimate the probability distribution of the amino acids in each column in a protein class. These 20-dimensional pmf's (one for each site in the protein) are needed for deciding whether some newly found protein belongs into the class — that is, for carrying out protein classifications, which is a fundamental and bulk-volume task in bioinformatics.

This task is rendered difficult by the fact that the sample of aligned proteins used to estimate these pmf's is typically rather small — here we have only 7 individuals in the sample. As can be seen in the segment of the alignment shown above, some columns have widely varying entries (e.g. the last column has K K V N E G A). In such a small sample, chances are high that an amino acid, which is generally rare at a given site in the animal/plant kingdom, will not appear in the training data set. The column K K V N E G A only features 6 out of 20 possible amino acids; 14 amino acids are missing at this site in the training data.

But the class of related proteins is huge: in every animal species one would expect at least one class member and typically many more — millions of members in the family of globulines! This implies that at every site, it is very likely that every amino acid will occur in *some* species, though maybe with low probability. A low-probability-in-a-specific-site amino acid, call it X, will often not be represented in that site's training data. Consequences:

- 1. a maximum-likelihood estimate of the pmf for that site will assign zero probability to X which in turn implies that,
- 2. when this model is used for protein classification and a new test protein *does* have X in that site, it will turn out that the test protein cannot possibly be a globuline because it has X at that site and in globulines the probability for this has been determined to be zero.

This wrong decision will be made for all test proteins which are globulines but happen to host one of the 14 amino acids not in the training set on that site. Since there are many sites in a protein (dozens to hundreds), chances are high for any test sequence to have a zero-probability amino-acid in some site. Most test sequences which are, in fact, globulines, would be mathematically proven to not be globulines.

Thus, maximum likelihood estimates of pmf's for sites of proteins cannot be used. A Bayesian approach is *mandatory* for this seemingly simple task!

The following lengthy derivations show how the general formula (93) for Bayesian model estimation is worked out in the case of amino acid distribution estimation.

The 20-dimensional pmf for the amino acid distribution at a given site is characterized by a 20-dimensional parameter vector

$$\theta = (\theta_1, \dots, \theta_{20})' = (P(X = \mathbf{A}), \dots, P(X = \mathbf{Y})'$$

(actually 19 parameters would be enough — why?).

We start with the probability of observed data, given that the true distribution is  $\theta$ . This corresponds to the pdf factor  $p_{\bigotimes_i X}(D \mid \theta)$  in (93), but here we have discrete counting data, and hence we use a pmf instead of a pdf. The observation data D consist in counts  $n_1, \ldots, n_{20}$  of the different amino acids found at a given site in the training population. In our column data example K K V N E G A this count vector would have one element with a count value of 2 (the K item), five elements with a value of 1 (for the observations V N E G A in the sample), and the remaining 14 count entries would be zero.

If one considers how such 7-datapoint samples are distributed across different repeated observation experiments (each retrieving 7 observations), these count vectors are distributed according to the *multinomial distribution* 

$$P(D \mid \theta) = \frac{N!}{n_1! \cdots n_{20}!} \prod_{j=1}^{20} \theta_j^{n_j},$$
(94)

where  $N = n_1 + \ldots + n_{20}$  is the total number of observations (in our example N = 7).

Next we turn to the Bayesian prior  $h(\theta)$  in (93). The "subjective" knowledge which is inserted into the picture is the general belief held by biologists that every amino acid *can* appear in any site in a protein of a given class, though maybe with small probability.

The Bayesian prior which reflects the biologists' insight that zero probabilities should not occur in  $\theta$  is expressed in a hyperdistribution over the hypothesis space  $\mathcal{H}$  of possible  $\theta$  vectors.  $\mathcal{H}$  is the subset of  $\mathbb{R}^{20}$  which contains strictly positive vectors whose components sum to one:

$$\mathcal{H} = \{ (\theta_1, \dots, \theta_{20})' \in \mathbb{R}^{20} \mid 0 < \theta_j < 1 \text{ and } \sum_j \theta_j = 1 \}.$$

This is a 19-dimensional hypervolume in  $\mathbb{R}^{20}$  (try to imagine its shape — impossible for  $\mathbb{R}^{20}$ , but you can do it for  $\mathbb{R}^2$  and  $\mathbb{R}^3$ ).

 $\mathcal{H}$  is a continuous space, thus a Bayesian prior distribution on  $\mathcal{H}$  can be represented by a pdf. But which distribution makes a reasonable prior? For reasons that will become clear later, one uses the *Dirichlet* distribution. The Dirichlet distribution is parametrized itself, with (in our case) a 20-dimensional parameter vector  $\alpha = (\alpha_1, \ldots, \alpha_{20})'$  (where all  $\alpha_i$  are positive reals). The Bayesian prior pdf  $h(\theta|\alpha)$  of the Dirichlet distribution with parameters  $\alpha$  is defined as

$$h(\theta|\alpha) = \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1}, \tag{95}$$

where  $Z(\alpha) = \int_{\mathcal{H}} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1} d\theta$  is the normalization constant which ensures that the integral of h over  $\mathcal{H}$  is one.

Fortunately the normalization denominator p(D) in (93) need not be analyzed in more detail because it will later cancel out.

Now we have everything together to calculate the Bayesian posterior distribution on  $\mathcal{H}$ :

$$p(\theta \mid D, \alpha) = \frac{P(D \mid \theta) h(\theta \mid \alpha)}{p(D)}$$

$$= \frac{1}{p(D)} \frac{N!}{n_1! \cdots n_{20}!} \prod_{j=1}^{20} \theta_j^{n_j} \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1}$$

$$= \frac{1}{p(D)} \frac{N!}{n_1! \cdots n_{20}!} \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{n_j + \alpha_j - 1}$$

$$= \frac{1}{p(D)} \frac{N!}{n_1! \cdots n_{20}!} \frac{Z(D + \alpha)}{Z(\alpha)} h(\theta \mid D + \alpha), \quad (96)$$

where  $D + \alpha = (n_1 + \alpha_1, \dots, n_{20} + \alpha_{20})'$ . Because both  $p(\theta | D, \alpha)$  and  $h(\theta | D + \alpha)$  are pdfs, the product of the three first factors in (96) must be one, hence the posterior distribution of models  $\theta$  is

$$p(\theta \mid D, \alpha) = h(\theta \mid D + \alpha).$$
(97)

In order to get the posterior mean estimate, we integrate over the model candidates  $\theta$  with the posterior distribution. I omit the derivation (can be found in Durbin et al. 2000) and only report the result:

$$\theta_j^{\mathsf{PME}} = \int_{\mathcal{H}} \theta_j \ h(\theta \mid D + \alpha) \ d\theta = \frac{n_j + \alpha_j}{N + A},\tag{98}$$

where  $N = n_1 + \cdots + n_{20}$  and  $A = \alpha_1 + \cdots + \alpha_{20}$ . If one compares this to the maximum-likelihood estimates

$$\theta_j^{\mathsf{ML}} = \frac{n_j}{N},\tag{99}$$

we can see that the  $\alpha_j$  parameters of the Dirichlet distribution can be understood as "pseudo-counts" that are added to the actually observed counts. These pseudocounts reflect the *subjective* intuitions of the biologist, and there is no (and can be no) formal rule of how to set them correctly. They are set by the biologist based on professional experience and insight, and the more of such wisdom the biologist has, the better will be the final model.

Adding the  $\alpha_j$  pseudocounts in (98) can also be considered just as a regularization tool in a frequentist setting. One would skip the Bayesian computations that give rise to the Bayesian posterior (97) and directly use (98), optimizing  $\alpha$ to navigate on the model flexibility scale in a cross-validation scheme. With  $\alpha$  set to all zero, one gets the maximum-likelihood estimate (99) which will usually be overfitting; with large values for  $\alpha$  one smoothes out the information contained in the data — underfitting.

You may ask, why go through all this complex Bayesian thinking and calculating, and not just do a regularized maximum-likelihood estimation with a solid cross-validation scheme to get the regularizers  $\alpha_j$ 's right? The reason is that the two methods will not give the same results, although both ultimately use the same formula  $\hat{\theta}_j = \frac{n_j + \alpha_j}{N+A}$ . In the frequentist, regularized, maximum-likelihood approach, the only information that is made use of is the data D, which is scarce and the careful cross-validation will merely make sure that the estimated model  $\hat{\theta} = \theta^{\mathsf{ML}}$ will be the best (with regards to not over- or underfitting) among the ones that can be estimated from D. In contrast, a Bayesian modeler inserts additional information by fixing the  $\alpha_j$ 's beforehand. If the Bayesian modeler has the right intuitions about these  $\alpha_j$ 's, the found model  $\hat{\theta} = \theta^{\mathsf{PME}}$  will generalize better than  $\theta^{\mathsf{ML}}$ , possibly much better.

The textbook of Durbin et al, from which this example is taken, shares some thoughts on to how a biosequence modeler should select the pseudo-counts. The proper investment of such soft knowledge makes all the difference in real-life machine learning problems when data are not abundant.

## 8.4 A brief summary

**The objectivist view** sees probabilities as measurable properties of physical systems. This *frequentist* view is the classical view represented in math-

ematics textbooks. "Frequentist" and "objectivist" are more or less synonymous. Its deep-rooting anchor is the Law of Large Numbers. There is one commonly accepted set of axioms for frequentist probability theory, the Kolmogorov axioms.

- Subjectivist views cast probability as degrees of belief or confidence held by a rational reasoning agent when he/she/it makes statements. Rigorous for-malizations take the shape of logic-like formalisms, of which there are several. Probability is thus a property of statements generalizations of the True and False properties in classical logic —, not a property of the reality described by statements. Such logic system are not usually taught to mathematics students and also not to psychology students in their dreaded statistics courses. Students of philosphy will hear about them (at least they should), and students of AI and linguistics are likely to learn a little about them. The Kolmogorov axioms can be *derived* in most subjectivist-probabilistic logic systems, sometimes in somewhat weaker versions. When it comes to calculation rules, there is thus much agreement between objectivist and subjectivist accounts of probability.
- Bayesian statistics (or Bayesian modeling) can be seen as a very special case of subjectivist probability. Bayesian modeling does not appear in the format of a logic-like calculus. It has developed into an bundle of very useful algorithmic techniques in machine learning. The subjectivist-philosophical heritage has been more or less been lost from sight. The link to the subjectivist philosophical attitude is the way how the modeler's private beliefs are inserted into the model estimation via hyperdistributions the "Bayesian priors". Furthermore, prominent modern approaches in the cognitive and neurosciences posit that human cognitive processing (Clark 2013; Tenenbaum, Griffiths, and Kemp 2006) and even the brain "hardware" (Friston 2003) are organized in functional hierarchies whose processing is informed by Bayesian priors.

# 9 Sampling algorithms

Many tasks that arise in machine learning (and in mathematical modeling of complex systems in general) require one to "draw" random samples from a probability distribution. Algorithms which compute "random" samples from a distribution are needed, for instance (very incomplete listing),

- when *noise* is needed, e.g. for regularization of learning algorithms,
- when *ensembles* of models are computed for better accuracy and generalization (as in random forests),
- generally, when natural or social or economical systems are *simulated*,
- when certain stochastic *neural network* models are trained and exploited (Hopfield networks, Boltzmann machines),
- when training and evaluating many sorts of graphical models, like Bayesian networks (next chapter in these LNs).

Sampling algorithms are an essential "fuel" which powers many modern computational techniques. Entire branches of physics, chemistry, biology (and meteorology and economics and ...) could only grow after efficient sampling algorithms became available.

Designing algorithms which produce (pseudo-)random numbers from a given distribution are not easy to design. Even something that looks as elementary as sampling from the uniform distribution — with pseudo-random number generating algorithms — becomes mathematically and algorithmically involved if one wants to do it well.

In this section I describe a number of design principles for sampling algorithms that you should know about.

# 9.1 What is "sampling"?

"Sampling" means to artificially simulate the process of randomly taking measurements from a distribution given by a pmf or pdf. Figure 51 gives a visual impression.

A "sampler" is an algorithm for doing this — an algorithm that can be run indefinitely long and generates a potentially endless sequence of examples "drawn" from the intended distribution. Samplers there are many, and I am not aware of a universally agreed definition. Here is my own definition (only for the ones of you with probability theory background; will not be asked in exams):



Figure 51: Given: a distribution  $P_X$  of a RV X with sample space  $S_X$ . In our graphics:  $S_X$  is a sector of the real line, and the distribution on  $S_X$  is represented by a pdf (blue line). During a sampling process, a potentially infinite sequence of examples  $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots$  from  $S_X$  is generated. The sampling process must re-generate the "profile" of the pdf in the long run (when  $n \to \infty$ , see panel C). In the short run, the sampling process can, but need not, generate examples by *independent* "draws" from the distribution. In A, it looks like the sampling first put new examples close to already created ones. Both is perfectly ok for a sampling process. The only condition that must be fulfilled is that in the limit of  $n \to \infty$ , the sampled examples pile up in perfect proportion to the pdf.

**Definition 9.1** Let  $P_X$  be a distribution on a measure space  $(E, \mathcal{B})$ . A sequence  $X_1, X_2, \ldots$  of random variables is a sampler for  $P_X$ , if for all  $A \in \mathcal{B}$ 

$$P_X(A) = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^N 1_A \circ X_i \qquad P\text{-almost surely},$$

where  $1_A$  is the indicator function for A.

The notion of a sample must be clearly distinguished from the notion of a sampler. A "sample" is more often than not implied to be an "i.i.d. sample", that is, it results from independent random variables  $X_1, \ldots, X_N$  that all have the same distribution as X, the random variable whose distribution  $P_X$  the sample is drawn from. This i.i.d. assumption is standardly made for real-world training data used in machine learning.

In contrast, the random variables  $X_1, X_2, \ldots$  of a "sampler" for  $P_X$  need not individually have the same distribution as X, and they need not be independent (like in Figure 51B). For instance, let  $P_X$  be the uniform distribution on E = [0, 1]. Here are some samplers:

- All  $X_i$  are i.i.d. with each of them being uniformly distributed on [0, 1]. This is a dream of a sampler. But nobody knows how to build such a sampler without using quantum computers.
- The subset  $X_{2i}$  of RVs with even indices are identically and uniformly distributed on [0, 1]. The RVs  $X_{2i+1}$  repeat the value of  $X_{2i}$ . Here the RVs  $X_i$  are identically but not independently distributed.
- The subset  $X_{2i}$  of RVs with even indices are identically and uniformly distributed on [0, 1/2] and the subset of  $X_{2i+1}$  are identically and uniformly distributed on [1/2, 1]. Here the RVs  $X_i$  are independently but not identically distributed.
- $X_1$  is always evaluating to 0, and the other  $X_i$  are inductively and deterministically defined to give the value  $x_i = x_{i-1} + \sqrt{2} \pmod{1}$ . This is a deterministic sequence that in the long run whill cover the interval [0, 1] uniformly. Here the RVs  $X_i$  are neither independent nor identically distributed.
- Let  $Y_i$  be an i.i.d. sequence where each  $Y_i$  draws a value  $y_i$  from  $\mathcal{N}(0, 1)$ , and let  $\varepsilon$  be a small positive real number.  $X_1$  is always evaluating to 0, and following  $X_i$  are inductively and stochastically defined by  $x_i = x_{i-1} + \varepsilon y_i$  ( mod 1). This gives a random walk (more specifically, a Brownian motion process) whose paths are slowly and randomly migrating across [0, 1]. This kind of samplers will turn out to be the most important type when it comes to sampling from complex distributions — this is a Markov chain Monte Carlo (MCMC) sampler.

# 9.2 Sampling by transformation from the uniform distribution

There is one sampler that you know very well and have often used: the function that creates pseudorandom numbers from the uniform distribution on the unit inverval [0, 1]. It comes by different names in different programming languages (and is internally implemented in many different ways). In Matlab it is called **rand**, and it creates double-precision numbers in [0, 1]. At any rate, a *pseudo-random number* generator of (almost) uniformly distributed numbers over [0, 1] is offered by every programming language that I know, including Microsoft Word. And that's about it; the only sampler you can directly call in most programming environments is just that, a uniform sampler. (Note that the function that is called **rand** in C generates pseudorandom integers from the discrete uniform distribution between 0 and some maximal value. This function can easily be used as a subroutine in a little program which creates pseudorandom numbers in [0, 1]).

It is by no means easy to program a good pseudo-random number generator – in fact, designing such generators is an active field of research. If you are interested – the practical guide to using pseudorandom number generators by Jones 2010 is fun to read and very illuminating.

Assume you have a sampler  $U_i$  for the uniform distribution on [0, 1], but you want to sample from another distribution  $P_X$  on the measure space  $E = \mathbb{R}$ , which has a pdf f(x). Then you can use the sampler  $U_i$  indirectly to sample from  $P_X$  by a coordinate transformation, as follows.

First, compute the cumulative density function  $\varphi : \mathbb{R} \to [0, 1]$ , which is defined by  $\varphi(x) = \int_{-\infty}^{x} f(u) du$ , and its inverse  $\varphi^{-1}$ . The latter may be tricky or impossible to do analytically – then numerical approximations must be used. Now obtain a sampler  $X_i$  for  $P_X$  from the uniform sampler  $U_i$  by

$$X_i = \varphi^{-1} \circ U_i.$$

Figure 52 illustrates visually why this works.



Figure 52: Sampling by transformation from the uniform distribution: Let [a, b] be an interval on the sample space  $\mathbb{R}$ , and A the area under the pdf f in this interval. A sampler for the distribution given by the pdf f must visit [a, b] with a probability A. But A is also the increment that the cumulative density  $\varphi(x)$  builds up from a to b. The uniform sampler  $U_i$  for [0, 1] visits the segment A (right end in the graphic) with probability A. When  $U_i$  picks a value from the segment A,  $X_i = \varphi^{-1} \circ U_i$  will give a value in the interval [a, b].

The remainder of this subsection is optional reading and will not be tested in exams (but may turn up in the exercise sheet).

Out of curiosity, I explored a little bit how one can sample from the standard normal distribution  $\mathcal{N}(0,1)$ . Computing it by transformation from the uniform distribution, as outlined above, appears not to be done because the inverse cumulative density function  $\varphi^{-1}$  for  $\mathcal{N}(0,1)$  can only be represented through a power series which converges slowly (my source, I must confess, is Wikipedia). Instead, special algorithms have been invented for sampling from  $\mathcal{N}(0,1)$ , which exploit mathematical properties of this distribution. One algorithm (see http://en.wikipedia. org/wiki/Normal\_distribution#Generating\_values\_for\_normal\_random\_variables ) which I found very elegant is the *Box-Muller algorithm*, which produces a  $\mathcal{N}(0, 1)$ distributed RV C from *two* independent uniform-[0,1]-distributed RVs A and B:

$$C = \sqrt{-2 \ln A} \, \cos(2 \pi B).$$

The computational cost is dominated by computing the logarithm and the cosine. An even (much) faster, but more involved (and very tricky) method is the *Ziggurat algorithm* (check Wikipedia for "Ziggurat\_algorithm" for an article written with tender care). There exist wonderful things under the sun.

Sampling by coordinate transformation can be generalized to higher-dimensional distributions. Here is the case of a 2-dimensional pdf  $f(x_1, x_2)$ , from which the general pattern should become clear. First define the cumulative density function in the first dimension as the cumulative density function of the marginal distribution of the first coordinate  $x_1$ 

$$\varphi_1(x_1) = \int_{-\infty}^{x_1} \left( \int_{-\infty}^{\infty} f(u, v) \, dv \right) \, du$$

and the conditional cumulative density function on  $x_2$  given  $x_1$ 

$$\varphi_2(x_2 \mid x_1) = \frac{\int_{-\infty}^{x_2} f(x_1, v) \, dv}{\int_{-\infty}^{\infty} f(x_1, v) \, dv}$$

Then, for sampling one value  $(x_1, x_2)$ , sample two values  $u_1, u_2$  from the uniform sampler U, and transform

$$x_1 = \varphi_1^{-1}(u_1), \quad x_2 = \varphi_2^{-1}(u_2 \mid x_1).$$

A widely used method for drawing a random vector  $\mathbf{x}$  from the *n*-dimensional multivariate normal distribution with mean vector  $\boldsymbol{\mu}$  and covariance matrix  $\boldsymbol{\Sigma}$  works as follows:

- 1. Compute the Cholesky decomposition (matrix square root) of  $\Sigma$ , that is, find the unique lower triangular matrix A such that  $AA' = \Sigma$ .
- 2. Sample *n* numbers  $z_1, \ldots, z_n$  from the standard normal distribution.
- 3. Output  $\mathbf{x} = \mu + A(z_1, \dots, z_n)'$ .

## 9.3 Rejection sampling

Sampling by transformation from the uniform distribution can be difficult or impossible if the pdf f one wishes to sample from has no cumulative density function with a simple-to-compute inverse. If this happens, it is sometimes possible to sample from a simpler distribution with pdf g that is loosely related to the target pdf f

and for which sampling by transformation works, and with a simple trick get from that a sampler for the pdf of interest, f. To understand this *rejection sampling* (also known as *importance sampling*) we first need to generalize the notion of a pdf:

**Definition 9.2** A proto-pdf on  $\mathbb{R}^n$  is any nonnegative function  $g_0 : \mathbb{R}^n \to \mathbb{R}$  with a finite integral  $\int_{\mathbb{R}^n} g_0(\mathbf{x}) d\mathbf{x}$ .

If one divides a proto-pdf  $g_0$  by its integral, one obtains a pdf g. Now consider a pdf f on  $\mathbb{R}^n$  that you want to sample from, but sampling by transformation doesn't work. However, you find a proto-pdf  $g_0 \geq f$  for whose associated pdf gyou know how to sample from, that is, you have a sampler for g. With that in your hands you can construct a sampler  $X_i$  for f as follows. In order to generate a random value x for  $X_i$ , carry out the following procedure:

Initialize: set found\_x = 0

While found\_x == 0 do

- 1. sample a *candidate*  $\tilde{x}$  from g.
- 2. with probability  $f(\tilde{x})/g_0(\tilde{x})$ , accept  $\tilde{x}$ ; otherwise, reject  $\tilde{x}$ .
  - If  $\tilde{x}$  is accepted

(a) set found\_x = 1

(b) return  $\tilde{x}$ 

Rejection sampling becomes immediately clear if you re-conceptualize "sampling from a pdf f" as "piling up the pdf by many grains of sand that you let fall at the various positions x of  $\mathbb{R}^n$  with a frequency proportional to f(x)". Look at Figure 53 where  $g_0$  and f are plotted. Think of the  $g_0$  curve as a sand dune that you get when sampling for g in this sand metaphor. Now, when you drop a grain of sand for modelling the  $g_0$  "sand curve", imagine you paint it orange with a probability of  $f(\tilde{x})/g_0(\tilde{x})$  before you let it fall down. I think this saves you a mathematical proof.



Figure 53: The principle of rejection sampling. Candidates  $\tilde{x}$  are first sampled from g, then accepted ("painted orange") with probability  $f(\tilde{x})/g_0(\tilde{x})$ .

The computational efficiency of rejection sampling clearly depends on how close  $g_0$  is to f. If the ratio  $f/g_0$  is on average small, there will be many rejections which slow down the algorithm. In high-dimensional spaces it is often quite difficult to avoid this.

### 9.4 Proto-distributions

The pdf's of parametrized continuous distributions over  $\mathbb{R}^k$  are typically represented by a formula of the kind

$$p(\mathbf{x} \mid \theta) = \frac{1}{\int_{\mathbb{R}^k} p_0(\mathbf{x} \mid \theta) \, d\mathbf{x}} \, p_0(\mathbf{x} \mid \theta), \tag{100}$$

where  $p_0 : \mathbb{R}^k \to \mathbb{R}^{\geq 0}$  defines the shape of the pdf and the factor  $1/(\int_{\mathbb{R}^k} p_0(\mathbf{x} \mid \theta) d\mathbf{x})$  normalizes  $p_0$  to integrate to 1. For example, in the pdf

$$p(\mathbf{x} \mid \mu, \Sigma) = \frac{1}{(2\pi)^{k/2} \det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)' \Sigma^{-1}(\mathbf{x} - \mu)\right)$$

for the multidimensional normal distribution, the shape is given by  $p_0(\mathbf{x}|\mu, \Sigma) = \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)' \Sigma^{-1}(\mathbf{x}-\mu)\right)$  and the normalization prefactor  $1/((2\pi)^{k/2} \det(\Sigma)^{1/2})$  is equal to  $1/\int_{\mathbb{R}^k} p_0(\mathbf{x} \mid \mu, \Sigma) d\mathbf{x}$ .

It is a quite general situation in machine learning that a complex model of a distribution is only known up to an undetermined scaling factor — that is, one has a proto-distribution but not a distribution proper. This is often good enough. For instance, in our initial TICS demo (Section 1.2.1), we saw that the TICS system generated a list of captions, ordered by plausibility. In order to produce this list, the TICS system did not need to compute a proper probability distribution on the space of possible captions — a proto-distribution is good enough to determine the best, second-best, etc. caption.

Proto-distributions are not confined to continuous distributions on  $\mathbb{R}^k$ , where they appear as proto-pdf's. In discrete sample spaces  $S = \{s_1, \ldots, s_N\}$  or  $S = \{s_1, s_2, \ldots,\}$  where the distribution is given by a parametrized pmf  $p(\theta)$ , the normalization factor becomes a sum over S:

$$p(x \mid \theta) = \frac{1}{\sum_{s \in S} p_0(s)} p_0(x),$$

where again  $p_0 : S \to \mathbb{R}^{\geq 0}$  is a shape-giving function which must have a finite sum. When S is large and  $p_0$  is analytically intractable, the normalization factor  $1/\sum_{s\in S} p_0(s)$  cannot be computed and one is limited to work with the proto-pmf  $p_0$ only. Very large sample spaces S appear standardly in systems which are modeled by many interacting random variables. We will meet with such proto-pmf's in the next section on graphical models. This is also a good moment to mention the *softmax* function. This is a ubiquitously used machine learning trick to transform any vector  $\mathbf{y} = (y_1, \ldots, y_n)' \in \mathbb{R}^n$ into an *n*-dimensional probability vector  $\mathbf{p}$  by

$$\mathbf{p} = \frac{1}{\sum_{i=1,\dots,n} \exp(\alpha \, y_i)} \, (\exp(\alpha \, y_1), \dots, \exp(\alpha \, y_n))'.$$

The softmax is standardly used, for instance, to transform the output vector of a neural network into a probability vector, which is needed when one wants to interpret the network output probabilistically.

The factor  $\alpha \geq 0$  determines the entropy of the resulting probability vector. If  $\alpha = 0$ , **p** is the uniform distribution on  $\{1, \ldots, n\}$ , and in the limit  $\alpha \to \infty$ , **p** becomes the binary vector which is zero everywhere except at the position *i* where **y** is maximal.

Notes:

- 1. The word "proto-distribution" is not in general use and can be found in the literature only rarely.
- 2. Proto-distributions occur always in Bayesian modeling (compare Section 8.3, Equation 96). The Bayesian posterior distribution  $p(\theta | D)$  is shaped by the product  $P(D | \theta) h(\theta)$ , whose integral will not usually be one. The functions  $P(D | \theta) h(\theta)$  are proto-distributions on  $\theta$ -space.
- 3. Many algorithms exist that allow sampling from proto-distributions, without the need of normalizing them. Rejection sampling is a point in case, but also the Gibbs and Metropolis sampler presented below work with proto-pdfs. The case study in Section 9.6 crucially relies on sampling from a proto-pdf.
- 4. There exists a large and important class of probability distributions on arbitrary sample spaces S, which are defined by an *energy function*  $E: S \to \mathbb{R}^{\geq 0}$ , or more generally, by a *cost function*  $C: S \to \mathbb{R}$ . In *energy-based models* of probability distributions, such an energy or cost function is turned into a probability distribution on S by defining the *Boltzmann distribution* which has the pdf

$$p(s \mid T) = \frac{1}{\int_{S} \exp(-C(s)/T) \, ds} \, \exp(-C(s)/T), \tag{101}$$

where T is a temperature parameter. In this context, the normalization factor  $\frac{1}{\int_{S} \exp(-C(s)/T) ds}$  is often written as 1/Z(T), and Z(T) is called the partition function of the distribution. The notation  $1/Z(\theta)$  and the name "partition function" are sometimes also used in a generalized fashion for other distributions given by a shape-defining proto-pdf. Boltzmann distributions are imported from statistical physics and thermodynamics, where

they play a leading role. In machine learning, Boltzmann distributions occur, for instance, in *Markov random fields*. The are the spatial analogue to the temporal Markov chains; besides many applications in theoretical physics, in computer science they are used in image analysis. Boltzmann distributions are a core ingredient for *Boltzmann machines* (Ackley, Hinton, and Sejnowski 1985), a type of neural network which I consider one of the most elegant and fundamental models for general learning systems (see Section 7 in my lecture notes for 'Neural Networks'). A structurally and computationally streamlined version called *restricted Boltzmann machines* (RBMs), became a starting point for the deep learning revolution (Hinton and Salakuthdinov 2006). Furthermore, Boltzmann distributions also are instrumental for *free-energy models* of intelligent agents, a class of models of brain dynamics popularized by Karl Friston (for example, Friston 2005) which explain the emergence of adaptive, hierarchical information representations in biological brains and artificial intelligent "agents", and which today are a mainstream approach in the cognitive neurosciences and cognitive science to understanding adaptive cognition. And last but not least, the Boltzmann distribution is the root mechanism in simulated annealing (Kirkpatrick, Gelatt, and Vecchi 1983), a major general-purpose strategy for finding minimal-cost solutions in complex search spaces.

Sadly in this course there is no time for treating energy-based models of information processing. I present this material in my course on neural networks in the Summer semester.

# 9.5 MCMC sampling

We now turn to Markov Chain Monte Carlo (MCMC) sampling techniques. These samplers are the only choice when one wants to sample from complex distributions of which one only knows a proto-pdf  $g_0$ . Inventing MCMC sampling techniques was a breakthrough in theoretical physics because it made it possible to simulate stochastic physical systems on computers. The original article Equation of state calculations by fast computing machines (Metropolis et al. 1953) marks a turning point in physics (Wikipedia has an article devoted to this algorithm, check out under Metropolis-Hastings algorithm). This work was carried out within a large-scale effort in the U.S. to develop the nuclear fission bomb... makes you think. In later decades, MCMC techniques were adopted in machine learning, where today they are used in several places, in particular for working with graphical models, which constitute the high-end modeling approach for complex real-world systems that are described by a multitude of interacting RVs. We will get a glimpse of graphical models in the next session of this course. A long tutorial paper (essentially, the PhD thesis) of Radford Neal 1993 was instrumental for establishing MCMC methods for statistical inference applications. This tutorial paper established not only MCMC methods in machine learning, but also established the professional

reputation of Radford Neal (http://www.cs.utoronto.ca/~radford/). If you ever want to implement MCMC samplers in a serious way, this tutorial is mandatory reading. And if you want to inform yourself about the use of MCMC in deep learning (advanced stuff), a talk of Radford Neal that he recently delivered in honor of Geoffrey Hinton (Neal 2019) gives you an orientation. And if you think of entering a spectacular academic career, it is not a bad idea to write a PhD thesis which "only" gives the first readable, coherent, comprehensive tutorial introduction to some already existing kind of mathematical modeling method which, until you write that tutorial, has been documented only in scattered technical papers written from different perspectives and using different notations, making that knowledge inaccessible to a wider audience.

#### 9.5.1 Markov chains

Before we can start with MCMC we need to refresh the basics of Markov chains.

**Definition 9.3** A Markov chain is a sequence of random variables  $X_1, X_2, \ldots$ , all taking values in the same sample space S, such that  $X_{n+1}$  depends only on  $X_n$ , *i.e. it is conditionally independent on all earlier values:* 

$$P(X_{n+1} = x_{n+1} | X_1 = x_1, \dots, X_n = x_n) = P(X_{n+1} = x_{n+1} | X_n = x_n).$$
(102)

Intuitively, a Markov chain is a discrete-time stochastic process which generates sequences of measurements/observations, where the next observation only depends on the current one; earlier observations are "forgotten". Markov chains are "stochastic processes without memory".

Do not confound the notion of a Markov chain with the notion of a *path* (or *realization* or *trajectory*) of a Markov chain. The Markov chain is the generative mechanism — in mathematical terminology: the *process* — which can be "executed" or "run" as often as one wishes, and in every run one will obtain a different concrete sequence  $x_1, x_2, \ldots$  of observations. The possible values  $x \in S$  are called the *states* of a Markov chain.

The times n for which a Markov process is defined can be

- from a finite interval  $[0, 1, ..., n_{max}]$ , typically started at n = 0 we then speak of a finite Markov chain,
- right-infinite: n = 0, 1, 2, ..., or
- left-right infinite:  $n \in \mathbb{Z}$ .

For defining samplers, on has to use  $n \in \mathbb{N}$ : a sampler must, by definition, be able to be run for arbitrarily long times.

A Markov chain on S is fully characterized by the *initial distribution*  $P_{X_0}$  (needed for starting finite and right-infinite chains but not needed for left-right infinite chains which have no start) and the conditional *transition probabilities* 

$$P_n(X_{n+1} = x \mid X_n = y) \quad \text{for all } x, y \in S, \tag{103}$$

which taken all together (for all combinations of x, y values) specify the conditional transition distribution - a mathematical object for which we also write (following Neal's notation)

$$T_n(x \mid y). \tag{104}$$

A common name for  $T_n(x | y)$  is transition kernel.  $T_n$  is a function of two arguments x, y and returns  $T_n(x | y) = P_n(X_{n+1} = x | X_n = y)$ . The transition kernel  $T_n(x | y)$  thus informs us by which probability we will observe x at the next time step n + 1 given that at the current time n we observed y.

If  $T_n(x | y) = T_{n'}(x | y)$  for all n, n', the Markov chain's transition law does not itself change with time; we then call the Markov chain *homogeneous*. With homogeneous Markov chains, the index n can be dropped from  $T_n(x|y)$  and we write T(x | y).

In order to generate a path from a homogeneous Markov chain with initial distribution  $P_{X_0}$  and transition kernel T(x | y), one carries out the following iterative procedure:

- 1. Generate the first value  $x_0$  by a random draw from  $P_{X_0}$ .
- 2. If  $x_n$  has been generated, generate  $x_{n+1}$  by a random draw from the distribution  $P_{X_{n+1}|X_n=x_n}$  which is specified by the transition kernel  $T(x \mid x_n)$ .

A note on notation: a mathematically correct and general definition and notation for transition kernels on arbitrary observation spaces S requires tools from measure theory which we haven't introduced. Consider the notation T(x | y) as a somewhat sloppy shorthand. When dealing with discrete distributions where S is finite, say,  $S = \{s_1, \ldots, s_k\}$ , then consider T(x | y) as a  $k \times k$  Markov transition matrix M where  $M(i, j) = P(X_{n+1} = s_j | X_n = s_i)$ . The *i*-th row of M has the vector of the probabilities by which the process will transit from state  $s_j$  to the states  $s_1, \ldots, s_k$  indexing the columns.

An example: consider the two-state observation space  $S = \{a, b\}$ . Then the initial distribution  $P_{X_0}$  is given by a 2-dimensional probability vector. For instance, (0.3, 0.7)' would mean that the process starts with an outcome  $X_0 = a$  with probability 0.3. A transition matrix might be

$$M = \left(\begin{array}{cc} 0.1 & 0.9\\ 0.4 & 0.6 \end{array}\right),$$

whose first row means that if at time n the process has generated state a, at time n + 1 one will observe a again with probability 0.1 and b with probability 0.9.

When dealing with continuous distributions of next states x (given y) which have a pdf, regard T(x | y) as denoting the conditional pdf  $p_{X|Y=y}$  of x. Note that when we write T(x | y), we refer not to a single pdf but to a family of pdf's; for every y we have another conditional pdf T(x | y). When a homogeneous Markov chain with finite state set  $S = \{s_1, \ldots, s_k\}$  and Markov transition matrix M is executed m times, the probabilities to transit from state  $s_i$  to state  $s_j$  after m steps can be found in the m-step transition matrix  $M^m$ :

$$P(X_{n+m} = s_j | X_n = s_i) = M^m(i, j),$$
(105)

where  $M^m = M \cdot M \cdot \ldots \cdot M$  (*m* times).

We now consider the sample space  $S = \mathbb{R}^k$  — this is the state space needed for most practical uses of MCMC sampling — and we assume that all distributions of interest are specified by pdf's. We consider a homogeneous Markov chain. Its transition kernel  $T(\mathbf{x}|\mathbf{y})$  can be identified with the pdf  $p_{X_{n+1}|X_n=\mathbf{y}}$ , and in the remainder of this section, we will write  $T(\mathbf{x}|\mathbf{y})$  for this pdf. Such a Markov chain with continuous sample space  $\mathbb{R}^k$  is specified by an initial distribution on  $\mathbb{R}^k$  which we denote by its pdf  $g^{(0)}$ . The pdf's  $g^{(n+1)}$  of distributions of subsequent RVs  $X_{n+1}$ can be calculated from the pdf's  $g^{(n)}$  of the preceding RV  $X_n$  by

$$g^{(n+1)}(\mathbf{x}) = \int_{\mathbb{R}^k} T(\mathbf{x} \mid \mathbf{y}) \ g^{(n)}(\mathbf{y}) \ d\mathbf{y}.$$
 (106)

 $g^{(n)}$  is the distribution over the process's state space S that we get if we would run the entire sequence-generating process over and over again and at time n look at the which states have been visited how often on average. Please make sure you understand this equation. It is the key to everything which follows.

For the theory of MCMC sampling, a core concept is an *invariant distribution* of a homogenous Markov chain.

**Definition 9.4** Let g be the pdf of some distribution on  $\mathbb{R}^k$ , and let  $T(\mathbf{x} | \mathbf{y})$  be the (pdf of the) transition kernel of a homogeneous Markov chain with values in  $\mathbb{R}^k$ . Then g is the pdf of an invariant distribution of  $T(\mathbf{x} | \mathbf{y})$  if

$$g(\mathbf{x}) = \int_{\mathbb{R}^k} T(\mathbf{x} \,|\, \mathbf{y}) \, g(\mathbf{y}) \, d\mathbf{y}.$$
(107)

Except for certain pathological cases, a transition kernel will have at least one invariant distribution.

Furthermore, it is often the case that there exists exactly one invariant distribution g of  $T(\mathbf{x} | \mathbf{y})$ , and the sequence of distributions  $g^{(n)}$  converges to g from any initial distribution. We will call the transition kernel  $T(\mathbf{x} | \mathbf{y})$  ergodic if it has this property. The (unique) invariant distribution g of an ergodic Markov chain is also called its asymptotic distribution or its stationary distribution or its equilibrium distribution.

#### 9.5.2 Constructing MCMC samplers through detailed balance

Let g be the pdf of some (usually very complex) distribution on  $\mathbb{R}^k$ . The goal is to design an MCMC sampler for g. The core idea is to find a transition kernel

 $T(\mathbf{x} | \mathbf{y})$  for a homogenous, ergodic Markov chain which has g as its invariant distribution. Then, by certain deep mathematical *ergodic theorems*, which we will not endeavour to understand, the Markov chain will be a sampler for g.

There are many ways to design such transition kernels. However, they will differ from each other with regards to computational cost. Quoting from Neal's survey,

"The amount of computational effort required to produce a good Monte Carlo estimate using the states generated by a Markov chain will depend on three factors: first, the amount of computation required to simulate each transition; second, the time for the chain to converge to the equilibrium distribution, which gives the number of states that must be discarded from the beginning of the chain; third, the number of transitions needed to move from one state drawn from the equilibrium distribution to another state that is almost independent, which determines the number of states taken from the chain at equilibrium that are needed to produce an estimate of a given accuracy. The latter two factors are related..."

A brief explanation: The best possible sampler for g would be one where each new sampled value is independent from the previous ones — in other words, one would like to have an i.i.d. sampler (Figure 54 **a**.) This is usually impossible to get. If, more realistically, observations  $\mathbf{x}_n$  obtained from a sampler depend on previous observations  $\mathbf{x}_{n-1}, \mathbf{x}_{n-2}, \ldots$ , there is redundant information in the sample path. Typically paths  $(\mathbf{x}_n)_{n=0,1,2,\ldots}$  obtained from running an MCMC sampler will have more or less strong dependencies between values observed at nearby times. The first values  $\mathbf{x}_1, \mathbf{x}_2, \ldots$  values will depend, to a decreasing degree, on the arbitrary initial value  $\mathbf{x}_0$  and should be discarded. After this initial "washout" phase, one usually keeps only those values  $\mathbf{x}_n, \mathbf{x}_{n+d}, \mathbf{x}_{n+2d}, \ldots$  whose distance d from each other is large enough to warrant that the dependency of  $\mathbf{x}_{n+d}$  on  $\mathbf{x}_n$  has washed out to a negligible amount, that is,  $P(X_{n+d} = \mathbf{x} | X_n = \mathbf{y}) \approx P(X_{n+d} = \mathbf{x})$ . Figure 54 **b**. illustrates the 'look and feel' of a typcial MCMC sampler.

A standard way to construct an MCMC transition kernel  $T(\mathbf{x} | \mathbf{y})$  which leads to a Markov chain that has the target distribution g as its invariant distribution is to ensure that the Markov chain  $(X_n)_{n=0,1,\dots}$  has the property of *detailed balance* with respect to g. Detailed balance connects  $X_0, X_1, X_2, \dots$  to g in a strong way. It says that if we pick some state  $\mathbf{x} \in \mathbb{R}^k$  with the probability given by g and multiply its probability  $g(\mathbf{x})$  with the transition probability density  $T(\mathbf{y}|\mathbf{x})$  that is, we consider the probability density of transiting from  $\mathbf{x}$  to  $\mathbf{y}$  weighted with the probability density of  $\mathbf{x}$  — then this is the same as the reverse weighted transiting probability density from  $\mathbf{y}$  to  $\mathbf{x}$ :

**Definition 9.5** Let g be a pdf on  $\mathbb{R}^k$  and let  $T(\mathbf{y}|\mathbf{x})$  be a transition kernel of a homogeneous Markov chain on  $\mathbb{R}^k$ . Then  $T(\mathbf{y}|\mathbf{x})$  has the detailed balance property with respect to g if

$$\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^k : \ T(\mathbf{y}|\mathbf{x}) \ g(\mathbf{x}) = T(\mathbf{x}|\mathbf{y}) \ g(\mathbf{y}).$$
(108)



Figure 54: Sampling from a 2-dimensional normal distribution (contour lines show pdf) with two different samplers. The red dot shows the first state that was generated. **a.** 200 sample points were obtained with i.i.d. sampling directly from the target distribution. This is possible for Gaussians (but i.i.d. samplers are not generally obtainable for other distributions). **b.** An MCMC sampler generating the first 200 points of a potentially much longer sampling sequence. It becomes apparent that the respective next sample points here are usually close to the previous one, leading to a 'slow crawl' behavior that covers the entire target distribution only after long runtimes.

If  $T(\mathbf{x}|\mathbf{y})$  has detailed balance with respect to g, then g is an invariant distribution of the Markov chain given by  $T(\mathbf{x}|\mathbf{y})$  because

$$\int_{\mathbb{R}^k} T(\mathbf{x}|\mathbf{y}) \ g(\mathbf{y}) \ d\mathbf{y} = \int_{\mathbb{R}^k} T(\mathbf{y}|\mathbf{x}) \ g(\mathbf{x}) \ d\mathbf{y} = g(\mathbf{x}) \ \int_{\mathbb{R}^k} T(\mathbf{y}|\mathbf{x}) \ d\mathbf{y} = g(\mathbf{x})$$

Detailed balance is sufficient but not necessary for a Markov chain to be a sampler for g. However, there are convenient recipies to design MCMC's based on detailed balance. I present the two most famous and most widely used: the *Gibbs* sampler and the *Metropolis algorithm*.

#### 9.5.3 The Gibbs sampler

Let g be a pdf on  $\mathbb{R}^k$ , from which we want to sample. For i = 1, ..., k and  $\mathbf{x} \in \mathbb{R}^k$ , where  $\mathbf{x} = (x_1, ..., x_k)'$ , let

$$g_{i}(\cdot | \mathbf{x}) : \mathbb{R} \to \mathbb{R}^{\geq 0}$$
  

$$g_{i}(y | \mathbf{x}) = \frac{g((x_{1}, \dots, x_{i-1}, y, x_{i+1}, \dots, x_{k})')}{\int_{\mathbb{R}} g((x_{1}, \dots, x_{i-1}, z, x_{i+1}, \dots, x_{k})') dz}$$

be the conditional density function of the coordinate *i* given the values of **x** on the other coordinates. Let  $g^{(0)}$  be the pdf of an initial distribution on  $\mathbb{R}^k$ , and let an initial value  $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_k^{(0)})'$  be drawn from  $g^{(0)}$  (this corresponds to drawing an example with  $X_0$ ). We define a Markov chain  $X_1, X_2, \ldots$  through transition kernels as follows. The idea is to cycle through the k coordinates and at some time  $\nu k + i$  ( $0 \le \nu, 1 \le i \le k$ ) change the previous state  $\mathbf{x}^{(\nu k+i-1)} = (x_1^{(\nu k+i-1)}, \ldots, x_k^{(\nu k+i-1)})'$  only in the *i*-th coordinate, by sampling from  $g_i$ . That is, at time  $\nu k + i$  we set

$$\mathbf{x}^{(\nu k+i)} = (x_1^{(\nu k+i-1)}, \dots, x_{i-1}^{(\nu k+i-1)}, y, x_{i+1}^{(\nu k+i-1)}, \dots, x_k^{(\nu k+i-1)})'$$

equal to the previous state except in coordinate i where the value y which is freshly sampled from  $g_i$ .

This method is known as the *Gibbs sampler*. It uses k different transition kernels  $T_1, \ldots, T_k$ , where  $T_i$  is employed at times  $\nu k + i$  and updates only the *i*-th coordinate. Of course, constructing such a Gibbs sampler requires one to have i.i.d. samplers for the 1-dimensional coordinate distributions  $g_i$  in the first place. This is, however, a simpler problem than the original multi-dimensional one.

This Markov chain is not homogeneous because we cycle through different transition kernels. However, we can condense a sequence of k successive updates into a single update that affects all coordinates by putting  $T = T_k \circ \cdots \circ T_1$ , which yields a homogeneous Markov chain  $(Y_n)_{n=1,2,\ldots}$  with transition kernel T whose path is derived from a path  $(\mathbf{x}_n)_{n=1,2,\ldots}$  of the "cycling" Markov chain by

$$\mathbf{y}^{(1)} = \mathbf{x}^{(1)},$$
  
 $\mathbf{y}^{(2)} = \mathbf{x}^{(k+1)},$   
 $\mathbf{y}^{(3)} = \mathbf{x}^{(2k+1)}, \dots$ 

It should be clear that g is an invariant distribution of this homogeneous Markov chain, because each transition kernel  $T_i$  leaves g invariant: only the *i*th component of the observation vector is affected at all, and it is updated exactly according to the conditional distribution  $g_i$  for this component. It even holds that T has detailed balance w.r.t. g (exercise — do it!).

It remains to ascertain that the Markov chain with kernel T is ergodic. This is certainly the case when the support of g in  $\mathbb{R}^k$  (that is, the subset of  $\mathbb{R}^k$  where g is positive) is k-dimensionally connected. This means that for any points  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$ with  $g(\mathbf{x}) > 0, g(\mathbf{y}) > 0$  there exists a finite sequence of k-dimensional open balls  $B_m$  of positive radius, the first containing  $\mathbf{x}$  and the last containing  $\mathbf{y}$ , such that  $B_m$  intersects with  $B_{m+1}$ . However, if the support of g is not k-dimensionally connected, then T may or may not be ergodic, and determining whether it is needs to be done on a case by case basis. For instance, if g is a distribution on  $\mathbb{R}^2$  whose support lies exclusively in the first and third orthant, T would not be ergodic, because the Gibbs sampler, when started from a point in the third orthant, would be unable to jump into the first orthant. This situation is depicted in Figure 55.

The Gibbs sampler is practically applicable only if one can easily sample from the 1-dimensional conditional distributions  $g_i$ . Therefore, the Gibbs sampler is



Figure 55: A bipartite pdf g where the Gibbs sampler would fail.

mostly employed in cases where these  $g_i$  are parametric, analytical distributions, or in cases where S is finite and the  $g_i$  thus become simple probability vectors (and would be represented by pmfs, not pdfs). The Gibbs sampler is attractive for its simplicity. A number of extensions and refinements of the basic idea is presented in Neal 1993.

### 9.5.4 The Metropolis algorithm

The Metropolis algorithm is more sophisticated than the Gibbs sampler and works in a much larger number of cases. The drawback is that it may be computationally more expensive. This algorithm explicitly constructs a Markov transition kernel with detailed balance. Like the Gibbs algorithm, the Metropolis algorithm can be cyclically applied to the k dimensions of the observation vectors in turn ("local" Metropolis algorithm) or it can be applied to all dimensions simultaneously ("global" algorithm). I describe the local version.

Like the Gibbs sampler, the local Metropolis algorithm updates the sample vector  $\mathbf{x}^{(\nu k+i-1)} = (x_1^{(\nu k+i-1)}, \dots, x_k^{(\nu k+i-1)})'$  only in the *i*-th coordinate, yielding an update

$$\mathbf{x}^{(\nu k+i)} = (x_1^{(\nu k+i-1)}, \dots, x_{i-1}^{(\nu k+i-1)}, y, x_{i+1}^{(\nu k+i-1)}, \dots, x_k^{(\nu k+i-1)})'$$

in two substeps, which together ensure detailed balance w.r.t. the conditional distribution  $g_i$ :

**Step 1:** Randomly choose a *candidate* value  $y^*$  for y. Usually a *proposal distribution*  $S_i(y^* | \mathbf{x}^{(\nu k+i-1)})$  is used (it may depend on  $\mathbf{x}^{(\nu k+i-1)}$ ), which is symmetric in the sense that

$$S_{i}(y \mid (x_{1}^{(\nu k+i-1)}, \dots, x_{i-1}^{(\nu k+i-1)}, y', x_{i+1}^{(\nu k+i-1)}, \dots, x_{k}^{(\nu k+i-1)})' = S_{i}(y' \mid (x_{1}^{(\nu k+i-1)}, \dots, x_{i-1}^{(\nu k+i-1)}, y, x_{i+1}^{(\nu k+i-1)}, \dots, x_{k}^{(\nu k+i-1)})'$$

for all  $y, y', \mathbf{x}$ .

**Step 2:** Randomly *accept* or *reject*  $y^*$  as the new value for  $x_i^{(\nu k+i)}$ , in a fashion that ensures detailed balance. In case of acceptance,  $x_i^{(\nu k+i)}$  is set to  $y^*$ ;

in case of rejection,  $x_i^{(\nu k+i)}$  is set to  $x_i^{(\nu k+i-1)}$ , i.e., the observation vector is not changed at all and is identically repeated in the sampling sequence. The probability for accepting  $y^*$  is determined by an *acceptance probability*  $A_i(y^*|\mathbf{x})$ .

Note that  $S_i(y^* | \mathbf{x}^{(\nu k+i-1)})$  is a probability distribution on the  $x_i$  coordinate space  $\mathbb{R}$ , while  $A_i(y^* | \mathbf{x})$  is a number (a probability value). An update step in the resulting Markov chain comprises two probabilistic operations: first, generating the candidate; and second, deciding whether to keep or reject it. Both operations together give the coordinate-specific transition kernels  $T_i$ ; or, if the k individual coordinate steps are lumped together into one full-cycle operation, the single global transition kernel T.

One (preferred, but not the only) way to ensure that the  $T_i$  (or T) lead to a Markov chain whose invariant distribution is the one that one wishes to sample from, is to ensure that the proposal distribution and the acceptance probability together must be designed to warrant detailed balance of  $T_i$ , that is, to ensure

$$\forall \mathbf{x} = (x_1, \dots, x_k)' \in \mathbb{R}^k, \forall y, y' \in \mathbb{R} : g_i(y \mid \mathbf{x}) \ T_i(y' \mid (x_1, \dots, x_{i-1}, y, x_{i+1}, \dots, x_k)') = g_i(y' \mid \mathbf{x}) \ T_i(y \mid (x_1, \dots, x_{i-1}, y', x_{i+1}, \dots, x_k)').$$
(109)

There are several ways to ensure this. For instance, it is not difficult to see (exercise!) that if  $S_i(y^* | \mathbf{x}^{(\nu k+i-1)})$  is symmetric, the acceptance function

$$A_{i}(y^{*} | (x_{1}, \dots, x_{k})') = \frac{g_{i}(y^{*} | (x_{1}, \dots, x_{k})')}{g_{i}(y^{*} | (x_{1}, \dots, x_{k})') + g_{i}(x_{i} | (x_{1}, \dots, x_{k})')}$$
(110)

leads to detailed balance of  $T_i$ . This is called the *Boltzmann acceptance function*. However, the Metropolis algorithm standardly uses another acceptance distribution, namely the *Metropolis acceptance function* 

$$A_{i}(y^{*} | (x_{1}, \dots, x_{k})') = \min\left(1, \frac{g_{i}(y^{*} | (x_{1}, \dots, x_{k})')}{g_{i}(x_{i} | (x_{1}, \dots, x_{k})')}\right).$$
(111)

That is, whenever  $g_i(y^* | (x_1, \ldots, x_k)') \ge g_i(x_i | (x_1, \ldots, x_k)')$ , — i.e. the proposed observation in component *i* has no lower probability than the current one —, accept with certainty; else accept with probability  $g_i(y^* | (x_1, \ldots, x_k)')/g_i(x_i | (x_1, \ldots, x_k)')$ .

For symmetric proposal distributions, the Metropolis acceptance function also implies detailed balance. It is clear that in the rejection case, where the current observation is not changed, the detailed balance condition trivially holds. In the acceptance case we verify (108) as follows:
$$g_{i}(y^{*} | (x_{1}, ..., x_{k})') T_{i}(x_{i} | (x_{1}, ..., y^{*}, ..., x_{n})') =$$

$$= g_{i}(y^{*} | \mathbf{x}) S_{i}(x_{i} | (x_{1}, ..., y^{*}, ..., x_{n})') A_{i}(x_{i} | \mathbf{x})$$

$$= S_{i}(x_{i} | (x_{1}, ..., y^{*}, ..., x_{n})') \min (g_{i}(y^{*} | \mathbf{x}), g_{i}(x_{i} | \mathbf{x}))$$

$$= S_{i}(y^{*} | (x_{1}, ..., x_{i}, ..., x_{n})') \min (g_{i}(y^{*} | \mathbf{x}), g_{i}(x_{i} | \mathbf{x}))$$

$$= g_{i}(x_{i} | \mathbf{x}) S_{i}(y^{*} | (x_{1}, ..., x_{i}, ..., x_{n})') A_{i}(y^{*} | \mathbf{x})$$

$$= g_{i}(x_{i} | \mathbf{x}) T_{i}(y^{*} | (x_{1}, ..., x_{i}, ..., x_{n})').$$

Notes:

- 1. Both the Boltzmann and the Metropolis acceptance functions also work with proto-pdf's  $g_i$ . This is an invaluable asset because often it is infeasible to compute the normalization factor 1/Z needed to turn a proto-pdf into a pdf.
- 2. Like the Gibbs sampler, this local Metropolis algorithm can be turned into a global one, then having a homogeneous Markov chain with transition kernel T, by condensing one k-cycle through the component updates into a single observation update.
- 3. Detailed balance guarantees that g is an invariant distribution of the homogeneous Markov chain with transition kernel T. Before the resulting Metropolis sampler  $Y_1, Y_2, \ldots$  can be used for sampling, it must in addition be shown that this Markov chain is ergodic. The same caveats that we met with the Gibbs sampler apply.
- 4. The Metropolis algorithm is more widely and typically also more easily applicable than the Gibbs sampler because it obviates the need to design an i.i.d. sampler for the coordinate distributions. The price one may have to pay is a higher computational cost, because (i) the rejection events lead to duplicate sample points which obviously leads to an undesirable "repetition redundancy" in the sample that has to be compensated by a larger sample size, and (ii) if the proposal distribution mostly proposes candidate points  $y^*$  for the next samplepoint that lie close to the current sample point, then there will be long washout times after initialization and long interim sequences whose points are strongly dependent on each other and thus contribute no relevant novel information about the target distribution (see Figure 54 b).
- 5. In statistical physics, the term "Monte Carlo Simulation" is almost synonymous with this Metropolis algorithm (says Neal — I am no physicist and couldn't check).
- 6. The quality of Metropolis sampling depends very much on the used proposal distribution. Specifically, the variance of the proposal distribution should neither be too small (then exploration of new states is confined to a narrow

neighborhood of the current state, implying that the Markov chain traverses the distribution very slowly) nor too large (then one will often be propelled far out in the regions of g where it almost vanishes, leading to numerous rejection events). A standard choice is a normal distribution centered on the current state.

7. I presented the Metropolis algorithm in the special case of sampling from a distribution over (a subset of)  $\mathbb{R}^k$ . This made the "cycle through dimensions" scheme natural which we also described for the Gibbs sampler. But the Metropolis algorithm also works on sample spaces S which do not have the vector space structure of  $\mathbb{R}^k$ ; proposal distributions then have to be designed as the individual case requires. The application study from the next subsection is an example.

## 9.6 Application example: determining evolutionary trees

This section is a condensed version of a paper by Mau, Newton, and Larget 1999, *Bayesian Phylogenetic Inference via Markov Chain Monte Carlo Methods*. It is a beautiful demonstration of how various modeling techniques are combined to characterize a highly complex pdf (namely, the conditional distribution of all possible evolutionary pasts, given DNA data from living species), and how the Metropolis algorithm is used to sample from it. The MCMC based method to determine evolutionary trees seems to have become a standard method in the field (I am not an expert to judge). A short paper (Huelsenbeck and Ronquist 2001) announcing a software toolbox for this method has been cited more than 24,000 times on Google Scholar, as of 2022. Methods to reconstruct phylogenetic trees from DNA obtained in surviving species are needed, for instance, to reconstruct trees of evolutionary descendence for man, or to derive hypotheses from which original virus in animals the Covid virus has descended, and when. Such studies often make it into the popular press.

This application example is instructive in several respects. First, it uses Metropolis sampling on a sample space that is very different from  $\mathbb{R}^k$  — the sample space will be a space of trees whose nodes are labelled with discrete symbols and whose edges are labelled with real numbers. This highlights the flexibility and generality of Metropolis sampling. Second, the distribution from which will be sampled is a hyperdistribution — this application example also serves as a demonstration of the usefulness (in this case, even necessity) of Bayesian modeling. And third, the distribution from which will be sampled is represented in a format that is not normalized (the normalization factor 1/Z is unknown).

**Data.** DNA strings from related l living species, each string of length N, have been aligned without gaps. In the reported paper, l = 32 African fish species from the same family (cichlids from central African lakes except one cichlid species from America that served as a control) were represented by DNA sequences of length 1044. 567 of these 1044 sites were identical across all considered species and thus carried no information about phylogeny. The remaining N = 477 sites represented the data D that was entered into the analysis. Reminder: the DNA symbol alphabet is  $\Sigma = \{A, C, G, T\}$ .

Task. Infer from data D the most likely phylogenetic tree, assuming that the considered living species have a common ancestor from which they all descended.

Modeling assumptions. Mutations act on all sites independently. Mutations occur randomly according to a "molecular clock", i.e. a probability distribution of the form

$$P^{\mathsf{clock}}(y \,|\, x, t, \theta) \tag{112}$$

specifing the probability that the symbol  $y \in \{A, C, G, T\}$  occurs at a given site where t years earlier the symbol x occurred.  $\theta$  is a set of parameters specifying further modeling assumptions about the clock mechanism. Mau et al. used the molecular clock model proposed in Hasegawa, Kishino, and Yano 1985 which uses two parameters  $\theta = (\phi, \kappa)$ , the first quantifying an overall rate of mutation, and the second a difference of rates between the more frequent mutations that leave the type of nucleic acid (purine or pyrmidine) unchanged ("transitions") vs. change the type ("transversions"). All that we need to know here, not pretending to be biologists, is that (112) can be efficiently computed. Note that  $\theta$  is not known beforehand but has to be estimated/optimized in the modeling process, based on the data D.

Representing phylogenetic trees. A phylogenetic tree is a binary tree  $\Psi$ . The nodes represent species; leaves are living species, internal nodes are extinct species, the root node is the assumed common ancestor. Mau et al. plot their trees bottom-up, root node at the bottom. Vertical distances between nodes metrically represent the evolutionary timespans t between nodes. Clades are subsets of the leaves that are children of a shared internal node. Figure 56 shows a schematic phylogenetic tree and some clades.

A given evolutionary history can be represented by trees in  $2^{n-1}$  different but equivalent ways (where *n* is the number of living species), through permuting the two child branches of an internal node. For computational purposes a more convenient representation than a tree graph is given by

- 1. a specification of a left-to-right order  $\sigma$  of leaves (in Figure 56,  $\sigma = (1, 4, 7, 2, 3, 6, 5)$ ), plus
- 2. a specification a of the graph distances between two successive leaves.

The graph distance is the length of the connecting path between the two leaves. In the example tree from Figure 56, the distances between leaves make the distance vector  $a = 2(t_1 + t_2 + t_3 + t_4, t_1, t_1 + t_2 + t_3 + t_4 + t_5, t_1 + t_2 + t_3, t_1 + t_2, t_1 + t_2 + t_3 + t_4 + t_5 + t_6)$ . A pair  $(\sigma, a)$  is a compact representation for a phylogenetic tree.

Likelihood of a phylogenetic tree. One subtask that must be solved in order to find the most probable evolutionary tree is to devise a fast method for



Figure 56: An examplary phylogenetic tree (from the Mau et al paper).  $\{4,7\}$ ,  $\{1$  4 7},  $\{2, 3, 6\}$  are examples of clades in this tree.

computing the likelihood of a particular tree  $\Psi$  and molecular clock parameters  $\theta$ , given the sequence data D:

$$L(\Psi, \theta) = P(D \mid \Psi, \theta). \tag{113}$$

Because the mutation processes at different sites are assumed to be independent,  $P(D | \Psi, \theta)$  splits into a product of single-site probabilities  $P(D_i | \Psi, \theta)$ , where  $D_i$ are the symbols found in the sequences from D at site i. Thus  $D_i$  is a symbol vector of length l. Therefore, we only must find a way to compute

$$L_i(\Psi, \theta) = P(D_i | \Psi, \theta). \tag{114}$$

We approach this task sideways, assuming first that we know the symbols  $y_{\nu}$  at the *i*-th site of the internal nodes  $\nu$  of  $\Psi$ . Let  $\rho$  be the root node and  $\pi_0$  a reasonable distribution of symbols in  $\rho$  in site *i* (for instance the global distribution of all symbols in all sites of all sequences in D). Then we get the probability of observed data  $D_i$  joined with the hypothetical data of these  $(y_{\nu})_{\nu \in \mathcal{I}}$  (where  $\mathcal{I}$  is the set of internal nodes of  $\Psi$ ) by

$$P(D_i, (y_{\nu})_{\nu \in \mathcal{I}} | \Psi, \theta) = \pi_0(y_{\varrho}) \prod_{\nu \text{ is non-root node of } \Psi} P^{\mathsf{clock}}(y_{\nu} | y_{\operatorname{par}(\nu)}, t_{\nu}, \theta), \quad (115)$$

where  $par(\nu)$  is the parent node of  $\nu$  and  $t_{\nu}$  is the timespan between  $par(\nu)$  and  $\nu$ . From (115) we could obtain (114) by summing over all possible assignments of symbols to internal nodes, which is clearly infeasible. Fortunately there is a cheap recursive way to obtain (114), which works top-down from the leaves, inductively assigning conditional likelihoods  $L_{\nu}(y) = P(D_i \upharpoonright \nu | \Psi, \theta, \text{node } \nu = y)$  to nodes  $\nu$ ,

where  $y \in \Sigma$  and  $D_i \upharpoonright \nu$  is the subset of the  $D_i$  which are siblings of node  $\nu$ , as follows:

**Case 1:** 
$$\nu \notin \mathcal{I}$$
:  $L_{\nu}(y) = \begin{cases} 1, & \text{if } y = y_{\nu} \\ 0, & \text{else} \end{cases}$   
**Case 2:**  $\nu \in \mathcal{I}$ :  $L_{\nu}(y) = \left(\sum_{z \in \Sigma} L_{\lambda}(z) P^{\mathsf{clock}}(z|y, t_{\lambda}, \theta)\right) \left(\sum_{z \in \Sigma} L_{\mu}(z) P^{\mathsf{clock}}(z|y, t_{\mu}, \theta)\right)$ 

where  $\lambda, \mu$  are the two children of  $\nu, t_{\lambda}$  is the timespan from  $\nu$  to  $\lambda$ , and  $t_{\mu}$  is the timespan from  $\nu$  to  $\mu$ . Then (114) is obtained from

$$L_i(\Psi, \theta) = \sum_{z \in \Sigma} \pi_0(z) \ L_\varrho(z),$$

from which (113) is obtained by

$$L(\Psi, \theta) = \prod_{i \text{ is site in } D} L_i(\Psi, \theta).$$

 $O(N|\Sigma|l)$  flops are needed to compute  $L(\Psi, \theta)$  – in our example, N = 477,  $|\Sigma| = 4$ , l = 32.

The posteriori distribution of trees and mutation parameters. We are actually not interested in the likelihoods  $L(\Psi, \theta)$  but rather in the distribution of models  $\Psi, \theta$  given D – a hyperdistribution. This hyperdistribution is obtained by Mau et al through a Bayesian modeling approach, by inserting prior expert information about the plausibilities of certain values for  $\Psi$  and  $\theta$ . The desired hyperdistribution then is the Bayesian posterior distribution  $P(\Psi, \theta|D)$  (Equation 92), which is proportional to the likelihood times the prior (hyper-)distribution  $P(\Psi, \theta)$  of  $\Psi, \theta$ :

$$P(\Psi, \theta \mid D) \sim P(D \mid \Psi, \theta) \ P(\Psi, \theta) = L(\Psi, \theta) \ P(\Psi, \theta).$$

Lacking a profound theoretical insight, Mau et al. assume for  $P(\Psi, \theta)$  a very simple, uniform-like distribution (such uninformedness is perfectly compatible with the Bayesian approach!). Specifically:

- 1. They bound the total hight of trees by some arbitrary maximum value, that is, all trees  $\Psi$  with a greater hight are assigned  $P(\Psi, \theta) = 0$ .
- 2. All trees of lesser height are assigned the same probability.
- 3. The mutation parameters  $\theta$  are assigned a prior distribution that is uniform on a range interval chosen generously large to make sure that all biologically plausible possibilities are contained in it.

The structure of  $P(\Psi, \theta | D)$ . Before we set forth to compute the posterior hyperdistribution  $P(\Psi, \theta | D)$ , let us take a closer look at the structure of the mathematical "space" in which the parameter pairs  $(\Psi, \theta)$  lie.

Remember that a tree  $\Psi$  is specified by  $(\sigma, a)$ , where  $\sigma$  is a permutation vector of  $(1, \ldots, l)$  and a is a numerical vector of length l - 1. Noticing that there are l!permutations, a pair  $(\sigma, a)$  reflects a point in a product space  $\{1, \ldots, l!\} \times \mathbb{R}^{l-1}$ ; together with the two real-valued parameters comprised in  $\theta$  this brings us to a space  $\{1, \ldots, l!\} \times \mathbb{R}^{l+1}$ . Specifically, this space is a product of a discrete space (the finite but very large set  $\{1, \ldots, l!\}$ ) with a continuous space  $(\mathbb{R}^{l+1})$ . As a consequence, one cannot mathematically describe a probability measure on this space with a pmf, nor with a pdf! And thus, one cannot "compute" a pmf or pdf for  $P(\Psi, \theta | D)$ . So — what *can* we compute for, about, with, or on  $P(\Psi, \theta | D)$ ? The answer is: we can get arbitrarily precise and exhaustive information about  $P(\Psi, \theta | D)$  by ... and only by ... you guess right: sampling.

The Metropolis algorithm at work. Mau et al. use the Metropolis algorithm (in a global version) to sample trees  $\Psi$  and evolutionary clock parameters  $\theta$  from  $P(\Psi, \theta | D)$ . A crucial design task is to find a good proposal distribution  $S((\Psi^*, \theta^*) | (\Psi, \theta))$ . It should lead from any plausible  $(\Psi, \theta)$  ["plausible" means that  $g((\Psi, \theta))$  is not very small] to another plausible  $(\Psi^*, \theta^*)$  which should be however as distinct from  $(\Psi, \theta)$  as possible. The way how Mau et al. go about this task is one of the core contributions of their work.

The authors alternate between updating only  $\theta$  and only  $\Psi$ . Proposing  $\theta^*$  from  $\theta$  is done in a straightforward way: the new \*-parameters are randomly drawn from a rectangular distribution centered on the current settings  $\theta$ .

The tricky part is to propose an as different as possible, yet "plausibilitypreserving" new tree  $\Psi^*$  from  $\Psi$ . Mau et al. transform  $\Psi = (\sigma, a)$  into  $\Psi^* = (\sigma^*, a^*)$  in two steps:

- 1. The current tree  $\Psi$  is transformed into one of its 2n-1 equivalent topological versions by randomly reversing with 0.5 probability every of its internal branches, getting  $\Psi' = (\sigma', a')$ .
- 2. In  $\Psi'$  the evolutionary inter-species time spans t are varied by changing the old values by a random increment drawn from the uniform distribution over  $[-\delta, \delta]$ , where  $\delta$  is a fixed bound (see Figure 57). This gives  $\Psi^* = (\sigma', a^*)$ .

Mau et al. show that this method yields a symmetric proposal distribution, and that every tree  $\Psi'$  can be reached from every other tree  $\Psi$  in a bounded number of such transformations – the Markov chain is thus ergodic.

Concretely, the Metropolis algorithm was run for 1,100,000 steps (20 hours CPU time on a 1998 Pentium 200 PC). The first 100,000 steps were discarded to wash out possible distortions resulting from the arbitrary starting tree. The remaining 1,000,000 trees were subsampled by 200 (reflecting the finding that after every 200 steps, trees were empirically independent [zero empirical cross-correlation at 200 step distance]), resulting in a final tree sample of size 5000.



Figure 57: Proposal candidate trees, attainable from the current tree, are found within timeshift intervals of size  $2\delta$ , centered at the current internal nodes, that constrain the repositioning of the internal nodes. Note that if the two rightmost internal nodes are shifted such that their relative heights become reversed (dashed blue circles), the topology of the tree would change (dashed blue lines). Figure adapted from the Mau et al paper.

Final findings. Recall that the overall goal of this entire effort was to determing which tree *topology* is best explaining the current genetic Data D, under the assumptions of the Bayesian prior  $P(\Psi, \theta)$ . After the sampling was done, 5000 trees had been collected whose sample frequencies reflect their Bayesian posterior probabilities. The rest is easy: sort these sampled trees into different subsets, each of which is defined by a specific tree topology, then interpret the relative sizes of these sets as probability estimates.

The 600 (!) most frequent topologies make up for 90% of the total probability mass. This high variability however is almost entirely due to minute variations within 6 clades (labelled A, ..., F) that are stable across different topologies. Figure 58 shows the two most frequently found topologies resolved at the clade level, with a very strong posterior probability indication for the first of the two.

For quality control, this was repeated 10 times, with no change of the final outcome, which makes the authors confident of their work's statistical reliability.



Figure 58: The two clade tree structures of highest posterior probability found by Mau et al.

# 10 Graphical models

My favourite motivational example for introducing graphical models is the Space Shuttle. In the launch phase, the engineering staff in the control center and the astronauts on board have to make many critical decisions under severe time pressure. If anything on board of the roaring Space Shuttle seems to go wrong, it must be immediately determined, for instance, whether to shut down an engine (means ditching the Space Shuttle in the ocean but hopefully saving the astronauts) or not (might mean explosion of the engine and death of astronauts — or it might be just the right thing to do if it's a case of sensor malfunction and in fact all is fine with the engine). The functioning of the Space Shuttle is monitored with many dozens of sensors, which under the enormous vibration stress are more prone to malfunction than you would believe. They are delivering a massive flow of information which is impossible for human operators to evaluate for fast decision-making. Socalled *decision support systems* are needed which automatically calculate probabilities for the actual system state from sensor readings and display to the human operators a condensed, human-understandable view of its most decision-critical findings. Read the original paper of Horvitz and Barry 1995 on the probabilistic modeling of the combined SpaceShuttle-Pilots-GroundControlStaff system if you want to see how thrilling and potentially life-saving statistical modeling can be.

Such decision support systems are designed around hidden and visible random variables. The visible RVs represent observable variables, for instance the pressure or temperature readings from Space Shuttle engine sensors, but also actions taken by the pilot. The hidden RVs represent system state variables which cannot be directly measured but which are important for decision making, for instance a binary RV "Sensor 32 is functioning properly — yes / no", or "pilot-in-command is aware of excess temperature reading in engine 3 — yes / no". There are many causal chains between the hidden and visible RVs which lead to chains of conditional probability assessments, for instance "If sensor reading 21 is 'normal' and sensor reading 122 is 'normal' and sensor reading 32 is 'excess temperature', the probability that sensor 32 is misreading is 0.6". Such causal dependencies between RVs are mathematically represented by arrows between the participating RVs, which

in total gives a directed graph whose nodes are the RVs. In such a graph, each RV X has its own sample space  $S_X$  which contains the possible values that X may take. When the graph is used (for instance, for decision support), for each RV a probability distribution  $P_X$  on  $S_X$  is computed. If  $X_1, \ldots, X_k$  are RVs with arrows fanning in on a RV Y, the probability distribution  $P_Y$  on  $S_Y$  is calculated as a conditional distribution which depends on the distributions  $P_{X_1}, \ldots, P_{X_k}$ . These calculations quickly become expensive when the graph is large and richly connected (exact statistical inference in such graphs is NP-hard) and require approximate solution strategies — for instance, through sampling methods. Despite the substantial complexity of algorithms in the field, such graphical models are to-day widely used. Many special sorts of graphical models have traditional special names and have been investigated in a diversity of application contexts long before the general, unifying theory of graphical models was developed. Such special kinds of graphical models or research traditions include

- *Hidden Markov models*, a basic model of stochastic processes with memory, which for decades have been the standard machine learning model for speech and handwriting recognition and today still are the reference model for DNA and protein sequences;
- models for *diagnostic reasoning* where a possible disease is diagnosed from symptoms and medical lab results (also in fault diagnostics in technical systems when your car [if you already possess one] is checked at a service point, collected sensor data are read out and pumped through a Bayesian network model of the car to see what are probably sources of malfunction);
- *decision support systems*, not only in Space Shuttle launching but also in economics or warfare to name but two;
- *human-machine interfaces* where the machine attempts to infer the user's state of mind from the user's actions the (in-)famous and now abolished Microsoft Office "paperclip" online helper was based on a graphical model;
- *Markov random fields* which model the interactions between local phenomena in spatially extended systems, for instance the pixels in an image;
- *Boltzmann machines*, a neural network model of hierarchical information processing.

Graphical models, and in particular Bayesian networks, have recently re-emerged in machine learning in a new, and possibly revolutionary, role. Since the early 1990's, the pioneer of Bayesian networks, Judea Pearl, has been busy working out ways how *causal* relationships between real-world state variables can be inferred from *statistical* relationships between them. It is by no means clear that this should be possible at all — as witnessed by the cautionary warning "correlation is not causation" that is given to all students in statistics classes. Using the representations of complex, multi-variable probability distributions in Bayesian networks as a starting point, Pearl nonetheless found ways to achieve exactly that, inferring causation from (conditional) probability. For a long time, these insights were hardly taken up by machine learners, although they had an impact in medical and epidemiologial statistics. This has changed. *Causal modeling* is becoming an important topic in deep learning. Specifically, structuring modular neural network systems along the lines of causal chains (and cycles) is emerging as a possible key for achieving artifical general intelligence — machine learning systems that can cope with many tasks of many sorts (like humans and unlike current deep learning systems), using only a few training examples (like humans and unlike current deep learning systems), and generalizing very robustly to out-of-distribution input (like humans and unlike current deep learning systems). Schölkopf, Locatello, et al. 2021 presents a very well-written tutorial on causal modeling.

In this section I will give an introduction to the general theory of graphical models. In my condensed treatment I lean heavily on the two tutorial texts by Pearl and Russell 2003 and K. Murphy 2001. This material is very useful by itself (e.g. if you want to help Space Shuttles to be launched safely), and it is also requisite background for understanding causal modeling.

Graphical models are a heavyweight branch of machine learning and would best be presented in an entire course of their own. The course *Probabilistic Graphical Models* of Stefano Ermon at Stanford University is a (beautifully crafted) example. The course homepage https://cs228.stanford.edu/ gives links to literature and programming toolboxes — and it serves you a transparently written set of lecture notes on https://ermongroup.github.io/cs228-notes/ which culminates in explaining *deep variational autoencoders*, a powerful deep learning method which includes a number of techniques from graphical models.

### 10.1 Bayesian networks

Bayesian networks (BNs) are graphical models where the graph structure is a finite, directed, acyclic (that is, the arrows never form cycles) graph. Today's general theory of graphical models emerged from research on Bayesian networks, and many themes of graphical models are best introduced with Bayesian networks.

The classical application domains for BNs are decision support, user modeling, and diagnostic systems. BN algorithms and fast computers have become indispensible for decision support because humans are simply bad at correctly combining pieces of stochastic evidence in uncertain situations. They tend to make gross, systematic prediction errors even in simple diagnostic tasks that involve only two observables — which is why in good medical schools, future doctors must take courses in diagnostic reasoning —, let alone in tasks concerning complex systems with dozens or hundreds of random variables.

For a first impression, we consider a classical (and I must say, somewhat stupid)

example which you can find in many BN tutorials. Let  $X_1, \ldots, X_5$  be five discrete random variables, indicating the following observations:

- $X_1$ : indicates the season of the year, has values in  $S_{X_1} = \{$ Winter, Spring, Summer, Fall $\},$
- $X_2$ : indicates whether it rains, with values  $\{0, 1\}$ ,
- $X_3$ : indicates whether the lawn sprinkler is on, has values  $\{0, 1\}$ ,
- $X_4$ : indicates whether the pavement (close to the lawn) is wet, values  $\{0, 1\}$ ,
- $X_5$ : indicates whether the pavement is slippery, values from  $\{0, 1\}$ , too.

There are certain causal influences between some of these random variables. For instance, the season co-determines the probabilities for rain; the sprinkler state co-determines whether the pavement is wet (but one would not say that the wetness of the pavement has a causal influence on the sprinkler state), etc. Such influences can be expressed by arranging the  $X_i$  in a directed acyclic graph (a DAG), such that each random variable becomes a node, with an edge (i, j)indicating that what is measured by  $X_i$  has some causal influence on what is measured by  $X_j$ . Of course there will be some subjective judgement involved in claiming a causal influence between two observables, and denying it for other pairs – such dependency graphs are not objectively "true", they are designed to represent one's view of a part of the world (note that these are not the causal links that are studied in causal modeling. In causal modeling, causal influences are not stipulated by a subjective researcher but become mathematically identified). Figure 59 is the DAG for our example.



Figure 59: A simple Bayesian network. Image taken from Pearl and Russell 2003.

A DAG must satisfy the following formal condition in order to qualify as a Bayesian network:

**Definition 10.1** A directed acyclic graph with nodes labelled by RVs  $\{X_1, \ldots, X_n\}$ (each with its own sample space  $S_i$ ) is a Bayesian network (BN) for the joint distribution  $P_{\mathbf{X}}$  of  $\mathbf{X} = X_1 \otimes \ldots \otimes X_n$  if every  $X_i$  is conditionally independent of its non-descendants in the graph given its parents.

For convenience of notation we often identify the nodes  $X_i$  of a BN with their indices *i*. The *descendants* of a node *i* in a DAG *G* are all nodes *j* that can be reached from *i* on a forward path in *G*. Descendance is a transitive relation: if *j* is a descendant of *i* and *k* a descendant of *j*, then *k* is a descendant of *i*. The non-descendants of *i* are all *j* that are not descendants of *i*.

The *parents* of a node i are all the immediate graph predecessors of i, that is, all j such that (j, i) is an edge in the graph.

For instance, in the DAG shown in Figure 59,  $X_3$  has parent  $X_1$  and descendants  $X_4, X_5$ , and the condition stated in the definition requires that  $P(X_3|X_1)$ is independent of  $X_2$ , that is  $P(X_3|X_1) = P(X_3|X_1, X_2)$ : our judgement whether the sprinkler is on or off is not influenced by our knowledge whether it rains or not, given that we know what season it is. Personal note: I always found this a weird Californian way of behaving: those guys would set the sprinkler on or off just depending on the season; I guess in Summer all Californian sprinklers sprinkled all Summer, come rain come shine (in the year 2000 when that tutorial was written; today Californians surely are more water-aware).

The independence relations expressed in a BN in terms of parents and nondescendants need not be the only independence relations that are actually true in the joint distribution. In our example, for instance, it may be the case that the pavement is always slippery because it was made from polished marble. Then  $X_5$  would be unconditionally independent from any of the other variables. The complete independence relations between the variables figuring in a BN depend on the particulars of the joint distribution and need not all be represented in the graph structure. A BN is only a partial model of the independence relations that may be present in the concerned variables.

Let's squeeze in a short remark on notation here. By  $P(X_4 | X_2, X_3)$  we denote the conditional *distribution* of  $X_4$  given  $X_2$  and  $X_3$ . In rigorous mathematical texts one would denote this distribution by  $P_{X_4 | X_2, X_3}$ , but I will follow the notation in the Pearl/Russell tutorial in this section. For discrete random variables, as in our example, such conditional distributions can be specified by a table — for  $P(X_4 | X_2, X_3)$  such a table might look like

$X_2$	$X_3$	$P(X_4 = 0)$	$P(X_4 = 1)$
0	0	1.0	0.0
0	1	0.1	0.9
1	0	0.1	0.9
1	1	0.01	0.99

To specify a BN, one must supply such conditional distributions for every RV  $X_i$  in the network. They give the probabilities for values of  $X_i$  for all value combinations of its parents. If a RV  $X_i$  has no parents (that is, it is a root node in the DAG), then this conditional distribution is conditioned on nothing — it is just a plain probability distribution of the RV  $X_i$ .

For continuous-valued random variables, such conditional distributions cannot in general be specified in a closed form (one would have to specify pdf's for each possible combination of values of the conditioning variables), except in certain special cases, notably Gaussian distributions. One must then supply a computable mathematical function which allows one to compute all concrete probability densities like  $p(X_4 = x_4 | X_2 = x_2, X_3 = x_3)$ , where p is a pdf.

I use lowercase  $P(x_4 | x_2, x_3)$  as a shorthand for  $P(X_4 = x_4 | X_2 = x_2, X_3 = x_3)$ . This denotes a single probability number for particular values of our random variables. – End of the remark on notation.

A Bayesian network can be used for reasoning about uncertain causes and consequences in many ways. Here are three kinds of arguments that are frequently made, and for which BNs offer algorithmic support:

- **Prediction.** "If the sprinkler is on, the pavement is wet with a probability  $P(X_4 = 1|X_3 = 1) = \#\#\#\#$ ": reasoning from causes to effects, along the arrows of the BN in forward direction. Also called *forward reasoning* in AI contexts.
- Abduction. "If the pavement is wet, it is more probable that the season is spring than that it is summer, by a factor of ### percent": reasoning from effects to causes, that is, diagnostic reasoning, backwards along the network links. (By the way, for backward reasoning you need Bayes' formula, which is what gave Bayesian networks their name.)
- Explaining away, and the rest. "If the pavement is wet and we don't know whether the sprinkler is on, and then observe that it is raining, the probability of the sprinkler being on, too, drops by ### percent: in "explaining away" there are several possible causes  $C_1, \ldots, C_k$  for some observed effect E, and when we learn that actually cause  $C_i$  holds true, then the probabilities drop that the other causes are likewise true in the current situation. There are many other variants of reasoning "sideways".

Bayesian networks offer *inference algorithms* to carry out such arguments and compute the correct probabilities, ratios of probabilities, etc. These inference algorithms are neither cheap nor easy to understand, and for some time researchers believed that these inferences cannot be computed in feasible runtimes at all. It came as a landslide surprise when practical algorithms were discovered in the mid-80's, and only then Bayesian networks began to make their way into applications. I was personally present at an AI workshop (the annual meeting of the AI section of the German Society for Computer Science) where the speaker – sadly I forget who the speaker was – described an early version of such an algorithm, and was met with skepticism or plain disbelief by the audience. I will explain a classical (and still widely used) algorithm in this section (the *join tree* algorithm for exact inference); it is efficient in BNs whose connectivity is not too dense. Because exact statistical inference in BNs is NP-hard, one has to take resort to approximate algorithms in many cases. There are two main families of such approximate algorithms, one based on sampling and the other on *variational approximation*. I will give a hint on inference by sampling and omit variational inference.

In a sense, the most natural (the only?) thing you can do when it comes to handle the interaction between many random variables is to arrange them in a causal influence graph. So it is no surprise that related formalisms have been invented independently in AI, physics, genetics, statistics, and image processing. However, the most important algorithmic developments have been in AI / machine learning, where feasible inference algorithms for large-scale BNs have first been investigated. The unrivalled pioneer in this field is Judea Pearl http://bayes.cs.ucla.edu/jp\_home.html, who laid the foundations for the algorithmic theory of BNs in the 1980's. These foundations were later developed to a more general theory of graphical models, a development promoted (among others) by Michael I. Jordan https://people.eecs.berkeley.edu/~jordan/. Michael Jordan not only helped to build the general theory of graphical models but also had a shaping influence on other areas in machine learning. He is a machine learning superpower. The list of his past students and postdocs on his homepage is awe-inspiring and reads like a Who's Who of machine learning.

In the world of BNs (and graphical models in general) there exist two fundamental tasks:

- **Inference:** Find algorithms that exploit the graph structure as ingeneously as possible to yield feasible algorithms for computing all the quantities asked for in prediction, abduction and explaining-away tasks.
- **Learning:** Find algorithms to estimate a BN from observed empirical data. Learning algorithms typically call inference algorithms as a subroutine, so we will first treat the latter.

#### 10.1.1 Brute-force inference in BNs

A Bayesian network is a probabilistic representation of a piece of reality – that piece which we observe through the visible RVs and that we hypothetically capture through the hidden RVs and all the arrows. A BN represents the *joint* probability distribution of all of these RVs. In inference tasks one computes distributions of target variables of interest ("probability that Space Shuttle engine will explode") as marginal or conditional probabilities derived from that joint distribution. In this subsection I explain how such marginal or conditional distributions are mathematically connected to the joint distribution in cases where all RVs have finite sample spaces, and how the graph structure can be exploited to *factorize* distributions of interest, which makes them computationally more tractable. The joint distribution in our Californian pavement example is a distribution over the sample space

$$\{\text{Winter, Spring, Summer, Fall}\} \times \{0, 1\} \times \{0, 1\} \times \{0, 1\} \times \{0, 1\} \times \{0, 1\},\$$

which has  $4 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 64$  elements. Thus one would need a 64-dimensional pmf to characterize it, which has 63 degrees of freedom (one parameter we get for free because the pmf must sum to 1). This pmf would not be given in vector format but as a 5-dimensional array of size  $4 \times 2 \times 2 \times 2 \times 2$ . We will now investigate brute-force methods for elementary inference in this joint probability array and see how the graph structure leads to a dramatic reduction in computational complexity (which however still is too high for practical applications with larger BNs – more refined algorithms to be presented later).

By a repeated application of the factorization formula (174) (in Appendix B), the joint distribution of our five random variables is

$$P(\mathbf{X}) = P(\bigotimes_{i=1,\dots,5} X_i) = P(X_1) P(X_2|X_1) P(X_3|X_1, X_2) P(X_4|X_1, X_2, X_3) P(X_5|X_1, X_2, X_3, X_4).$$
(117)

Exploiting the conditional independencies expressed in the BN graph, this reduces to

$$P(\mathbf{X}) = P(X_1) P(X_2|X_1) P(X_3|X_1) P(X_4|X_2, X_3) P(X_5|X_4).$$
(118)

For representing the factors on the right-hand side of (118) by tables like (116), one would need tables of sizes  $1 \times 4$ ,  $4 \times 2$ ,  $4 \times 2$ ,  $4 \times 2$ , and  $2 \times 2$ , respectively. Because the entries per row in each of these tables must sum to 1, one entry per row is redundant, so these tables are specified by 3, 4, 4, 4 and 2 parameters, respectively. All in all, this makes 17 parameters needed to specify  $P(\mathbf{X})$ , as opposed to the 63 parameters needed for the naive representation  $P(\mathbf{X})$  in a 5-dimensional array.

In general, the number of parameters required to specify the joint distribution of n discrete random variables with maximally  $\nu$  values each, arranged in a BN with a maximum fan-in of k, is  $O(n\nu^k)$  as opposed to the raw number of parameters  $O(\nu^n)$  needed for a naive characterization of the joint distribution. This is a reduction from a space complexity that is exponential in n to a space complexity that is linear in n! This simple fact has motivated many a researcher to devote his/her life to Bayesian networks.

Any reasoning on BNs (predictive, abductive, sidestepping or other) boils down to calculate conditional or marginal probabilities. For instance, the abductive question, "If the pavement is wet, by which factor y is it more probable that the season is spring than that it is summer", asks one to compute

$$y = \frac{P(X_1 = \text{spring} \mid X_4 = 1)}{P(X_1 = \text{summer} \mid X_4 = 1)}.$$
(119)

Such probability ratios are often sought in diagnostic reasoning — as for instance in "by which factor is it more probable that my symptoms are due to cancer, than that they are due to having eaten too much sauerkraut".

Any conditional probability  $P(y_1, \ldots, y_m | z_1, \ldots, z_l)$ , where the  $Y_i$  and  $Z_j$  are among the RVs in the BN, can be computed from the joint distribution of all variables in the BN by first transforming  $P(y_1, \ldots, y_m | z_1, \ldots, z_l)$  into a fraction of two marginal probabilities,

$$P(y_1, \dots, y_m \,|\, z_1, \dots, z_l) = \frac{P(y_1, \dots, y_m, z_1, \dots, z_l)}{P(z_1, \dots, z_l)}$$

and then computing the denominator and the enumerator by marginalization from the joint distribution of all RVs in the BN, exploiting efficient BN factorizations of the kind exemplified in Equation 118. The probability  $P(X_1 = \text{spring} | X_4 = 1)$ , for instance, can be computed by

$$P(X_{1} = \operatorname{spring} | X_{4} = 1) =$$

$$= \frac{P(X_{1} = \operatorname{spring}, X_{4} = 1)}{P(X_{4} = 1)}$$

$$= \frac{\sum_{x_{2}, x_{3}, x_{5}} P(X_{1} = \operatorname{spring}, x_{2}, x_{3}, X_{4} = 1, x_{5})}{\sum_{x_{1}, x_{2}, x_{3}, x_{5}} P(X_{1} = \operatorname{s}) P(x_{2} | X_{1} = \operatorname{s}) P(X_{3} | X_{1} = \operatorname{s}) P(X_{4} = 1 | x_{2}, x_{3}) P(x_{5} | X_{4} = 1)}$$

$$= \frac{\sum_{x_{1}, x_{2}, x_{3}, x_{5}} P(X_{1} = \operatorname{s}) P(x_{2} | X_{1} = \operatorname{s}) P(X_{4} = 1 | x_{2}, x_{3}) P(x_{5} | X_{4} = 1)}{\sum_{x_{1}, x_{2}, x_{3}, x_{5}} P(x_{1}) P(x_{2} | x_{1}) P(x_{3} | x_{1}) P(X_{4} = 1 | x_{2}, x_{3}) P(x_{5} | X_{4} = 1)},$$

where I abbreviated "spring" to "s" in order to squeeze the expression into a single line.

(120) can be computed by a brute-force evaluation of the concerned summations. The sum to be taken in the denominator would run over  $4 \times 2 \times 2 \times 2 =$ 32 terms, each of which is a product of 5 subterms; we thus would incur 128 multiplications. It is apparent that this approach generally incurs a number of multiplications that is exponential in the size of the BN.

A speedup strategy is to try pulling the sum into the product as far as possible, and evaluate the resulting formula from the inside of bracketing levels. This is called the method of *variable elimination*. For example, an equivalent formula for the sum in the denominator of (120) would be

$$\sum_{x_1} \left( P(x_1) \sum_{x_2, x_3} \left( P(x_2 | x_1) P(x_3 | x_1) P(X_4 = 1 | x_2, x_3) \left( \sum_{x_5} P(x_5 | X_4 = 1) \right) \right) \right).$$

To evaluate this expression from the inside out, we note that the sum over  $x_5$  in the innermost term is 1 and need not be explicitly calculated. For the remaining calculations, 15 sums and 36 multiplications are needed, as opposed to the 31 sums and 128 multiplications needed for the naive evaluation of the denominator in (120). However, finding a summation order where this pulling-in leads to the minimal number of summations and multiplications is again NP-hard, although greedy algorithms for that purpose are claimed to work well in practice.

In such a situation, computer scientists can choose between three options:

- 1. Restrict the problem by suitable constraints, earning a tractable problem. For BNs, this was the first strategy that was used with success: early (now classical) inference algorithms were defined only for BNs with tree graphs.
- 2. Use heuristic algorithms, i.e. algorithms that embody human insight for computational shortcuts. Heuristic algorithms need not always be successful in leading to short runtimes; if they do, their result is perfectly accurate. The goal is to find heuristics which lead to fast runtimes almost always and which run too long only rarely. The "join tree" algorithm which we will study later contains heuristic elements.
- 3. Use approximate algorithms, i.e. algorithms that yield results always and fast, but with an error margin (which should be controllable). For BN inference, a convenient class of approximate algorithms is based on sampling. In order to obtain an estimate of some marginal probability, one samples from the distribution defined by the BN and uses the sample as a basis to estimate the desired marginal. There is an obvious tradeoff between runtime and precision. Another class of approximate algorithms is to use variational inference. Variational algorithms need some insight into the shape of the conditional distributions in a BN. Their speedup results from restricting the admissible shapes of these distributions to analytically tractable classes. I will not go into this topic in this course — it's not easy. Jordan, Ghahramani, et al. 1999 give a tutorial introduction.

#### 10.1.2 BN inference with the join-tree algorithm

In this subsection I will unfold the classical exact inference algorithm that can compute conditional and marginal probabilities on BNs at (often) affordable cost. This algorithm is often referred to as *join tree algorithm*, or *junction tree algorithm*. It is an exact algorithm: it computes exact, not merely approximate values of the desired probabilities. And it is a heuristic algorithm, because some substeps can be set up in several ways, and deciding which set-up one uses is often done based on the heuristic 'this has been reported in the literature to mostly work well'.

The join-tree algorithm isn't simple and this subsection will be long. I will follow the description given in Huang and Darwiche 1994.

Before the BN can be used with the join tree inference algorithm, the original DAG must be transformed into an auxiliary graph structure, called a *join tree*, by a rather involved process. This has to be done only once for a given BN

however; the same join tree can be re-used for all subsequent inference calls. The transformation from a DAG to a join tree runs through 4 steps. Here is a first overview:

**Given:** A BN with DAG  $G_d$  with nodes  $\mathbf{X} = \{X_1, \ldots, X_n\}$ 

- Step 1: Transform  $G_d$  to an undirected graph  $G_m$ , called the *moral undirected* graphical model (moral UGM), which is an equivalent way of expressing the independencies expressed in  $G_d$ , but using another graph separation mechanism for expressing conditional independencies.
- Step 2: Add some further undirected edges to  $G_m$  which triangulate  $G_m$ , obtaining  $G_t$ . This may destroy some of the original independence relations between  $\{X_1, \ldots, X_n\}$  but will not introduce new ones. This step has no unique single solution, and heuristics enter the game to decide which triangulation to use.
- Step 3: Detect all maximal cliques  $\mathbf{C}_i$  in  $G_t$ . While this is NP-complete for general undirected graphs, it can be done efficiently for triangulated undirected graphs (Note: a *clique* in an undirected graph is a subset of nodes that are all pairwise connected to each other. A subset of a clique is again a clique. A clique  $\mathbf{C}$  is maximal if no vertex outside  $\mathbf{C}$  has edges to all vertices in  $\mathbf{C}$ .)
- Step 4: Build an undirected *join tree*  $\mathcal{T}$  with nodes  $\mathbf{C}_i$ . This is the desired target structure. It represents an elegant factorization of the joint probability  $P(\mathbf{X})$  which in turn can be processed with a fast, local inference algorithm known as *message passing*. Again there is no unique way to create the join tree and heuristics are used to obtain one that leads to fast inference algorithms.

We will go through these steps in a mostly informal manner. The purpose of this presentation is not to provide you with a detailed recipe for building executable code. There exist convenient BN toolboxes that you would rather use. The purpose of this subsection is to provide you with a navigation guide to gain an overview of the general picture, then study the more detailed paper Huang and Darwiche 1994, then start using (and understanding) a toolbox.

**Undirected graphical models.** Before we can delve into the join tree algorithm, we have to introduce the concept of undirected graphical models (UGMs), because these will be constructed as intermediate data structures when a BN is transformed to a join tree.

The essence of BNs is that conditional independency relationships between RVs are captured by directed graph structures, which in turn guide efficient inference algorithms. But it is also possible to use undirected graphs. This leads to undirected graphical models (UGMs), which have a markedly different flavour from directed BNs. UGMs originated in statistical physics and image processing, while BNs were first explored in Artificial Intelligence. A highly readable nontechnical overview and comparison of directed and undirected models is given in Smyth 1997.

We will use the following compact notation for statistical, conditional independence between two sets  $\mathbf{Y}$  and  $\mathbf{Z}$  of random variables, given a another set  $\mathbf{S}$  of random variables:

**Definition 10.2** Two sets  $\mathbf{Y}$  and  $\mathbf{Z}$  of random variables are independent given  $\mathbf{S}$ , if

$$P(\mathbf{Y}, \mathbf{Z} \mid \mathbf{S}) = P(\mathbf{Y} \mid \mathbf{S}) \ P(\mathbf{Z} \mid \mathbf{S}).$$

We write  $\mathbf{Y} \perp \mathbf{Z} \mid \mathbf{S}$  to denote this conditional independence.

It is easy to see that if  $\mathbf{Y} \perp \mathbf{Z} \mid \mathbf{S}$  and  $\mathbf{Y}' \subseteq \mathbf{Y}, \mathbf{Z}' \subseteq \mathbf{Z}$ , then  $\mathbf{Y}' \perp \mathbf{Z}' \mid \mathbf{S}$ .

In an UGM, independence relations between sets of random variables are defined in terms of a graph separation property:

**Definition 10.3** Let G = (V, E) be an undirected graph with vertices V and edges  $E \subseteq V \times V$ . Let  $Y, Z, S \subset V$  be disjoint, nonempty subsets. Then Y is separated from Z by S if every path from some vertex in Y to any vertex in Z contains a node from S. S is called a separator for Y and Z.

And here is how statistical independence is reflected in a graph's separation properties:

**Definition 10.4** An undirected graph with nodes  $\mathbf{X} = \{X_1, \ldots, X_n\}$  is an undirected graphical model *(UGM)* for  $P(\mathbf{X})$  if for all  $\mathbf{Y}, \mathbf{Z} \subseteq \mathbf{X}$  it holds that if  $\mathbf{Y}$  is separated from  $\mathbf{Z}$  by  $\mathbf{S}$ , then  $\mathbf{Y} \perp \mathbf{Z} \mid \mathbf{S}$ .

UGMs are sometimes defined in a slightly different but equivalent way. We formulate this other definition as a proposition:

**Proposition 10.1** An undirected graph with nodes  $\mathbf{X} = \{X_1, \ldots, X_n\}$  is an UGM for  $P(\mathbf{X})$  iff for all  $i = 1, \ldots, n$  it holds that  $P(X_i | (X_j)_{j \neq i}) = P(X_i | (X_j)_{j \in \mathcal{N}_i})$ , where  $\mathcal{N}_i$  is the set of all direct graph neighbors of node *i*.

In an UGM, the set  $\mathcal{N}_i$  of all direct neighbors of  $X_i$ , which makes  $X_i$  independent from all other RVs in the model, is also called the *Markov blanket* of  $X_i$ . Notice that Markov chains (see Section 9.5.1) can be considered as UGMs where the (only) edges are between subsequent  $X_n, X_{n+1}$  and the Markov blanket of  $X_n$  is given by its temporal neighbors  $\{X_{n-1}, X_{n+1}\}$ . Step 1: The moral UGN. After this quick introduction to UGMs we return to the join tree algorithm for BNs. The first step is to transform the directed graph structure of a BN into an UGM structure. This can be done in many ways. The art lies in transforming a BN into an UGM such that as few as possible of the valuable independence relations expressed in the BN get lost. The standard thing to do as a first step is to *moralize* the directed BN graph  $G_d$  into an undirected graph  $G_m$  by

- 1. converting all edges from directed to undirected ones,
- 2. for every node  $X_i$ , adding undirected edges between all parents of  $X_i$ .

See Figure 60 for an example. The peculiar name "moralizing" comes from the act of "marrying" previously unmarried parents. The moral UGM  $G_m$  implies the same conditional independence relations as the BN  $G_d$  from which it was derived (proof omitted here).



Figure 60: A BN and its associated moral UGM. Image taken from Huang and Darwiche 1994.

**Step 2: Triangulation.** An undirected graph is *triangulated* if every cycle of length at least 4 contains two nodes not adjacent in the cycle which are connected by an edge.

Any undirected graph can be triangulated by adding edges. There are in general many ways of doing so. For best efficiency of UGM inference algorithms, one should aim at triangulating in a way that minimizes the size of cliques of the triangulated graph. This is (again) an NP-hard problem. However there are heuristic algorithms which yield close to optimal triangulations in (fast) polynomial time. One of them is sketched in the Huang/Darwiche guideline for BN inference design.



Figure 61: Left: A triangulated version  $G_t$  of the moral UGM  $G_m$  from Figure 60. Right: the 6 cliques in  $G_t$ . Image adopted from Huang and Darwiche 1994.

After triangulation, we have an UGM graph  $G_t$ . Figure 61 shows a possible triangulation of the UGM from Figure 60.

The reason for triangulating is that triangulated graphs can be decomposed into junction trees in Step 4.

Step 3: Finding the cliques. For a change, finding all maximal cliques in a triangulated graph is not NP-hard — efficient algorithms for finding all maximal cliques in a triangulated graph are known. In our running example, we get 6 cliques, namely all the "triangles" ABD, ACE, ADE, DEF, CGE, and EGH (Figure 61 (right)). Our example conveys maybe a wrong impression that we always get cliques of size 3 in triangulated graphs. In general, one may find maximal cliques of any size  $\geq 2$  in such graphs.

Step 4: Building the join tree. This is a more interesting and involved step. After BNs and UGMs, join trees are our third graph-based representation of independence relations governing a set  $\mathbf{X}$  of random variables. We will first discuss join trees in their own right, and then consider how a join tree can be obtained from the cliques of a triangulated UGM.

**Definition 10.5** Let  $\mathbf{X} = \{X_1, \ldots, X_n\}$  be a set of discrete random variables with joint distribution  $P(\mathbf{X})$ , where  $X_i$  takes values in a sample space  $S_{X_i}$ . A join tree for  $P(\mathbf{X})$  is an undirected tree whose nodes are labelled with subsets  $\mathbf{C}_i \subseteq \mathbf{X}$  (called clusters) and whose edges are labelled with subsets  $\mathbf{S}_j \subseteq \mathbf{X}$  (called the separator sets or sepsets). Furthermore, each cluster  $\mathbf{C} = \{Y_1, \ldots, Y_k\}$  is associated with a belief potential  $\varphi_{\mathbf{C}} : S_{Y_1} \times \ldots \times S_{Y_k} \to \mathbb{R}^{\geq 0}$ , and each sepset  $\mathbf{S} = \{S_1, \ldots, S_l\}$  with a belief potential  $\varphi_{\mathbf{S}} : S_{S_1} \times \ldots \times S_{S_l} \to \mathbb{R}^{\geq 0}$ , such that the following conditions are satisfied:

1. The graph has the running intersection property, that is, given two cluster nodes labelled with C and C', all labels C<sup>\*</sup> of cluster nodes on the path between C and C' must have labels containing  $C \cap C'$ .

- 2. Each edge between two nodes labelled by  $\mathbf{C}$  and  $\mathbf{C}'$  is labelled by  $\mathbf{S} = \mathbf{C} \cap \mathbf{C}'$ .
- 3. For each cluster  $\mathbf{C} = \{Y_1, \ldots, Y_l, Y_{l+1}, \ldots, Y_k\}$  and neighboring sepset  $\mathbf{S} = \{Y_1, \ldots, Y_l\}$  (note that  $l \leq k$  because  $\mathbf{S} \subseteq \mathbf{C}$ ), it holds that  $\varphi_{\mathbf{C}}$  is consistent with  $\varphi_{\mathbf{S}}$  in the sense that

$$\sum_{y_{l+1},\dots,y_k} \varphi_{\mathbf{C}}(y_1,\dots,y_k) = \varphi_{\mathbf{S}}(y_1,\dots,y_l), \qquad (121)$$

that is,  $\varphi_{\mathbf{S}}$  is obtained by marginalization from  $\varphi_{\mathbf{C}}$ .

4. The joint distribution  $P(\mathbf{X})$  is factorized by the belief potentials according to

$$P(\mathbf{X}) = \frac{\prod_{i} \varphi_{\mathbf{C}_{i}}}{\prod_{j} \varphi_{\mathbf{S}_{j}}},\tag{122}$$

where i ranges over all clusters and j over all sepsets.



Figure 62: A join tree derived from the triangulated UGN shown in Figure 61. Image taken from Huang and Darwiche 1994.

I mention without proof how statistical independence relations are reflected in a join tree:

**Proposition 10.2** Let  $\mathcal{T}$  be a join tree for  $P(\mathbf{X})$  and  $\mathbf{Y}, \mathbf{Z}, \mathbf{S} \subseteq \mathbf{X}$ . Re-interpret the link labels as nodes (so the join tree from Figure 62 would become an undirected tree with 11 nodes and unlabelled links), obtaining a "homogenized" tree  $\mathcal{T}'$ . If for all  $Y \in \mathbf{Y}, Z \in \mathbf{Z}$  the path from any node in  $\mathcal{T}'$  containing Z to any node containing Y passes through a node whose labels include  $\mathbf{S}$ , then  $\mathbf{Y} \perp \mathbf{Z} \mid \mathbf{S}$ .

In join trees, the belief potentials are the marginal distributions of their variables (again, I don't give the proof):

**Proposition 10.3** Let  $\mathcal{T}$  be a join tree for  $P(\mathbf{X})$ , and let  $\mathbf{K} = \{X_1, \ldots, X_k\} \subseteq \mathbf{X} = \{X_1, \ldots, X_k, X_{k+1}, \ldots, X_n\}$  be a cluster or sepset label set. Then for any value instantiation  $x_1, \ldots, x_k$  of the variables from  $\mathbf{K}$ , it holds that

$$\sum_{x_{k+1}\in S_{X_{k+1}},\ldots,x_n\in S_{X_n}} P(x_1,\ldots,x_n) = \varphi_{\mathbf{K}}(x_1,\ldots,x_k),$$
(123)

that is,  $\varphi_{\mathbf{K}}$  is the marginal distribution of the variables in  $\mathbf{K}$ .

We will from now on use the shorthand notation

$$\sum_{\mathbf{X}\setminus\mathbf{K}} P(\mathbf{X}) = \varphi_{\mathbf{K}}$$
(124)

to denote marginalization.

Ok., now that we know what a join tree is, we return to Step 4: constructing a join tree from a triangulated moral UGM  $G_t$ . A join tree is specified through (i) its labelled graph structure and (ii) its belief potentials. We first treat the the question how one can derive the join tree's graph structure from the triangulated UGM  $G_t$ .

There is much freedom in creating a join tree graph from  $G_t$ . One goal for optimizing the design is that one strives to end up with clusters that are as small as possible (because the computational cost of using a join tree for inference will turn out to grow exponentially in the maximal size of clusters). On the other hand, in order to compute belief potentials later, any clique in  $G_t$  must be contained in some cluster. This suggests to turn the cliques identified in Step 3 into the cluster nodes of the join tree. This idea indeed gives a general recipe for constructing a join tree from a triangulated UGM  $G_t$ , which I rephrase from Huang and Darwiche 1994:

- 1. Begin with an empty set SEP, and a completely unconnected graph whose nodes are the m maximal cliques  $C_i$  found in Step 3.
- 2. For each distinct pair of cliques  $\mathbf{C}_i, \mathbf{C}_j$  create a candidate sepset  $\mathbf{S}_{ij} = \mathbf{C}_i \cap \mathbf{C}_j$ , and put it into SEP. SEP will then contain m(m-1)/2 such  $\mathbf{S}_{ij}$  (some of which may be the empty set).
- 3. From SEP iteratively choose m-1 sepsets and use them to create connections in the node graph, such that each newly chosen sepset connects two subgraphs that were previously unconnected. This necessarily yields a tree structure.

A note on the word "tree structure": in most cases when a computer scientist talks about tree graphs, there is a special "root" node. Here we use a more general notion of trees to mean undirected graphs which (i) are connected (there is a path between any two nodes) and (ii) where these paths are unique, that is between any two nodes there is exactly one connecting path. In such tree graphs any node can be designated as "root" if one wishes to see the familiar appearance of a tree.

This general recipe leaves much freedom in choosing the sepsets from SEP in step 3. Not all choices will result in a valid join tree. In order to ensure the join tree properties, we choose, in every step from 3., the candidate sepset that has the largest number of contained variables.

This is not the only possible way of constructing a join tree, and it is still underspecified (there may be several maximal-size sepsets at our disposal in a step from 3.) Huang and Darwiche propose a full specification that heuristically optimizes the join tree with respect to the ensuing inference algorithms.

If the original BN was not connected, some of the sepsets used in the join tree will be empty; we get a join forest then.

We now turn to the second subtask in Step 4 and construct belief potentials  $\varphi_{\mathbf{C}}, \varphi_{\mathbf{S}}$  for the clusters and sepsets, such that Equations 121 and 122 hold. These belief potentials are constructed in two steps. First, the potentials are initialized in a way such that (122) holds. Second, by a sequence of *message passes*, local consistency (121) is achieved.

**Initialization.** Initialization works in two steps:

1. For each clique or sepset **K** (we use symbol **K** for cliques **C** or sepsets **S**), set  $\varphi_{\mathbf{K}}$  to the constant function

$$\varphi_{\mathbf{K}} \equiv 1.$$

2. For each variable X, do the following. Assign to X a clique  $\mathbf{C}_X$  that contains X and its parents  $\mathbf{\Pi}_X$  from the original BN. Due to the moralizing, such a clique must exist. Multiply  $\varphi_{\mathbf{C}_X}$  by  $P(X \mid \mathbf{\Pi}_X)$ :

$$\varphi_{\mathbf{C}_X} \leftarrow \varphi_{\mathbf{C}_X} P(X \mid \mathbf{\Pi}_X).$$

This is a shorthand notation. In more detail: first, interpret  $P(X | \Pi_X)$ as a real-valued function f of all variables contained in  $\mathbf{C}_X$ , as follows. Let  $\mathbf{C}_X = \{X_1, \ldots, X_k, X_{k+1}, \ldots, X_l\}$  where  $\mathbf{\Pi}_X = \{X_1, \ldots, X_k\}$  are the parents of  $X = X_{k+1}$ . Then set  $f(x_1, \ldots, x_k, x_{k+1}, \ldots, x_l) = P(X_{k+1} = x_{k+1} | X_1 = x_1, \ldots, X_k = x_k)$ . Second, for all value combinations  $\bar{x} = (x_1, \ldots, x_l)$ , update

$$\varphi_{\mathbf{C}_X}(\bar{x}) \leftarrow \varphi_{\mathbf{C}_X}(\bar{x}) \cdot f(\bar{x}).$$

After this initialization, the conditional distributions  $P(X_k | \mathbf{\Pi}_{X_k})$  of all variables (and hence the information from the BN) have been multiplied into the clique potentials, and (122) is satisfied:

$$\frac{\prod_{i} \varphi_{\mathbf{C}_{i}}}{\prod_{j} \varphi_{\mathbf{S}_{j}}} = \frac{\prod_{k=1,\dots,n} P(X_{k} | \mathbf{C}_{X_{k}})}{1} = \prod_{k=1,\dots,n} P(X_{k} | \mathbf{\Pi}_{X_{k}}) = P(\mathbf{X}),$$

where i ranges over all cliques, j over all sepsets, and k over all RVs.

After having initialized the join tree potentials, we make them locally consistent by propagating the information, which has been locally multiplied-in, across the entire join tree. This is done through a suite of *message passing* operations, each of which makes one clique/sepset pair consistent. We first describe a single message pass operation and then show how they can be scheduled such that a message pass does not destroy consistency of clique/sepset pairs that have been made consistent in an earlier message passing. A single message pass. Consider two adjacent cliques C and D with an intervening sepset S and their associated belief potentials  $\varphi_{C}, \varphi_{D}, \varphi_{S}$ . A message pass from C to D occurs in two steps:

1. "Projection": create a copy  $\varphi_{\mathbf{S}}^{\mathsf{old}}$  of  $\varphi_{\mathbf{S}}$  for later use, then recompute  $\varphi_{\mathbf{S}}$  by marginalization from **C**:

$$\varphi^{\mathsf{old}}_{\mathbf{S}} \leftarrow \varphi_{\mathbf{S}}, \quad \varphi_{\mathbf{S}} \leftarrow \sum_{\mathbf{C} \setminus \mathbf{S}} \varphi_{\mathbf{C}}.$$

This makes  $\varphi_{\mathbf{S}}$  consistent with  $\varphi_{\mathbf{C}}$  according to Equation 121. The joint distribution  $P(\mathbf{X})$  becomes changed through this operation by a factor of  $\varphi_{\mathbf{S}}^{\mathsf{old}}/\varphi_{\mathbf{S}}$  (notice that  $\varphi_{\mathbf{S}}$  appears in the denominator of (122)).

2. "Absorption": multiply the belief potential of **D** by the inverse of  $\varphi_{\mathbf{S}}^{\mathsf{old}}/\varphi_{\mathbf{S}}$  in order to restore the joint distribution:

$$\varphi_{\mathbf{D}} \leftarrow \varphi_{\mathbf{D}} \; \frac{\varphi_{\mathbf{S}}}{\varphi_{\mathbf{S}}^{\mathsf{old}}}.$$

A technical detail: if  $\varphi_{\mathbf{S}}^{\mathsf{old}}(\mathbf{s}) = 0$ , it can been shown that also  $\varphi_{\mathbf{S}}(\mathbf{s}) = 0$ ; in this case set  $\varphi_{\mathbf{S}}(\mathbf{s})/\varphi_{\mathbf{S}}^{\mathsf{old}}(\mathbf{s}) = 0$ .

After this step,  $\mathbf{C}$  is consistent with  $\mathbf{S}$  in the sense of (121). To also make  $\mathbf{D}$  consistent with  $\mathbf{S}$ , a message passing in the reverse direction must be carried out. An obvious condition is that this reverse-direction pass must preserve the consistency of  $\mathbf{C}$  with  $\mathbf{S}$ . This is warrented if a certain order of passes is observed, to which we now turn our attention.

Coordinating all message passes. In order to achieve local consistency (121) for all neighboring clique-sepset pairs in the join tree, twice as many message passes as there are such pairs must be executed, two for each pair. In order to avoid that local consistency for a pair C, S achieved by a message pass from C to D (where S is the sepset between C and D) is not destroyed by a subsequent message pass in the reverse direction, the global order of these passes is crucial. We will motivate a global propagation scheme by considering some connection C – S – D within the tree, as depicted in Figure 63.



Figure 63: A connection  $\mathbf{C} - \mathbf{S} - \mathbf{D}$  within the tree.

This connection will be hit twice by a message pass, one in each direction. Assume that the first of the two passes went from **C** to **D**. After this pass, we have potentials  $\varphi_{\mathbf{C}}^0, \varphi_{\mathbf{S}}^0, \varphi_{\mathbf{D}}^0$ , and **C** is consistent with **S**:

$$\varphi^0_{\mathbf{S}} = \sum_{\mathbf{C} \setminus \mathbf{S}} \varphi^0_{\mathbf{C}}$$

At some later time, another message pass sweeps back from **D** to **C**. Before this happens, the potential of **D** might have been affected by some other passes, so it is  $\varphi_{\mathbf{D}}^1$  when the pass from **D** to **C** occurs. After this pass, we have

$$\varphi_{\mathbf{S}}^{1} = \sum_{\mathbf{D} \setminus \mathbf{S}} \varphi_{\mathbf{D}}^{1}$$
 and  $\varphi_{\mathbf{C}}^{1} = \varphi_{\mathbf{C}}^{0} \frac{\varphi_{\mathbf{S}}^{1}}{\varphi_{\mathbf{S}}^{0}}$ .

It turns out that still **S** is consistent with **C**:

$$\varphi_{\mathbf{S}}^{1} = \varphi_{\mathbf{S}}^{0} \; \frac{\varphi_{\mathbf{S}}^{1}}{\varphi_{\mathbf{S}}^{0}} = \left(\sum_{\mathbf{C} \setminus \mathbf{S}} \varphi_{\mathbf{C}}^{0}\right) \; \frac{\varphi_{\mathbf{S}}^{1}}{\varphi_{\mathbf{S}}^{0}} = \sum_{\mathbf{C} \setminus \mathbf{S}} \left(\varphi_{\mathbf{C}}^{0} \; \frac{\varphi_{\mathbf{S}}^{1}}{\varphi_{\mathbf{S}}^{0}}\right) = \sum_{\mathbf{C} \setminus \mathbf{S}} \varphi_{\mathbf{C}}^{1}.$$

In summary, we see that if a connection  $\mathbf{C} - \mathbf{S} - \mathbf{D}$  is hit by two passes, one for each direction,  $\mathbf{S}$  will be consistent with  $\mathbf{C}$  and  $\mathbf{D}$ . In order to ensure consistency for all connections in the tree, we must make sure that after some connection  $\mathbf{C} - \mathbf{S} - \mathbf{D}$  has been passed back and forth, neither  $\mathbf{C}$  nor  $\mathbf{D}$  take part in any further passes, as this might again disrupt the already achieved consistency. The following global scheduling scheme assures this condition:

- 1. To start, single out any clique **C** and call it the "center".
- 2. In a first phase ("collect evidence" in the Huang/Darwiche paper), carry out all passes that are oriented towards the center. Carry out these passes in any order such that a pass "leaves" some node on some connection only after all other "incoming" connections have been used for passes.
- 3. In a second phase ("distribute evidence"), carry out all passes that are oriented away from the center, in an inside-out spreading order.

Figure 64 shows a possible global scheduling for our example join tree.

After all of this toiling, we have a magic join tree  $\mathcal{T}$  — a tree graph adorned with belief potentials that are locally consistent (Equation 121) and globally represent the joint distribution  $P(\mathbf{X})$  (Equation 122). The join tree is now ready for use in inference tasks.



Figure 64: A scheduling for the global propagation of message passing. The center is ACE. Figure taken from the Huang/Darwiche paper.

Inference task 1: compute a single marginal distribution. For any RV X in the BN, computing the marginal distribution P(X) is a simple two-step procedure:

- 1. Identify a clique or sepset  $\mathbf{K}$  which contains X.
- 2. Obtain P(X) by marginalization

$$P(X) = \sum_{\mathbf{K} \setminus \{X\}} \varphi_{\mathbf{K}}.$$

This follows from Proposition 10.3.

**Inference task 2: Compute conditional distributions.** Most use-cases of BNs concern the calculation of probabilities for certain events, given that some information is known / measured / observed. For instance, in the Space Shuttle example, one wants to know the probability that the engine will explode given certain sensor readings; or in medical diagnostic one wants to know the probability that the patient has cancer given lab results.

In formal terms, this means to calculate conditional distributions of the kind  $P(Z \mid E_1 = e_1, \ldots, E_k = e_k)$ , where  $Z, E_1, \ldots, E_k$  are variables in the BN, and current values for the  $E_i$  are known. The known facts  $E_i = e_i$  are called *evidence*.

Let  $e_1, \ldots, e_k = \mathbf{e}$  be an evidence, that is values for RVs  $E_1, \ldots, E_k = \mathbf{E}$ . To feed this information into  $\mathcal{T}$ , we introduce a new kind of belief potentials called (in this context) *likelihoods*. For each  $E \in \mathbf{E}$  we define the likelihood  $\Lambda_E : S_E \to \{0, 1\}$  by

$$\Lambda_E(e) = \begin{cases} 1 & \text{if } e \text{ is the observed value of } E\\ 0 & \text{for all other } e \in S_E \end{cases}$$

In order to compute  $P(Z | E_1 = e_1, \ldots, E_k = e_k) = P(Z | \mathbf{e})$ , we have to go through the routine of initializing  $\mathcal{T}$  and making it consistent via global message propagation again, using an augmented version of the method described previously. Here is how.

**Initialization:** Exactly as described above. This yields initial belief potentials  $\varphi_{\mathbf{C}}, \varphi_{\mathbf{S}}$  for all cliques and sepsets.

**Observation entry:** for each  $E \in \mathbf{E}$ , do the following:

- 1. Identify a clique  $\mathbf{C}_E$  which contains E.
- 2. Update  $\varphi_{\mathbf{C}_E} = \varphi_{\mathbf{C}_E} \Lambda_E$ .

Here  $\varphi_{\mathbf{C}_E} \Lambda_E$  is a shorthand for the function  $\varphi_{\mathbf{C}_E} \Lambda_E : S_E \times S_{A_1} \times \ldots \times S_{A_l} \to \mathbb{R}^{\geq 0}$  defined by

$$\varphi_{\mathbf{C}_E} \Lambda_E(\tilde{e}, a_1, \dots, a_l) = \varphi_{\mathbf{C}_E}(\tilde{e}, a_1, \dots, a_l) \cdot \Lambda_E(\tilde{e}),$$

where  $\mathbf{C}_E$  has labels  $(E, A_1, \ldots, A_l)$ , that is,  $\varphi_{\mathbf{C}_E} \Lambda_E$  simply resets  $\varphi_{\mathbf{C}_E}$  to zero for all arguments that have a different value for E than the observed one. With the new potentials, the tree globally encodes  $P(\mathbf{X}) \mathbf{1}_{\mathbf{e}}$ , where  $\mathbf{1}_{\mathbf{e}} : S_{\bigotimes X_i} \to \{0, 1\}$  is the indicator function of the set  $\{(x_1, \ldots, x_n) \in S_{\bigotimes X_i} \mid x_i = e_j \text{ for } X_i = E_j, j = 1, \ldots, k\}$ :

$$\frac{\prod_i \varphi_{\mathbf{C}_i}}{\prod_j \varphi_{\mathbf{S}_j}} = P(\mathbf{X}) \Lambda_{E_1} \dots \Lambda_{E_k} = P(\mathbf{X}) \mathbf{1}_{\mathbf{e}} =: P(\mathbf{X}, \mathbf{e}).$$

Note furthermore that

$$\sum_{\mathbf{X}\setminus\mathbf{E}} = P(\mathbf{e}).$$

Global propagation: Global propagation: exactly as before, but starting from the updated potentials obtained by observation entry. After this is completed, the join tree is locally consistent. Furthermore, each clique or sepset K has a potential satisfying

$$\varphi_{\mathbf{K}} = P(\mathbf{K}, \mathbf{e}).$$

**Normalization:** In order to compute  $P(Z | \mathbf{e})$ , determine a clique  $\mathbf{C}_Z$  which contains Z. When we marginalize this clique's potential to Z, we obtain the probability of Z and  $\mathbf{e}$ :

$$\sum_{\mathbf{C}_Z \setminus \{X\}} \varphi_{\mathbf{C}_Z} = P(Z, \mathbf{e}).$$

But our goal is to compute  $P(Z|\mathbf{e})$ , the distribution of Z given  $\mathbf{e}$ . We obtain this by normalizing  $P(Z|\mathbf{e})$ :

$$P(Z \mid \mathbf{e}) = \frac{P(Z, \mathbf{e})}{P(\mathbf{e})} = \frac{P(Z, \mathbf{e})}{\sum_{z \in S_Z} P(Z, \mathbf{e})}$$

#### 10.1.3 Learning Bayesian networks

We have seen how we can calculate statistical inferences on the basis of a given BN. But who gives it to us, and how?

A common approach, especially in AI, is to distil the conditional probability tables from interviews with experts of the target domain. Domain experts often have clear conceptions of local conditional probabilities that connect a few variables. You might ask, why use a BN if such experts are available? The answer is that while humans may have a good insight on local interdependencies between a few variables, they are psychologically and intellectually poorly equipped to use this local knowledge for making sound statistical inferences which connect many variables in a globally connected causal interaction system.

In other cases, empirical observations are available that allow one to estimate the conditional probability tables from data. For instance, the table shown in Equation (116) could have been estimated from counts # of observed outcomes as suggested in Figure 65.



Figure 65: Example of estimating a probability table from frequency counts.

Notice that each row in such a table is estimated independently from the other rows. The task of estimating such a row is the same as estimating a discrete distribution from data. When there is no abundance of data, Bayesian model estimation via Dirichlet priors is the way to go, as in the protein frequency estimation in Section 8.3.

When data like in Figure 65 are available for all nodes in a BN, estimating the local conditional probabilities by the obvious frequency counting ratios gives maximum likelihood estimates of the local conditional probabilities. It can be shown that this is also the maximum likelihood estimate of the joint distribution  $P(\mathbf{X})$  (not a deep result, the straightforward derivation can be found in the online lecture notes of Ermon 2019, chapter "Learning in directed models").

It is very often the case that for some of the variables, neither empirical observations nor an expert's opinion is available, either because simply the observations have not been carried out or because these quantities are in principle unobservable. Such unobservable variables are called *hidden variables*. To get an impression of the nature and virtues of hidden variables, consider the BN in Figure 66.

Maritial status is hardly causal for political preferences. According to some



Figure 66: A BN for use by a social worker (apologies to professionals in the field)



Figure 67: A BN for use by a psychologically and statistically enlightened social worker

(folk) psychology, the BN shown in Figure 67 would more appropriately capture the causal logics connecting the variables from the model in Figure 66.

However, nobody has yet found a convincing way to directly measure selfconfidence — it is an *explanatory* concept, and becomes a hidden variable in a statistical model. While all other variables in this BN can be readily measured, self-confidence can't. Yet the augmented BN is, in an intuitive sense, more valuable than the first one, because it tries to reveal a causal mechanism whereas the former one only superficially connects variables by arrows that can hardly be understood.

Besides being intellectually more pleasing, the second BN offers substantial computational savings: its join tree (construct it!) is much more lightweight than the first BN's, so statistical inference algorithms will run much faster.

Generalizing from this simplistic example, it should be clear that hidden variables are a great asset in modeling reality. But — they are hidden, which means that the requisite probability tables cannot be directly estimated from empirical data.

When there are hidden variables in a BN for whose conditional distribution no data are available, one uses EM algorithms to estimate their probability tables. The basic version of EM for BNs which seems standard today has been introduced in a paper by Lauritzen 1995. The algorithm is also described in a number of online tutorials and open access papers, for instance Mouafo et al. 2016. The algorithm is complex. In the E-step it uses inference in join trees as a subroutine.

# 10.2 Undirected graphical models

Undirected graphical models (UGMs) play a much larger role than just helping out as intermediate representations on the way from a BN to a join tree. UGMs have been (re-)discovered independently in different fields and under different names, for instance as

- Markov random fields, a quite generic concept/word used in statistics but also in a more special sense in image processing;
- Ising models, a class of models of 2D or 3D magnetic / spintronic / quantumdynamical materials explored in statistical physics,
- Boltzmann machines, a kind of stochastic, recurrent neural networks which give a wonderfully elegant and powerful model of a universal learning system, unfortunately, computationally intractable,
- restricted Boltzmann machines, a stripped-down version of the former, computationally tractable, and instrumental in kicking off the deep learning revolution (Hinton and Salakuthdinov 2006).

I originally planned to have a section on these models, exploring their deep connection to physics and giving a demo in an image denoising task. But the BN part of this section has grown sooooo long that I think it would be too much for one session of this course. If you are interested — the online course notes of Ermon 2019 include an easy-reading fast introduction to Markov random fields in the chapter with the same name, and my legacy lecture notes on "Statistical and Algorithmical Modeling" (https://www.ai.rug.nl/minds/uploads/LN\_AlgMod.pdf) have a substantial Section 6.3 on UGMs.

# 10.3 Hidden Markov models

I also once had planned to introduce *hidden Markov models* (HMMs) in this session. HMMs are models of stochastic processes with memory — *the* standard model of such processes. A modern view of HMMs is to see them as graphical models, and since they are designed around hidden RVs (hence their name), they are trained with a dedicated and efficient version of EM. All of this would be very nice to have but Christmas is approaching and we should better be starting to think about stopping to think about machine learning for a while. If you are very motivated — the classical tutorial text on HMMs is Rabiner 1990.

HMMs are models of stochastic processes — they describe how observations made in the past change conditional probabilities of things that may be observed in the *future*. HMMs use DAGs, like Bayesian networks, but with HMMs, there are causal arrows that mean causation in time: the causing RV  $X_n$  is earlier than the influenced RV  $X_{n+1}$ . This leads to potentially infinite DAGs (if time just runs on), and connects graphical models to the theory of stochastic processes. In recent years, directed graphical models which contain arrows that mean timesteps and which are more complex than HMMs have become studied. A pioneer is Kevin Murphy who more or less established this field with his PhD thesis (K. P. Murphy 2002) — another case of starting a stellar career by tying together all sorts of existing loose ends in one unifying, tutorial work.

# 11 Online adaptive modeling

Often an end-user of a machine learning model finds himself/herself in a situation where the distribution of the input data changes over time. A learnt model will then become inaccurate because it was trained with training data that had another distribution of the input data. It would be desirable if the model, once trained initially, could continue learning while being used and *adapt* itself to the new statistics of the data it has to process. Two examples:

- A speech recognition system on a smartphone is used while walking in a city and doing some shopping the background noise will change every few seconds and the recognition system has to adapt its "de-noising" continuously to changing sorts of noise.
- A credit risk prediction system is used month after month but when an economical crisis changes the loantakers' payback morale, the system should change its prediction biases.

Never-ending adaptation of machine learning systems has always been an issue for machine learning. Currently this theme is receiving renewed attention in deep learning in a special, somewhat restricted version, where it is discussed under the headline of *continual learning* (or *continuous learning*). Here one seeks solutions to the problem that, if an already trained neural network is subsequently trained even more on new incoming training data, the previously learnt competences will be destructively over-written by the process of continuing learning on new data. This phenomenon is known since long as *catastrophic forgetting*. Methods for counter-acting catastrophic forgetting are developing fast. If you are interested — the PhD thesis of Xu He (He 2023), added to Brightspace underneath the lecture notes, starts with a very readable overview on today's continual learning landscape.

In this section I will however not deal with continual (deep) learning, for two reasons: (i) this material is advanced and requires substantial knowledge of deep learning methods, (ii) the currently available methods still fall short of the desired goal to enable ongoing, "life-long" learning.

Instead, I will present methods which have since long been explored and successfully used in the field of signal processing and control. This material is not normally treated in machine learning courses — I dare say, mostly because machine learners are just not aware of this body of knowledge, and maybe also if they are, they find these methods "too linear" (no neural networks involved!). But I find this material most valuable to know,

• because these techniques are broadly applicable, especially in application contexts that involve signal processing and control — like robotics, industrial engineering applications, and modeling biological systems including brains;

- because these techniques are mathematically elementary and transparent and give you a good understanding of conditions when gradient descent optimization of loss functions becomes challenged — it's a perfect primer to get into the learning algorithms of deep learning — you will understand better why they sometimes become veeeeery slow or numerically instable;
- and finally, because I think that machine learning is an interdisciplinary enterprise and I believe that these signal processing flavored methods will become important in a young and fast-growing field of research called *neuro-morphic computing* which has a large overlap with machine learning my own field since a few years.

Throughout this section, I am guided by the textbook *Adaptive Filters: The*ory and *Applications* (Farhang-Boroujeny 1998), which has been my steadfast companion since I bought it back in 1999. An updated edition came out 2013.

### 11.1 The adaptive linear combiner

In this subsection I introduce basic terminology and notation and the fundamental system model that I will discuss throughout this section, the *adaptive linear combiner*.

I start with a refresher on systems and signals terminology. A discrete-time, real-valued *signal* is a left-right infinite, real-valued sequence  $\mathbf{x} = (x(n))_{n \in \mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}}$ (we will only consider discrete-time, real-valued signals). Note that in this section,  $\mathbf{x}$  refers to a complete sequence of values; if we want to single out a particular signal value at time n, we write x(n). I will often use the shorthand (x(n)) for  $(x(n))_{n \in \mathbb{Z}}$ .

A filter (or system — this word is a synonym for "filter" in the world of signal processing) H is a mapping from signals to signals, written  $\mathbf{y} = H(\mathbf{x})$ . The signal processing folks like to put this in a diagram where the filter is a box, see Figure 68.



Figure 68: Diagram representation of a filter (or system) H.

The signal  $\mathbf{x}$  is the *input* signal and  $\mathbf{y}$  is the *output* signal of H.

There are arbitrarily complex filters — only think of a filter where the input signals are (sampled versions of) microphone recordings of an English speaker and the output is synthesized speech of an online Dutch translation of the input signal. In this section we will restrict ourselves to a particularly simple, yet immensely useful class of filters called *transversal filters*. A transversal filter generates the output by a linear transform of the L preceding input values, where L is the *length* of the transversal filter:

**Definition 11.1** Let  $L \ge 1$  be an integer, and let  $\mathbf{w} \in \mathbb{R}^L$  be a vector of filter weights. The transversal filter  $H_{\mathbf{w}}$  is the filter which transforms an input signal  $\mathbf{x} = (x(n))$  into the output  $\mathbf{y} = H_{\mathbf{w}}(\mathbf{x})$  defined by

$$y(n) = \mathbf{w}' (x(n), x(n-1), \dots, x(n-L+1))'.$$
(125)

The input vector  $(x(n), x(n-1), \ldots, x(n-L+1))'$  is advantageously padded with a constant 1 'bias' signal. This enables the filter to produce outputs y(n)that have arbitrary shifts away from the zero line. We have seen a similar trick in the linear regression rehearsal (Section 3.1). In fact, training a transversal filter on a least mean square loss (which is how we will do it below) amounts to computing a linear regression.

Note that here we consider the vector  $\mathbf{w}$  as a column vector (often in the literature, this specific weight vector is understood as row vector — as we did in our earlier treatment of linear regression). Transversal filters are linear filters. A filter H is called *linear* if for all  $a, b \in \mathbb{R}$  and signals  $\mathbf{x}_1, \mathbf{x}_2$ 

$$H(a\mathbf{x}_1 + b\mathbf{x}_2) = a H(\mathbf{x}_1) + b H(\mathbf{x}_2).$$
(126)

The proof that transversal filters are linear is an easy exercise.

I remark in passing that the theory of signals and systems works with complex numbers throughout both for signals and filter parameters; for us it is however good enough if we only use real-valued signals and model parameters.

The unit impulse  $\delta = (\delta(n))$  is the signal that is zero everywhere except for n = 0, where  $\delta(0) = 1$ . The unit impulse response of a filter H is the signal  $H(\delta)$ . For a transversal filter  $H_{\mathbf{w}}$ , the unit impulse response is the signal which repeats  $\mathbf{w} = (w_1, \ldots, w_L)'$  at times  $n = 0, 1, \ldots, L - 1$ :

$$(H_{\mathbf{w}}(\delta))(n) = \begin{cases} w_i & \text{if } n = i - 1 \ (i = 1, \dots, L), \\ 0 & \text{else.} \end{cases}$$
(127)

Figure 69 shows the structure of a transversal filter in the graphical notation used in signal processing.

The theme of this section is online adaptive modeling. In the context of filters, this means that the filter changes over time in order to continue complying with new situational conditions. Concretely, we consider scenarios of supervised training adaptation, where a teacher output signal is available. We denote this "desirable" filter output signal by (d(n)). The objective of online adaptive filtering is to continually adapt the filter weights  $w_i$  such that the filter output  $y(n) = \mathbf{w}'(x(n), \ldots, x(n-L+1))'$  stays close to the desired output d(n).

The situational conditions may change both with respect to the input signal x(n), which might change its statistical properties, and/or with respect to the teacher d(n), which might also change. For getting continual optimal performance this implies that the filter weights must change as well: the filter weight vector **w** 



Figure 69: A transversal filter (black parts) and an adaptive linear combiner (blue parts). Boxes labeled  $z^{-1}$  are what signal processing people call "unit delays" — elementary filters which delay an input signal by one timestep. The triangular boxes mean "multiply with". In such box-and-arrow flow diagrams in the field of signal processing, diagonal arrows spearing through some box indicate that what is in the box is becoming *adapted* on the basis of the information that arrives with the arrow.

becomes a temporal variable  $\mathbf{w}(n)$ . Using the shorthand  $\mathbf{x}(n) = (x(n), \ldots, x(n - L + 1))'$  for the last L inputs up to the current x(n), this leads to the following online adaptation task:

Given at time n: the filter weight  $\mathbf{w}(n-1)$  calculated at the end of the previous timestep; new input and desired output data points x(n), d(n).

Compute error at time *n*:  $\varepsilon(n) = d(n) - \mathbf{w}'(n-1)\mathbf{x}(n)$ .

Adaptation goal: Compute new filter weights

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \Delta_{\mathbf{w}}(n)$$

such that the squared error

$$\varepsilon^2(n+1) = (d(n+1) - \mathbf{w}'(n)\mathbf{x}(n+1))^2$$

at the *next* (!) timestep can be expected to be smaller than without the adaptation. Notes:

• At the time of computing  $\mathbf{w}(n)$ , the next input/target pair x(n+1), d(n+1) is not yet known. The filter weights  $\mathbf{w}(n)$  can only be optimized with regards to the *expected* next input/target pair x(n+1), d(n+1).
- Because the future signals x and d at time n + 1 and later are not yet known, the best one can do is to assume that the x and d signals in the next timesteps will follow the same rules as during the known past  $x(n), d(n), x(n-1), d(n-1), \ldots$  up to the present moment. The filter weights  $\mathbf{w}(n)$  will (have to) be adapted such that the known squared errors that would be obtained in the past, namely  $\varepsilon^2(n-l) = (d(n-l) - \mathbf{w}'(n)\mathbf{x}(n-l))^2$  (where l = 0, 1, 2, ...), are minimized on average. Since the input-output rule may be changing with time, only a limited-horizon past can be used to minimize the average past error. We will later see that the best option is to use a discounted past: the filter weights  $\mathbf{w}(n)$  will be optimized to minimize more recent squared errors more strongly than squared errors that lie further in the past. This will automatically happen if the modification  $\Delta_{\mathbf{w}}(n)$  is based only on the current error  $\varepsilon(n) = d(n) - \mathbf{w}'(n-1)\mathbf{x}(n)$ . Information from earlier errors will already be incorporated in  $\mathbf{w}(n-1)$ . This leads to an error feedback based weight adaptation which is schematically shown in the blue parts of Figure 69. A diagonal arrow through a box is the way how parameter adaptation is depicted in such signal processing diagrams. Such continually adapted transversal filters are called *adaptive linear combiners* in the Farhang/Boroujeny textbook.
- Notice that this task setting is entirely similar to the supervised learning task that we know from the ML context. The input-teacher pairs  $(x(n), d(n)), (x(n-1), d(n-1)), \ldots$  up to the current time n are known training data, and the future input-teacher pairs  $(x(n + 1), d(n + 1)), (x(n + 2), d(n + 2)), \ldots$  are test data that are not known at time n when the weights have to be updated. All the difficulties that we are now so familiar with are again present a main challenge is to avoid overfitting and underfitting and find the weight updates that will best generalize to new data. To this familiar challege is added the enormous new difficulty that we have a 'moving target': the statistics of the learning task changes as times goes on.
- The error signal  $\varepsilon^2(n)$  which the adaptive filter tries to keep low need not always be obtained by comparing the current model output  $\mathbf{w}(n-1)'\mathbf{x}(n)$  with a teacher d(n). Often the "error" signal is obtained in other ways than by comparison with a teacher. In all cases, the objective for the adaptation algorithm is to keep the "error" amplitude at low levels. The error signal itself is the only source of information to steer the adaptation (compare Figure 69). Several examples in the next subsection will feature "error" signals which are not computed by comparison with a teacher signal.

## 11.2 Basic applications of adaptive linear combiners

Before explaining concrete algorithms for weight adaptation I want to highlight the importance and versatility of adaptive linear combiners with a choice of application examples, mostly taken from the Farhang/Boroujeny textbook.

#### 11.2.1 System identification

This is the most basic task: reconstruct from  $\mathbf{x}$  and teacher  $\mathbf{d}$  a filter ("model system", "system model", "identification model") whose output  $\mathbf{y}$  approximates  $\mathbf{d}$ . This kind of task is called *system identification*. A schematic block diagram for this kind of application is shown in Figure 70.



Figure 70: Schema of the system identification task.

The following game is played here. Some target system is monitored while it receives an input signal  $\mathbf{x}$  and generates an output  $\mathbf{g}$  which is observed with added observation noise  $\nu$ ; this observed signal output becomes the teacher  $\mathbf{d}$ . The model system is a transversal filter  $H_{\mathbf{w}}$  whose weights are adapted such that the model system's output  $\mathbf{y}$  stays close to the observed output  $\mathbf{d}$  of the target system.

Obtaining and maintaining a model system  $H_{\mathbf{w}}$  is a task which occurs ubiquitously in systems engineering. The model system can be used for manifold purposes because it allows to *simulate* the target system. Such simulation models are needed, for instance, for making predictions about the future behavior of the original system, or for assessing whether the target system might be running into malfunction modes. Almost every *control* or *predictive maintenance* task in electric or mechanical engineering requires system models. I illustrate the use of system models with two concrete examples taken from the Farhang/Boroujeny book.

**Example 1: Geological exploration.** In geological prospecting, before one digs expensive holes, one wishes to obtain an idea of what's what underground. Figure 71 shows the basic principle of exploring the structure of the ground under the earth surface. At some point A, the earth surface is excited by a strong acoustic



Figure 71: Geological exploration via impulse response of learnt earth model. A. Physical setup. B. Analysis of impulse response.

signal (explosion or large vibrating mass). An earth microphone is placed at a distant point B, picking up a signal **d**. A model  $H_{\mathbf{w}}$  (a "dummy earth") is learnt. This model captures how the (simulated) earth transmits the acoustic signal from the generator to the microphone, - capturing all the delays and deformations that the input signal has suffered when it arrives at the microphone (communication engineers would call this earth model a *channel* model). After the model  $H_{\mathbf{w}}$  has been learnt, one analyses the impulse response  $\mathbf{w}$  of this model (which, as we have briefly mentioned earlier, in these simple transversal filters happens to coincide with the tap weight vector). The peaks of  $\mathbf{w}$  give indications about reflecting layers in the earth crust between A and B, which correspond to different delayed responses  $p_i$  of the input impulse.

**Example 2:** Adaptive open-loop control. In general terms, an *open-loop* (or *direct* or *inverse* or *feedforward*) controller is a filter that generates an input signal **u** into a system (called "plant" in control engineering) such that the system output **y** follows a reference (or target) signal **r** as closely as possible. For example, when the plant is an electric motor which one desires to run at a rotation speed (r(n)), the task for an open-loop controller is to generate a voltage signal (u(n)) such that the motor's actual speed (y(n)) comes close to the target (r(n)).

motors are nonlinear and have inertia effects of mass acceleration and magnetic field build-up which delay the speed response to input voltage changes, this is not necessarily a simple task.

One way to achieve this objective is to maintain a transversal filter model  $\hat{H} = H_{\mathbf{w}}(n)$  of the plant H by online weight adaptation and analytically compute an *inverse filter*  $H_{\mathbf{w}}^{-1}(n)$  for  $H_{\mathbf{w}}(n)$ . The inverse  $G^{-1}$  of a filter G is a filter which cancels the effects of G, that is, for any input signal  $\mathbf{x}$ , one has  $G^{-1}(G(\mathbf{x})) = \mathbf{x}$ . Figure 72 shows the idea.

Note that the inverse of a transversal filter  $H_{\mathbf{w}}$  is not itself a transversal filter — it will be a filter that needs its own output fed back as additional input; we will not consider such filters in this course.



Figure 72: Schema of online adaptive direct control.

#### 11.2.2 Inverse system identification

This is the second most basic task: given an unknown system H which on input **d** produces output **x**, learn an inverse system that on input **x** produces output **d** (note the reversal of variable roles and names). A typical setup is shown in Figure 73.



Figure 73: Schema of inverse system identification.

This is an alternative to the analytical inversion of an estimated system model. Introducing the delay  $z^{-\Delta}$  is not always necessary but typically improves stability of the learnt system. Equalization of a communication channel. A prime application of inverse system modeling is in digital telecommunication, where a binary signal  $\mathbf{s}$  (a bit sequence) is distorted when it is passed through a noisy channel H, and should be un-distorted ("equalized") by passing it through an equalizing filter  $H^{-1}$  (for short, called an *equalizer*). In order to train the equalizer, the correct signal  $\mathbf{s}$  must be known by the receiver when the equalizer is trained. But of course, if  $\mathbf{s}$  would be already known, one would not need the communication in the first place. This henand-egg problem is often solved by using a predetermined, fixed training sequence  $\mathbf{s} = \mathbf{d}$ . From time to time (especially at the initialization of a transmission), the sender transmits  $\mathbf{s} = \mathbf{d}$ , which is known by the receiver and enables it to estimate an inverse channel model. But also while useful communication is taking place, the receiver can continue to train its equalizer, as long as the receiver is successful in restoring the binary signal  $\mathbf{s}$ : in that case, the correctly restored signal  $\mathbf{s}$  can be used for continued training. The overall setup is sketched in Figure 74.



Figure 74: Schema of adaptive online channel equalization. Delays (which one would insert for stability) are omitted. The box on the right with the step function indicates a filter that converts the continuous-valued equalizer output  $\mathbf{y}$  to a binary signal — assuming that we want a channel for binary bitstream signals.

#### Feedback error learning for a composite direct / feedback controller.

Pure open-loop control cannot cope with external disturbances to the plant. For example, if an electric motor has to cope with varying external loads, the resulting braking effects would remain invisible to the kind of open-loop controller shown in Figure 72. One needs to establish some sort of *feedback control*, where the observed mismatch between the reference signal  $\mathbf{r}$  and the plant output  $\mathbf{y}$  is used to insert corrective input to the plant. There are many ways to design feedback controllers.

The scheme shown in Figure 75 (proposed in Jordan and Wolpert 1999 in a nonlinear control context, using neural networks — yes, the same Michael Jordan who also contributed so much to graphical models was likewise interested in biological motor control, and many other things!) trains an open-loop inverse (feedforward) controller in conjunction with the operation of a fixed, untrainable feedback controller.



Figure 75: Schema of feedback error learning for a composite control system.

Some explanations for this ingenious architecture:

- The control input u(n) is the sum of the outputs  $u_{\rm fb}(n)$  of the feedback controller and  $u_{\rm ff}(n)$  of the feedforward controller.
- There is no teacher signal **d** in this architecture. The feedforward controller is trained on an "error" signal which is not defined as a difference between a teacher and a system output. That is just as well; the adaptation of the feedforward controller just attempts to minimize the squared "error" signal.
- If the feedforward controller works perfectly, the feedback controller detects no discrepancy between the reference r(n) and the plant output y(n) and therefore produces a zero output  $u_{\rm fb}(n)$  that is, the feedforward controller sees zero error  $\varepsilon$  and does not change.
- When the overall control system does not work perfectly, the feedback controller will output a nonzero control output  $u_{\rm fb}(n)$ , which acts as an error signal for adaptating of the feedforward controller. The feedforward controller tries to minimize this error — that is, it changes its way to generate output  $u_{\rm ff}(n)$  such that the feedback controller's output is minimized, that is, such that  $(r(n) - y(n))^2$  is minimized, that is, such that the control improves (an admittedly superficial explanation).
- When the plant characteristics change, or when external disturbances set in, the feedback controller jumps to action, inducing further adaptation of the feedforward controller.

## 11.2.3 Interference cancelling, "denoising"

Assume that there is a signal  $\mathbf{s} + \nu_0$  that is an additive mixture of a useful signal  $\mathbf{s}$  and a noise component  $\nu_0$ . You want to cancel the interfering component  $\nu_0$  from this mixture. Assume further that you also have another signal source  $\nu_1$  that is mainly generated by the same noise source as  $\nu_0$ , but contains only weak traces of  $\mathbf{s}$ . In this situation you may use the denoising scheme shown in Figure 76.



Figure 76: Schema of denoising filter.

Denoising has many applications, for instance in airplane cockpit crew communication (cancelling the acoustic airplane noises from pilot intercom speech), postprocessing of live concert recordings, or (like in one of the suggested semester projects) cancelling the mother's ECG signal from the unborn child's in prenatal diagnostics. In the Powerpoint file *denoisingDemo.pptx* which you find on Brightspace together with the lecture notes, you can find an acoustic demo that I once programmed.

**Explanations**:

- The "error" which the adaptive denoising filter tries to minimize is  $\mathbf{s} + \nu_0 \mathbf{y}$ , where  $\mathbf{y}$  is the filter output.
- The only information that the filter has to achieve this error minimization is its input  $\nu_1$ . Because this input is (ideally) independent of **s**, but related to  $\nu_0$  via some noise-to-noise filter, all that the filter can do is to subtract from  $\mathbf{s} + \nu_0$  whatever it finds correlates in  $\mathbf{s} + \nu_0$  with  $\nu_1$ . Ideally, this is  $\nu_0$ . Then, the residual "error"  $\hat{\mathbf{s}}$  would be just  $\mathbf{s}$  — the desired de-noised signal.
- This scheme is interesting (not just a trivial subtraction of  $\nu_1$  from  $\mathbf{s} + \nu_0$ ) because the mutual relationship between  $\nu_1$  and  $\nu_0$  may be complex, involving for instance a superposition of delayed versions of  $\nu_0$ .

More sophisticated methods for denoising are known today, often based on mathematical principles from *independent component analysis* (ICA). You find an acoustic demo on https://www.youtube.com/watch?v=tkkm6zVUDXo.

# 11.3 Iterative learning algorithms by gradient descent on performance surfaces

So far in this course, we haven't encountered iterative learning algorithms for supervised learning tasks (the EM algorithms which we already saw are iterative, but were used for the estimation of distribution models, which is an unsupervised task). The online adaptive model estimation algorithms studied in the present section, and the "backpropagation" algorithm for training neural networks presented in the next section, are the first (and only) iterative supervised learning algorithms treated in this course. Both work by *gradient descent on performance surfaces.* In this subsection I give a general explanation of what that means.

#### 11.3.1 Iterative supervised learning algorithms

A typical iterative supervised learning algorithm unfolds as follows:

**Given:** Training data  $(\mathbf{x}_i, \mathbf{y}_i)_{i=1,...,N}$  where  $\mathbf{y}_i \in S_Y$ ; a parametric family  $\mathcal{H}$  of candidate models  $h_{\theta}$  where each parameter vector  $\theta \in \mathcal{H}$  defines a possible model ("decision function"); and a loss function  $L: S_Y \times S_Y \to \mathbb{R}^{\geq 0}$ .

Wanted: An optimal model

(\*) 
$$\theta_{\mathsf{opt}} = \operatorname*{argmin}_{\theta \in \mathcal{H}} R^{\mathsf{emp}}(\theta) = \operatorname*{argmin}_{\theta \in \mathcal{H}} \frac{1}{N} \sum_{i=1,\dots,N} L(h_{\theta}(\mathbf{x}_i), \mathbf{y}_i).$$

- **But reality strikes:** The optimization problem (\*) often cannot be solved directly, for instance because it is analytically intractable (the case for neural network training) or because the training data come in as a time series and their statistical properties change with time (as in adaptive online modeling).
- Second best approach: Design an iterative algorithm which produces a sequence of models (= parameter vectors)  $\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, \ldots$  with decreasing empirical risk  $R^{\mathsf{emp}}(\theta^{(0)}) > R^{\mathsf{emp}}(\theta^{(1)}) > \ldots$  The model  $\theta^{(n+1)}$  is typically computed by an incremental modification of the previous model  $\theta^{(n)}$ . The first model  $\theta^{(0)}$  is a guess provided by the experimenter.

In neural network training, the hope is that this series converges to a model  $\theta^{(\infty)} = \lim_{n \to \infty} \theta^{(n)}$  whose empirical risk is close to the minimal possible empirical risk. In online adaptive modeling, the hope is that if one incessantly-iteratively tries to minimize the empirical risk, one stays close to the moving target of the current best model (we'll see how that works).

This scheme only treats the approach where one tries to minimize the training loss. We know that in standard supervised learning settings this invites overfitting and that one should better employ a regularized loss function together with some cross-validation procedure, a complication that we ignore here. In online adaptive modeling, the empirical risk  $R^{emp}(\theta)$  is time-varying, another complication that for the time being we will ignore. Such complications notwithstanding, the general rationale of iterative supervised learning algorithms is to compute a sequence of models with decreasing empirical risk.

In standard (not online adaptive) settings, such iterative algorithms, if they converge, can find only locally optimal models. A model  $\theta^{(\infty)}$  is locally optimal

if every slight modification of it will lead to a higher empirical risk. The final converged model  $\lim_{n\to\infty} \theta^{(n)}$  will depend on the initial guess  $\theta^{(0)}$  — coming up with a method for good initial guesses was the crucial innovation that started the deep learning revolution (Hinton and Salakuthdinov 2006).

#### **11.3.2** Performance surfaces

First a quick recap: the graph of a function. Recall that the graph of a function  $f: A \to B$  is the set  $\{(a, f(a)) \in A \times B \mid a \in A\}$  of all argument-value pairs of the function. For instance, the graph of the square function  $f: \mathbb{R} \to \mathbb{R}, x \mapsto x^2$  is the set of points in the familiar parabola curve in  $\mathbb{R}^2$ .

A *performance surface* is the graph of a risk function. Depending on the specific situation, the risk function may be the empirical risk, the risk, a risk defined at a particular time (in online adaptive scenarios), or some other "cost". I will use the symbol  $\mathcal{R}$  to denote a generic risk function of whatever kind.

Performance surfaces are typically discussed with parametric model families where each model h in a hypothesis space  $\mathcal{H}$  is identified with a k-dimensional parameter vector  $\theta$ , thus we can identify  $\mathcal{H}$  with some subset of  $\mathbb{R}^k$ . The performance surface then becomes the graph of the function  $\mathcal{R} : \mathcal{H} \to \mathbb{R}^{\geq 0}$ .

Specifically, if we have k-dimensional parameter vectors (that is,  $\mathcal{H} \subseteq \mathbb{R}^k$ ), the performance surface is a k-dimensional surface in  $\mathbb{R}^{k+1}$ .

Other terms are variously used for performance surfaces, for instance *performance landscape* or *error landscape* or *error surface* or *cost landscape* or similar wordings.

Performance surfaces can be folded in structures of stunning complexity. In neural network training (next section in these lecture notes), they are *tremendously* complex. Figure 77 shows a neural network error landscape randomly picked from the web.

## 11.3.3 Iterative model learning by gradient descent on a performance surface

Often a risk function  $\mathcal{R} : \mathcal{H} \to \mathbb{R}^{\geq 0}$  is differentiable. Then, for every  $\theta \in \mathcal{H}$  the *gradient* of  $\mathcal{R}$  with respect to  $\theta = (\theta_1, \ldots, \theta_k)'$  is defined:

$$\nabla \mathcal{R}(\theta) = \left(\frac{\partial \mathcal{R}}{\partial \theta_1}(\theta), \cdots, \frac{\partial \mathcal{R}}{\partial \theta_k}(\theta)\right)'.$$
 (128)

The gradient  $\nabla \mathcal{R}(\theta)$  is the vector which points from  $\theta$  in the direction of the steepest ascent ("uphill") of the performance surface. The negative gradient  $-\nabla \mathcal{R}(\theta)$  is the direction of the steepest descent (Figure 78).

The idea of model optimization by gradient descent is to iteratively move toward a minimal-risk solution  $\theta^{(\infty)} = \lim_{n\to\infty} \theta^{(n)}$  by always "sliding downhill" in the direction of steepest descent, starting from an initial model  $\theta^{(0)}$ . This idea is as natural and compelling as can be. Figure 79 shows one such itinerary.



Figure 77: A (2-dimensional cross-section of) a performance surface for a neural network. The performance landscape shows the variation of the loss when (merely) 2 weights in the network are varied. Source: http://www.telesens.co/2019/01/16/neural-network-loss-visualization/

The general recipe for iterative gradient descent learning goes like this:

**Given:** A differentiable risk function  $\mathcal{R} : \mathcal{H} \to \mathbb{R}^{\geq 0}$ .

Wanted: A minimal-risk model  $\theta_{opt}$ .

**Start:** Guess an initial model  $\theta^{(0)}$ .

Iterate until convergence: Compute models

$$\theta^{(n)} = \theta^{(n-1)} - \mu \,\nabla \mathcal{R}(\theta^{(n-1)}). \tag{129}$$

The adaptation rate (or learning rate)  $\mu$  is set to a small positive value.

An obvious weakness of this elegant and natural approach is that the final model  $\theta^{(\infty)}$  depends on the choice of the initial model  $\theta^{(0)}$ . In complex risk landscapes (as the one shown in Figure 77) there is no hope of guessing an initial model which guarantees to end in the global minimum. This circumstance is generally perceived and accepted. There is a substantial mathematical literature that amounts to "if the initial model is chosen with a good heuristic, the local minimum that will be reached will be a rather good one with high probability".

We will see that the real headaches with gradient descent optimization are of a different nature — specifically, there is a inherently difficult tradeoff between speed of convergence (one does not want to invest millions or even billions of iterations) and stability (the iterations must not lead to erratic large-sized jumps that lead away from the downhill direction). The adaptation rate  $\mu$  plays a key role in this difficult tradeoff. For good performance (stability plus satisfactory speed



Figure 78: A performance surface for a 2-dimensional model family with parameters  $\theta_1, \theta_2$ , with its contour plot at the bottom. For a model  $\theta$  (yellow star in contour plot) the negative gradient is shown as black solid arrow. It marks the direction of steepest descent (broken black arrow) on the performance surface.

of convergence), the adaptation rate must be adapted online while the iterations proceed. And doing that sagaciously is not trivial. The situation shown in Figure 79 is deceptively simplistic; say goodbye to any hope that gradient descent works as smoothly as that in real-life applications.

These difficulties raise their ugly head already in the simplest possible risk surfaces, namely the ones that arise with iterative solutions to linear regression problems. Making you aware of these difficulties in an analytically tractable, transparent setting is one of the two main reasons why I believe a machine learning student should know about adaptive online learning of transversal filters. You will learn a lot about the challenges in neural network training as a side effect. The other main reason is that adaptive online learning of transversal filters is really, really super useful in many practical tasks.



Figure 79: A gradient descent itinerary, re-using the contour map from Figure 78 and starting from the initial point shown in that figure. Notice the variable jump length and the sad fact that from this initial model  $\theta^{(0)}$  the global minimum is missed. Instead, the itinerary slides toward a local minimum at  $\theta^{(\infty)}$ . The blue arrows show the negative gradient at raster points. They are perpendicular to the contour lines and their length is inversely proportional to the spacing between the contour lines.

## 11.3.4 The performance surface of a stationary online learning task for transversal filters

We will now return to the specific scenario of online adaptive filtering. Recall that we are facing a left-right infinite signal of real-valued input data  $(x(n))_{n\in\mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}}$ paired with a desired output signal  $(d(n))_{n\in\mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}}$ . Our goal is to compute, at every time *n* and on the basis of the input and teacher signals up to time *n*, an *L*dimensional weight vector  $\mathbf{w}(n)$  which we want to minimize the expected squared error  $\varepsilon^2_{\mathbf{w}(n)}(n+1) = (d(n+1) - \mathbf{w}'(n)\mathbf{x}(n+1))^2$  at the next timestep. For starters we will work under the assumption that the input and teacher

For starters we will work under the assumption that the input and teacher statistics does not change with time. In mathematical terms, this means that the signals  $\mathbf{x}$  and  $\mathbf{d}$  are generated by a *stationary* stochastic process. That is, these signals come from random variables  $(X_n)_{n\in\mathbb{Z}}$  and  $(D_n)_{n\in\mathbb{Z}}$ . Each time that we would "run" this process, we would get another pair of signals  $\mathbf{x}, \mathbf{d}$ . To define what "stationary" means in our context, let

$$[X^{L}D]_{n} := X_{n} \otimes X_{n-1} \otimes \dots \otimes X_{n-L+1} \otimes D_{n}$$
(130)

be the combined RV which picks, at time n, the previous L inputs up to and including time n and the teacher at time n. We now say that the process is *stationary* with respect to these RVs  $[X^LD]_n$ , if  $[X^LD]_k$  and  $[X^LD]_m$  have the same distribution for all times k and m. Intuitively speaking, this means that if we look at signal sniplets  $\mathbf{x}(k), d(k)$  and  $\mathbf{x}(m), d(m)$  at different times m and k, the distribution of these sniplets across different runs of the stochastic process will be the same at both times. The process does not change its statistical properties over time. If this condition is met, the expectations

$$\mathbf{p} := E[\mathbf{x}(n) d(n)] = (E[x(n)d(n)], \dots, E[x(n-L+1)d(n)])'$$
(131)

$$\mathbf{R} := E[\mathbf{x}(n) \ \mathbf{x}'(n)], \tag{132}$$

where the expectation is taken with respect to time, are well-defined and independent of n. **R** is called (in the field of signal processing) the *correlation matrix* of the input process; it has size  $L \times L$ . **p** is an *L*-dimensional vector.

In a stationary process there is no need to have different filter weights at different times. We can thus, for now, drop the time dependence from  $\mathbf{w}(n)$  and consider unchanging weights  $\mathbf{w}$ . For any such weight vector  $\mathbf{w}$  we consider the expected squared error

$$R(\mathbf{w}) = E[\varepsilon_{\mathbf{w}}^{2}(n)]$$

$$= E[((d(n) - \mathbf{w}' \mathbf{x}(n)) ((d(n) - \mathbf{w}' \mathbf{x}(n))]$$

$$= E[(d(n))^{2}] - 2 \mathbf{w}' E[\mathbf{x}(n) d(n)] + \mathbf{w}' E[\mathbf{x}(n) \mathbf{x}'(n)] \mathbf{w}$$

$$= E[(d(n))^{2}] - 2 \mathbf{w}' \mathbf{p} + \mathbf{w}' \mathbf{R} \mathbf{w}.$$
(133)

This expected squared error for a filter  $H_{\mathbf{w}}$  is the risk that we want to minimize. We now take a close look at the performance surface, that is at the graph of the risk function  $R : \mathbb{R}^L \to \mathbb{R}$ . Its geometrical properties are the key for mastering the adaptive filtering task. Figure 80 gives a visual impression of the performance surface for the case of L = 2 dimensional weight vectors  $\mathbf{w} = (w_1, w_2)'$ .



Figure 80: The performance surface in the case of two-dimensional weight vectors (black parts of this drawing taken from drip.colorado.edu/~kelvin/links/ Sarto\_Chapter2.ps many years ago, page no longer online). An iterative algorithm for weight determination would try to determine a sequence of weights  $\ldots, \mathbf{w}^{(n)}, \mathbf{w}^{(n+1)}, \mathbf{w}^{(n+2)}, \ldots$  (green) that moves toward  $\mathbf{w}_{opt}$  (blue). The eigenvectors  $\mathbf{u}_j$  (red) of the correlation matrix  $\mathbf{R}$  lie on the principal axes of the hyperellipsiods given by the level curves of the performance surface.

I mention without proof some basic geometric properties of the performance surface. The function (133) is a (multidimensional) quadratic function of  $\mathbf{w}$ . The general form of a multidimensional quadratic function  $F: \mathbb{R}^k \to \mathbb{R}$  for kdimensional vectors **x** is  $F(\mathbf{x}) = a + \mathbf{x'b} + \mathbf{x'Cx}$ , where  $a \in \mathbb{R}$  is a constant offset,  $\mathbf{x'b}$  is the linear term, and  $\mathbf{x'Cx}$  is the quadratic term. C must be a positive definite matrix or negative definite matrix. The graph of a multidimensional quadratic function shares many properties with the graph of the one-dimensional quadratic function  $f(x) = a + bx + cx^2$ . The one-dimensional quadratic function graph has the familiar shape of a parabola, which depending on the sign of c is opened upwards or downwards. Similarly, the graph of k-dimensional quadratic function has the shape of a k-dimensional paraboloid, which is opened upwards if C is positive definite and opened downwards if C is negative definite. A kdimensional paraboloid is a "bowl" whose vertical cross-sections are parabolas. The contour curves of a k-dimensional paraboloid (the horizontal cross-sections) are k-dimensional ellipsoids whose main axes lie in the directions of the k orthogonal eigenvectors  $\mathbf{u}_1, \ldots, \mathbf{u}_k$  of  $\mathbf{C}$ .

In the 2-dimensional case of a performance surface shown in Figure 80, the paraboloid must open upwards because the risk R is an expected squared error and hence cannot be negative; in fact, the entire surface must be nonnegative. The figure also shows a projection of the ellipsoid contour curves of the paraboloid on the  $w_1$ - $w_2$  plane, together with the eigenvectors of  $\mathbf{R}$ .

Importantly, such quadratic performance surfaces have a single minimum (provided that **R** has full rank, which we tacitly take for granted). This minimum marks the position of the optimal (minimal risk) weight vector  $\mathbf{w}_{opt} = \underset{\mathbf{w} \in \mathbb{R}^L}{\operatorname{argmin} R(\mathbf{w})}$ . An iterative learning algorithm would generate a sequence  $\mathbf{w} \in \mathbb{R}^L$ ...,  $\mathbf{w}^{(n)}, \mathbf{w}^{(n+1)}, \mathbf{w}^{(n+2)}, \ldots$  of weight vectors where each new  $\mathbf{w}^{(n)}$  would be positioned at a place where the performance surface is deeper than at the previous weight vector  $\mathbf{w}^{(n-1)}$  (green dots in the figure).

The optimal weight vector  $\mathbf{w}_{opt}$  can be calculated analytically, if one exploits the fact that the partial derivatives  $\partial R / \partial w_i$  are all zero (only) for  $\mathbf{w}_{opt}$ . The gradient  $\nabla R$  is given by

$$\nabla R = \left(\frac{\partial R}{\partial w_1}, \cdots, \frac{\partial R}{\partial w_L}\right)' = 2 \mathbf{R} \mathbf{w} - 2 \mathbf{p}$$

which follows from (133) (exercise). Setting this to zero gives the *Wiener-Hopf* equation

$$\mathbf{R}\mathbf{w}_{\mathsf{opt}} = \mathbf{p},\tag{134}$$

which yields the optimal weights as

$$\mathbf{w}_{\mathsf{opt}} = \mathbf{R}^{-1} \, \mathbf{p}. \tag{135}$$

If you consider the definitions of  $\mathbf{R}$  and  $\mathbf{p}$  (Equations 132 and 131), you will find that this solution for  $\mathbf{w}_{opt}$  is just the solution for the linear regression problem that we already saw much earlier in Equation (19), and which we here have derived using calculus.

In order to appreciate the challenges inherent in iterative gradient descent on quadratic performance surfaces we have to take a closer look at the shape of the hyperparaboloid.

First we use (133) and the Wiener-Hopf equation to express the expected residual error  $R_{\min}$  (see Figure 80) that we are left with when we have found  $\mathbf{w}_{opt}$ :

$$R_{\min} = E[(d(n))^{2}] - 2 \mathbf{w}_{opt}' \mathbf{p} + \mathbf{w}_{opt}' \mathbf{R} \mathbf{w}_{opt}$$
  

$$= E[(d(n))^{2}] - \mathbf{w}_{opt}' \mathbf{p}$$
  

$$= E[(d(n))^{2}] - \mathbf{w}_{opt}' \mathbf{R} \mathbf{w}_{opt}$$
  

$$= E[(d(n))^{2}] - \mathbf{p}' \mathbf{R}^{-1} \mathbf{p}.$$
(136)

For a more convenient analysis we rewrite the error function R in new coordinates such that it becomes centered at the origin. Observing that the paraboloid is centered on  $\mathbf{w}_{opt}$ , that it has "elevation"  $R_{min}$  over the weight space, and that the shape of the paraboloid itself is determined by  $\mathbf{w}'_{opt} \mathbf{R} \mathbf{w}_{opt}$ , we find that we can rewrite (133) as

$$R(\mathbf{v}) = R_{\min} + \mathbf{v}' \,\mathbf{R} \,\mathbf{v},\tag{137}$$

where we introduced shifted (and transposed) weight coordinates  $\mathbf{v} = (\mathbf{w} - \mathbf{w}_{opt})$ . Differentiating (137) with respect to  $\mathbf{v}$  yields

$$\frac{\partial R}{\partial \mathbf{v}} = \left(\frac{\partial R}{\partial v_1}, \dots, \frac{\partial R}{\partial v_L}\right)' = 2 \,\mathbf{R} \,\mathbf{v}.$$
(138)

Since **R** is positive semi-definite, its SVD factorization is  $\mathbf{R} = \mathbf{U}\mathbf{D}\mathbf{U}' = \mathbf{U}\mathbf{D}\mathbf{U}^{-1}$ , where the columns of **U** are made from *L* orthonormal eigenvectors of **R** and **D** is a diagonal matrix containing the corresponding eigenvalues (which are real and non-negative) on its diagonal. Note that the eigenvectors  $\mathbf{u}_j$  of **R** lie on the principal axes of the hyperellipsoid formed by the contour lines of the performance surface (see Figure 80, red arrows).

By left-multiplication of the shifted coordinates  $\mathbf{v} = (\mathbf{w} - \mathbf{w}_{opt})$  with  $\mathbf{U}'$  we finally get *normal* coordinates  $\tilde{\mathbf{v}} = \mathbf{U}'\mathbf{v}$ . The coordinate axes of the  $\tilde{\mathbf{v}}$  system are in the direction of the eigenvectors of  $\mathbf{R}$ , and (138) becomes

$$\frac{\partial R}{\partial \tilde{\mathbf{v}}} = 2 \,\mathbf{D} \,\tilde{\mathbf{v}} = 2 \,(\lambda_1 \tilde{v}_1, \dots, \lambda_L \tilde{v}_L)',\tag{139}$$

from which we get the second derivatives

$$\frac{\partial^2 R}{\partial \tilde{\mathbf{v}}^2} = 2 \left( \lambda_1, \dots, \lambda_L \right)', \tag{140}$$

that is, the eigenvalues of  $\mathbf{R}$  are (up to a factor of 2) the curvatures of the performance surface in the direction of the central axes of the hyperparabeloid. We will shortly see that the computational efficiency of gradient descent on the performance surface depends critically on these curvatures.

#### 11.3.5 Gradient descent on a quadratic performance surface

Now let us do an iterative gradient descent on the performane surface, using the normal coordinates  $\tilde{\mathbf{v}}$ . The generic gradient descent rule (129) here is

$$\tilde{\mathbf{v}}^{(n+1)} = \tilde{\mathbf{v}}^{(n)} - \mu \,\nabla R(\tilde{\mathbf{v}}^{(n)}),$$

which spells out (observing (139)) to

$$\tilde{\mathbf{v}}^{(n+1)} = (\mathbf{I} - 2\mu \mathbf{D}) \,\tilde{\mathbf{v}}^{(n)}.\tag{141}$$

Because  $\mathbf{I} - 2\mu \mathbf{D}$  is a diagonal matrix, this gradient descent operates on the *L* coordinates of  $\tilde{\mathbf{v}}$  without mutual coupling, yielding for every individual coordinate a separate update rule

$$\tilde{v}_j^{(n+1)} = (1 - 2\mu\lambda_j) \, \tilde{v}_j^{(n)}.$$
 (142)

This is a geometric sequence. If started in  $\tilde{v}_i^{(0)}$ , one obtains

$$\tilde{v}_j^{(n)} = (1 - 2\mu\lambda_j)^n \, \tilde{v}_j^{(0)}.$$
(143)

If we would define and carry out gradient descent in the original weight coordinates  $\mathbf{w}^{(n)}$  via  $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \mu \nabla R(\mathbf{w}^{(n)})$ , we would get the same model sequence (up to the coordinate change  $\mathbf{w} \leftrightarrow \tilde{\mathbf{v}}$ ) as given by Equations 141, 142 and 143. The sequence  $\mathbf{w}^{(n)}$  converges to  $\mathbf{w}_{opt}$  if and only if  $\tilde{v}_j^{(n)}$  converges for all coordinates  $1 \leq j \leq L$ . Equation 143 implies that this happens if and only if  $|1 - 2\mu\lambda_j| < 1$  for all j. These inequalities can be re-written equivalently as

$$0 < \mu < 1/\lambda_j \quad \text{for all } j. \tag{144}$$

Specifically, we must make sure that  $0 < \mu < 1/\lambda_{\max}$ , where  $\lambda_{\max}$  is the largest eigenvalue of R. Depending on the size of  $\mu$ , the convergence behavior of (141) can be grouped in four classes which may be referred to as *overdamped*, *underdamped*, and two types of *unstable*. Figure 81 illustrates how  $\tilde{v}_j(n)$  evolves in these four classes.

We can find an explicit representation of  $\mathbf{w}^{(n)}$  if we observe that

$$\mathbf{w}^{(n)} = \mathbf{w}_{\mathsf{opt}} + \mathbf{v}^{(n)} = \mathbf{w}_{\mathsf{opt}} + \sum_{j} \mathbf{u}_{j} \, \tilde{v}_{j}^{(n)},$$

where the  $\mathbf{u}_j$  are again the orthonormal eigenvectors of **R**. Inserting (143) gives us

$$\mathbf{w}^{(n)} = \mathbf{w}_{\mathsf{opt}} + \sum_{j=1,\dots,L} \tilde{v}_j^{(0)} \, \mathbf{u}_j \, (1 - 2\mu\lambda_j)^n.$$
(145)

This representation reveals that the convergence of  $\mathbf{w}^{(n)}$  toward  $\mathbf{w}_{opt}$  is governed by an additive overlay of L exponential terms, each of which describes convergence in the direction of the eigenvectors  $\mathbf{u}_j$  and is determined in its convergence speed by  $\lambda_j$  and the stepsize parameter  $\mu$ . One speaks of the L modes of convergence with geometric ratio factors  $(1 - 2\mu\lambda_j)$ . If all eigenvalues are roughly equal, convergence rates are roughly identical in the L directions. If however two eigenvalues are very different, say  $\lambda_1 \ll \lambda_2$ , and  $\mu$  is small compared to the eigenvalues, then convergence in the direction of  $\mathbf{u}_1$  will be much slower than in the direction of  $\mathbf{u}_2$  (see Figure 82).

Next we turn to the question how the error R evolves over time. Recall from (137) that  $R(\mathbf{v}) = R_{\min} + \mathbf{v}' \mathbf{R} \mathbf{v}$ , which can be re-written as  $R(\mathbf{v}) = R_{\min} + \tilde{\mathbf{v}}' \mathbf{D} \tilde{\mathbf{v}}$ . Thus the error in the *n*-th iteration is

$$R^{(n)} = R_{\min} + \tilde{\mathbf{v}}^{\prime(n)} \mathbf{D} \, \tilde{\mathbf{v}}^{(n)} = R_{\min} + \sum_{j} \lambda_j \, (1 - 2\mu\lambda_j)^{2n} \tilde{v}_j^{(0)}. \tag{146}$$



Figure 81: The development of  $\tilde{v}_j^{(n)}$  [plotted in the *y*-axis] versus *n* [*x*-axis]. The qualitative behaviour depends on the stepsize parameter  $\mu$ . **a**. Overdamped case:  $0 < \mu < 1/(2\lambda_j)$ . **b**. Underdamped case:  $1/(2\lambda_j) < \mu < 1/\lambda_j$ . **c**. Unstable with  $\mu < 0$  and **d**. unstable with  $1/\lambda_j < \mu$ . All plots start with  $\tilde{v}_j^{(0)} = 1$ .

For suitable  $\mu$  (considering (144)),  $R^{(n)}$  converges to  $R_{\min}$ . Plotting  $R^{(n)}$  yields a graph known as *learning curve*. Equation 146 reveals that the learning curve is the sum of L decreasing exponentials (plus  $R_{\min}$ ).

How this learning curve looks like depends on the size of  $\mu$  relative to the eigenvalues  $\lambda_j$ . If  $2\mu\lambda_j$  is close to zero for all j, the learning curve separates into sections that each are determined by the convergence of one of the j components. Figure 83 shows a three-mode learning curve for the case of small  $\mu$ , rendered in linear and logarithmic scale.

This separation of the learning curve into approximately linear sections (in logarithmic rendering) can be mathematically explained as follows. Each of the terms  $(1 - 2\mu\lambda_j)^{2n}$  is characterized by a time constant  $\tau_j$  according to

$$(1 - 2\mu\lambda_j)^{2n} = \exp(-n/\tau_j).$$
(147)

If  $2\mu\lambda$  is close to zero,  $\exp(2\mu\lambda)$  is close to  $1 + 2\mu\lambda$  and thus  $\log(1 - 2\mu\lambda) \approx -2\mu\lambda$ . Using this approximation, solving (147) for  $\tau_j$  yields for the *j*-th mode a time constant of

$$\tau_j \approx \frac{1}{4\mu\lambda_j}.$$

That is, the convergence rate (i.e. the inverse of the time constant) of the *j*-th mode is proportional to  $\lambda_j$  (for small  $\mu$ ).



Figure 82: Two quite different modes of convergence (panel **a**.) versus rather similar modes of convergence (panel **b**.). Plots shows contour lines of performance surface for two-dimensional weights  $\mathbf{w} = (w_1, w_2)$ . Violet dotted lines indicate some initial steps of weight evolution.



Figure 83: A learning curve with three modes of convergence, in linear (**a**.) and logarithmic (**b**.) scaling. This plot shows the qualitative behavior of modes of convergence when  $\mu$  is small.  $R_{\min}$  is assumed zero in these plots.

However, this analysis is meaningless for larger  $\mu$ . If we want to maximize the speed of convergence, we should use significantly larger  $\mu$  as we will presently see. The final rate of convergence is dominated by the slowest mode of convergence, which is characterized by the geometrical sequence factor

$$\max\{|1 - 2\mu\lambda_j| \mid j = 1, \dots, L\} = \max\{|1 - 2\mu\lambda_{\max}|, |1 - 2\mu\lambda_{\min}|\}.$$
 (148)

In order to maximize convergence speed, the learning rate  $\mu$  should be chosen such that (148) is minimized. Elementary considerations reveal that this minimum is attained for  $|1 - 2\mu\lambda_{max}| = |1 - 2\mu\lambda_{min}|$ , which is equivalent to

$$\mu_{\mathsf{opt}} = \frac{1}{\lambda_{\mathsf{max}} + \lambda_{\mathsf{min}}}.\tag{149}$$

For this optimal learning rate,  $1 - 2\mu_{opt}\lambda_{min}$  is positive and  $1 - 2\mu_{opt}\lambda_{max}$  is negative, corresponding to the overdamped and underdamped cases shown in Figure 81. However, the two modes converge at the same speed (and all other modes are faster). Concretely, the optimal speed of convergence is given by the geometric

factor  $\beta$  of the slowest mode of convergence,

$$\beta = 1 - 2\mu_{\rm opt}\lambda_{\rm min} = \frac{\lambda_{\rm max}/\lambda_{\rm min} - 1}{\lambda_{\rm max}/\lambda_{\rm min} + 1},$$

which can be derived by substituting (149) for  $\mu_{opt}$ .  $\beta$  has a value between 0 and 1. There are two extreme cases: if  $\lambda_{max} = \lambda_{min}$ , then  $\beta = 0$  and we have convergence in a single step. As the ratio  $\lambda_{max}/\lambda_{min}$  increases,  $\beta$  approaches 1 and the convergence slows down toward stillstand. The ratio  $\lambda_{max}/\lambda_{min}$  thus plays a fundamental role in limiting the convergence speed of steepest descent algorithms. It is called the *eigenvalue spread*.

All of the derivations in this subsection were done in the context of quadratic performance surfaces. This may seem a rather limited class of performance landscapes. However, in almost all differentiable performance surfaces, the shape of the surface in the vicinity of a local minimum can be approximated by a quadradic surface. The stability conditions and the speed of convergence toward local minima in generic gradient descent situations therefore is typically subject to the same limiting conditions that we saw in this subsection.

Specifically, the eigenvalue spread (of the quadratic approximation) around a local minimum sets stability limits to the achieveable rate of convergence. It is a sad fact that this eigenvalue spread is typically large in neural networks with many layers — today famously known as deep networks. For almost twenty years (roughly, from 1985 when gradient descent training for neural networks became widely adopted, to 2006 when deep learning started) no general, good ways were known to achieve neural network training that was simultaneously stable and exhibited a fast enough speed of convergence. Only shallow neural networks (typically with a single hidden layer) could be effectively trained, limiting the applications of neural network modeling to only mildly nonlinear tasks. In own research (Jaeger et al. 2007) we once numerically investigated curvatures of performance landscapes in recurrent neural networks and found eigenvalue spreads in the order of  $10^{19}$ , which would demand learning rates smaller than  $10^{-19}$ ! The deep learning revolution is based, among other factors, on an assortment of "tricks of the trade" to overcome the limitations of large eigenvalue spreads by clever modifications of gradient descent, which cannot work in its pure form. If you are interested — Section 8 in the deep learning bible (I. Goodfellow, Bengio, and Courville 2016) is all about these refinements and modifications of, and alternatives to, pure gradient descent.

## 11.4 Stochastic gradient descent with the LMS algorithm

The update formula  $\mathbf{w}^{(n)}$  via  $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \mu \nabla R(\mathbf{w}^{(n)})$  for steepest gradient descent is not applicable in practice because the gradient  $\nabla R$  of the expected squared error  $R(\mathbf{w}^{(n)}) = E[\varepsilon_{\mathbf{w}^{(n)}}^2]$  is not known. Given filter weights  $\mathbf{w}^{(n)}$ , we need to estimate  $E[\varepsilon_{\mathbf{w}^{(n)}}^2]$ . At first sight, one would need to estimate an expected

squared error by collecting information over time — namely, to observe the ongoing filtering with weights  $\mathbf{w}^{(n)}$  for some time and then approximate  $E[\varepsilon_{\mathbf{w}^{(n)}}^2]$  by averaging over the errors seen in this observation interval. But we don't have this time — because we want to be efficient and update  $\mathbf{w}$  at every time step; and furthermore, in online model estimation scenarios the error statistics may change over time.

One ruthless way out of this impasse is to just use the *momentary* squared error as an approximation to its *expected* value, that is, use the brutal approximation

$$R^{(n)} \approx \varepsilon_{\mathbf{w}^{(n)}}^2(n) := (d(n) - \mathbf{w}^{\prime(n)} \mathbf{x}(n))^2.$$
(150)

The gradient descent update formula  $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \mu \nabla R(\mathbf{w}^{(n)})$  then becomes

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \mu \,\nabla \varepsilon_{\mathbf{w}^{(n)}}^2(n), \tag{151}$$

where we can calculate  $\nabla \varepsilon_{\mathbf{w}^{(n)}}^2(n)$  as follows:

$$\nabla \varepsilon_{\mathbf{w}^{(n)}}^{2}(n) = 2 \varepsilon_{\mathbf{w}^{(n)}}(n) \nabla \varepsilon_{\mathbf{w}^{(n)}}(n) 
= 2 \varepsilon_{\mathbf{w}^{(n)}}(n) \left( \frac{\partial \varepsilon_{\mathbf{w}^{(n)}}(n)}{\partial w_{1}}, \dots, \frac{\partial \varepsilon_{\mathbf{w}^{(n)}}(n)}{\partial w_{L}} \right) 
= -2 \varepsilon_{\mathbf{w}^{(n)}}(n) \left( \frac{\partial \mathbf{w}'^{(n)} \mathbf{x}(n)}{\partial w_{1}}, \dots, \frac{\partial \mathbf{w}'^{(n)} \mathbf{x}(n)}{\partial w_{L}} \right) 
[use \varepsilon_{\mathbf{w}^{(n)}}(n) = d(n) - \mathbf{w}'^{(n)} \mathbf{x}(n)] 
= -2 \varepsilon_{\mathbf{w}^{(n)}}(n) (x(n), \dots, x(n-L+1)) 
= -2 \varepsilon(n) \mathbf{x}(n), \quad (152)$$

where in the last step we simplified the notation  $\varepsilon_{\mathbf{w}^{(n)}}(n)$  to  $\varepsilon(n)$ . Inserting this into (151) gives

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + 2\,\mu\,\varepsilon(n)\,\mathbf{x}(n),\tag{153}$$

which is the weight update formula of one of the most compact, cheap powerful and widely used algorithms I know of. It is called the *least means squares* (LMS) algorithm in signal processing, or *Widrow-Hoff* learning rule in neuroscience where the same weight adaptation rule has been (re-)discovered independently from the signal processing tradition. In fact, this online weight adaptation algorithm for linear regression has been independently discovered and re-discovered many times in many fields.

For completeness, here are all the computations needed to carry out one full step of online filtering and weight adaptation with the LMS algorithm:

1.	read in input and compute output:	$y(n) = \mathbf{w}^{\prime(n)}  \mathbf{x}(n),$
	(inputs $\mathbf{x}$ are best padded with a 1)	
2.	compute current error:	$\varepsilon(n) = d(n) - y(n),$
3.	compute weight update:	$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + 2\mu\varepsilon(n)\mathbf{x}(n).$

One fact about the LMS algorithm should always be kept in mind: being a stochastic version of steepest gradient descent, the LMS algorithm inherits the problems connected with the eigenvalue spread of the input process  $X_n$ . If its eigenvalue spread is very large, the LMS algorithm will not work satisfactorily.

Because of its eminent usefulness (if the input vector correlation matrix has a reasonably small eigenvalue spread), the LMS algorithm has been analysed in minute detail. I conclude this section by reporting the most important insights without mathematical derivations. At the same time I introduce some of the standard vocabulary used in the field of adaptive signal processing.

For starters, we again assume that  $[X^L D]$  is a stationary processes (recall (130)). The evolution  $\mathbf{w}^{(n)}$  of weights is now also a stochastic process, because the LMS weight update depends on the stochastic vectors  $\mathbf{x}(n)$ . One interesting question is how fast the LMS algorithm converges in comparison with the ideal steepest gradient descent algorithm  $\tilde{\mathbf{v}}^{(n+1)} = (\mathbf{I}-2\mu\mathbf{D})\,\tilde{\mathbf{v}}^{(n)}$  from (141). Because we now have a stochastic update, the vectors  $\tilde{\mathbf{v}}^{(n)}$  become random variables and one can only speak about their *expected* value  $E[\tilde{\mathbf{v}}^{(n)}]$  at time n. (Intuitive explanation: this value would be obtained if many (infinitely many in the limit) training runs of the adaptive filter would be carried out and in each of these runs, the value of  $\tilde{\mathbf{v}}^{(n)}$ . The following can be shown (using some additional assumptions, namely, that  $\mu$  is small and that the signal (x(n)) has no substantial autocorrelation for time spans larger than L):

$$E[\tilde{\mathbf{v}}^{(n+1)}] = (I - 2\mu \mathbf{D}) E[\tilde{\mathbf{v}}^{(n)}].$$
(154)

Rather to our surprise, if the LMS algorithm is used, the weights converge — on average across different trials — as fast to the optimal weights as when the ideal algorithm (141) is employed. Figure 84 shows an overlay of the deterministic development of weights according to (141) with one run of the stochastic gradient descent using to the LMS algorithm.

The fact that on average the weights converge to the optimal weights by no means implies that  $R^{(n)}$  converges to  $R_{\min}$ . To see why, assume that at some time n, the LMS algorithm actually would have found the correct optimal weights, that is,  $\mathbf{w}^{(n)} = \mathbf{w}_{opt}$ . What would happen next? Well, due to the random weight adjustment, these optimal weights would become misadjusted again in the next time step! So the best one can hope for asymptotically is that the LMS algorithms lets the weights  $\mathbf{w}^{(n)}$  jitter randomly in the vicinity of  $\mathbf{w}_{opt}$ . But this means that the effective best error that can be achieved by the LMS algorithm in the asymptotic average is not  $R_{\min}$  but  $R_{\min} + R_{excess}$ , where  $R_{excess}$  comes from the random scintillations of the weight update. It is intuitively clear that  $R_{excess}$  depends on the stepsize  $\mu$ . The larger  $\mu$ , the larger  $R_{excess}$ . The absolute size of the excess error is not so interesting as is the ratio  $\mathcal{M} = R_{excess}/R_{\min}$ , that is the relative size of the excess error compared to the minimal error. The quantity  $\mathcal{M}$  is called the *misadjustment* and describes what fraction of the *residual error*  $R_{\min} + R_{excess}$  can



Figure 84: Illustrating the similar performance of deterministic (pink) and stochastic (red) gradient descent.

be attributed to the random oscillations effected by the stochastic weight update [i.e.,  $R_{\text{excess}}$ ], and what fraction is inevitably due to inherent limitations of the filter itself [i.e.,  $R_{\min}$ ]. Notice that  $R_{\text{excess}}$  can in principle be brought to zero by tuning down  $\mu$  toward zero — however, that would be at odds with the objective of fast convergence.

Under some assumptions (notably, small  $\mathcal{M}$ ) and using some approximations (Farhang-Boroujeny, Section 6.3), the misadjustment can be approximated by

$$\mathcal{M} \approx \mu \operatorname{trace}(\mathbf{R}),$$
 (155)

where the *trace* of a square matrix is the sum of its diagonal elements. The misadjustment is thus proportional to the stepsize and can be steered by setting the latter, if trace( $\mathbf{R}$ ) is known. Fortunately, trace( $\mathbf{R}$ ) can be estimated online from the sequence (x(n)) simply and robustly [how? — easy exercise].

Another issue that one has always to be concerned about in online adaptive signal processing is stability. We have seen in the treatment of the ideal case that the adaptation rate  $\mu$  must not exceed  $1/\lambda_{max}$  in order to guarantee convergence. But this result does not directly carry over to the stochastic version of gradient descent, because it does not take into account the stochastic jitter of the gradient descent, which is intuitively likely to be harmful for convergence. Furthermore, the value of  $\lambda_{max}$  cannot be estimated robustly from few data points in a practical situation. Using again middle-league maths and several approximations, in the book of Farhang-Boroujeny the following upper bound for  $\mu$  is derived:

$$\mu \le \frac{1}{3 \operatorname{trace}(\mathbf{R})}.\tag{156}$$

If this bound is respected, the LMS algorithm converges stably.

In practical applications, one often wishes to achieve an initial convergence that is as fast as possible: this can be done by using  $\mu$  close to the stability boundary (156). After some time, when a reasonable degree of convergence has been attained, one wishes to control the misadjustment  $\mathcal{M}$ ; then one switches into a control mode where  $\mu$  is adapted dynamically according to  $\mu = \mathcal{M}/\text{trace}(\mathbf{R})$ , which follows from (155).

Up to here we have been analyzing LMS under the assumption of a stationary process. But the real working arena for LMS are nonstationary processes, where the objective is to *track* the changing statistics of the  $[X^L D]_n$  process by continually adapting the model  $\mathbf{w}^{(n)}$ . In this situation one still uses the same LMS rule (153). However, roughly speaking, the modeling error  $R^{(n)}$  is now a sum of three components:  $R^{(n)} = R_{\min}(n) + R_{\text{excess}}(n) + R_{\text{lag}}(n)$ , all of which are temporally changing. The new component  $R_{\text{lag}}(n)$  reflects the fact that iterative model adaptation always needs time to converge to a certain error level — when the signal statistics change with time, the model adaptation always lags behind the changing statistics. A rigorous analysis of this situation is beyond the scope of this course. In practice one must tune  $\mu$  online such that a good compromise is maintained between, on the one hand, making  $R_{\text{lag}}(n)$  small (which means speeding up convergence by increasing  $\mu$ , in order to not lag behind the changing input statistics too much), and on the other hand, minimizing  $R_{\text{excess}}(n)$  (which means small  $\mu$ ); all this while watching out for staying stable.

The LMS algorithm has been since 50 years or so the workhorse of adaptive signal processing and numerous refinements and variants have been developed. Here are some:

- 1. An even simpler stochastic gradient descent algorithm than LMS uses only the sign of the error in the update:  $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + 2 \mu \operatorname{sign}(\varepsilon(n))\mathbf{x}(n)$ . If  $\mu$ is a power of 2, this algorithm does not need a multiplication (a shift does it then) and is suitable for very high throughput hardware implementations which are often needed in communication technology. There exist yet other "sign-simplified" versions of LMS [cf. Farhang-Boroujeny p. 169].
- 2. Online stepsize adaptation: at every update use a locally adapted stepsize  $\mu(n) \sim 1/||\mathbf{x}||^2$ . This is called *normalized LMS* (NLMS). In practice this pure NLMS is apt to run into stability problems; a safer version is  $\mu(n) \sim \mu_0/(||\mathbf{x}||^2 + \psi)$ , where  $\mu_0$  and  $\psi$  are hand-tuned constants [Farhang-B. p. 172]. In my own experience, normalized LMS sometimes works wonders in comparison with vanilla LMS.
- 3. Include a *whitening* mechanism into the update equation:  $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + 2\mu \varepsilon(n) \mathbf{R}^{-1} \mathbf{x}(n)$ . This *Newton-LMS* algorithm has a single mode of convergence, but a problem is to obtain a robust (noise-insensitive) approximation to  $\mathbf{R}^{-1}$ , and to get that cheaply enough [Farhang-B. p. 210].
- 4. Block implementations: for very long filters (say, L > 10,000) and high update rates, even LMS may become too slow. Various computationally efficient *block LMS* algorithms have been designed in which the input stream

is partitioned into blocks, which are processed in the frequency domain and yield weight updates after every block only [Farhang-B. p. 247ff].

To conclude this section, it should be said that besides LMS algorithms there is another major class of online adaptive algorithms for transversal filters, namely *recursive least squares* (RLS) adaptation algorithms. RLS algorithms are not steepest gradient-descent algorithms. The background metaphor of RLS is not to minimize  $R_n(\mathbf{w})$  but to minimize the accumulated squared error up to the current time,  $\zeta(n) = \sum_{i=1,\dots,n} (d(i) - y(i))^2$ , so the performance surface we know from LMS plays no role for RLS. The main advantages and disadvantages of LMS vs. RLS are:

- 1. LMS has computational cost O(L) per update step, where L is filter length; RLS has cost  $O(L^2)$ . Also the space complexity of RLS is an issue for long filters because it is  $O(L^2)$ .
- 2. LMS is numerically robust when set up diligently. RLS is plagued by numerical instability problems that are not easy to master.
- 3. RLS has a single mode of convergence and converges faster than LMS, *very* much faster when the input signal has a high eigenvalue spread.
- 4. RLS is more complicated than LMS and more difficult to implement in robust, stable ways.
- 5. In applications where fast tracking of highly nonstationary systems is required, LMS may have better tracking performance than RLS (says Farhang-Boroujeny).

The use of RLS in signal processing has been boosted by the development of *fast RLS algorithms* which reach a linear time complexity in the order of O(20 L) [Farhang-B. Section 13].

Both LMS and RLS algorithms play a role in a field of recurrent neural networks called *reservoir computing* (Jaeger 2007), which happens to be one of my personal playgrounds. In reservoir computing, the training of neural networks is reduced to computing a linear regression. Reservoir computing has recently become particularly relevant for low-power microchip hardware implementations of neural networks. If time permits I will give an introduction to this emerging field in a tutorial or extra session.

# 12 Feedforward neural networks: the Multilayer Perceptron

Artificial neural networks (ANNs) have been investigated for more than half a century in two scientific domains:

- In computational neuroscience, ANNs are studied as mathematical abstractions and computer simulation models of biological neural systems. These models aim at biological plausibility and serve as a research vehicle to better understand information processing in real brains.
- In machine learning, ANNs are used for creating complex information processing architectures whose function can be shaped by training from sample data. The goal here is to solve complex learning tasks in a data engineering spirit, aiming at models that combine good generalization with highly nonlinear data transformations.

Historically these two branches of ANN research had been united. The ancestor of all ANNs, the *perceptron* of Rosenblatt 1958, was a computational model of optical character recognition (as we would say today) which was explicitly inspired by design motifs imported from the human visual system (check out Wikipedia on "perceptron"). In later decades the two branches diverged further and further from each other, despite repeated attempts to re-unite them. Today most ANN research in machine learning has more or less lost its connections to its biological origins. In this course we only consider ANNs in machine learning.

Even if we only look at machine learning, ANNs come in many kinds and variations. The common denominator for most (but not all) ANNs in ML can be summarized as follows.

- An ANN is composed of a (possibly large) number of interconnected processing units. These processing units are called "neurons" or just "units". Each such unit typically can perform only a very limited computational operation, for instance applying a fixed nonlinear function to the sum of its inputs.
- The units of an ANN are connected to each other by links called "synaptic connections" (an echo of the historical past) or just "connections" or "links".
- Each of the links is weighted with a parameter called "synaptic weight", "connection weight", or just "weight" of the link. Thus, in total an ANN can be represented as a labelled graph whose nodes are the neurons and whose edges are the links, labelled with their weights. The structure of an ANN with L units can thus be represented by its  $L \times L$  sized connection weight matrix  $\mathbf{W}$ , where the entry  $\mathbf{W}(i, j) = w_{ij}$  is the weight on the link leading from unit j to unit i. When  $w_{ij} = 0$ , then unit j has no connection to unit i. The nonzero elements in  $\mathbf{W}$  therefore determine the network's

connection graph. Often it is more convenient to split the global weight matrix in submatrices, one for each "layer" of weights. In what follows I will use the generic symbol  $\theta$  for the vector of all weights in an ANN.

- Computing with an ANN means to compute a real-valued *activation*  $x_i$  for each unit i (i = 1, ..., L). In an ANN with L units, all of these activations together are combined into an *state vector*  $\mathbf{x} \in \mathbb{R}^L$ .
- The calculation of the state vector in a feedforward ANN depends on the input and connection weights, so we may write  $\mathbf{x} = f(\mathbf{u}, \theta)$ .
- The state calculation function f is almost always *local*: the activation  $x_i$  of unit i depends only on the activations  $x_j$  of units j that "feed into" i, that is, where  $w_{ij} \neq 0$ .
- The external functionality of an ANN results from the combined local interactions between its interconnected units. Very complex functionalities may thus arise from the structured local interaction between large numbers of simple processing units. This is, in a way, analog to Boolean circuits – and indeed some ANNs can be mapped on Boolean circuits. In fact, the famous paper which can be regarded as the starting shot for computational neuroscience, A logical calculus of the ideas immanent in nervous activity (McCulloch and Pitts 1943), compared biological brains directly to Boolean circuits.
- The global functionality of an ANN is determined by the connection weights *θ*. Dramatic changes in functionality can be achieved by changing the weights in *θ*. For instance, an ANN with a given connectivity structure can serve as a recognizer of handwritten digits with some *θ*, and can serve as a recognizer of facial expressions with another *θ'* wherein only a fraction of the weights have been replaced.
- The hallmark of ANNs is that its functionality is *learnt* from training data. Most learning procedures that are in use today rely on some sort of iterative model optimization with a flavor of gradient descent.

This basic scenario allows for an immense spectrum of different ANNs, which can be set up for tasks as diverse as dimension reduction and data compression, approximate solving of NP-hard optimization problems, time series prediction, nonlinear control, game playing, dynamical pattern generation and many more.

In this course I give an introduction to a particular kind of ANNs called *feed-foward neural networks*. Often they are also called — for historical reasons – *multilayer perceptrons* (MLPs).

MLPs are used for the supervised learning of input-output tasks where both input and output come in vector formats. In such tasks the training sample is of the kind  $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\dots,N}$ , where  $\mathbf{u} \in \mathbb{R}^n$ ,  $\mathbf{y} \in \mathbb{R}^k$  are drawn from a joint distribution  $P_{U,Y}$ .

Note that in this section my notation departs from the one used in earlier sections: I now use  $\mathbf{u}$  instead of  $\mathbf{x}$  to denote input patterns, in order to avoid confusion with the network states  $\mathbf{x}$ .

The MLP is trained to produce outputs  $\mathbf{y} \in \mathbb{R}^k$  upon inputs  $\mathbf{u} \in \mathbb{R}^n$  in a way that this input-output mapping is similar to the relationships  $\mathbf{u}_i \mapsto \mathbf{y}_i$  found in the training data. Similarity is measured by a suitable loss function.

Supervised learning tasks of this kind – which we have already studied in previous sections – are generally called *function approximation* tasks or *regression* tasks. It is fair to say that today MLPs and their variations are the most widely used workhorse in machine learning when it comes to learning nonlinear function approximation models.

An MLP is a neural network structured equipped with n input units and k output units. An n-dimensional input pattern  $\mathbf{u}$  is set as the state vector for the input neurons, then the MLP does some interesting internal processing, at the end of which the k-dimensional result vector of the computation can be read from the activation of the k output units. An MLP  $\mathcal{N}$  with n input units and k output units thus instantiates a function  $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^k$ . Since this function is shaped by the synaptic connection weights  $\theta$ , one could also write  $\mathcal{N}_{\theta} : \mathbb{R}^n \to \mathbb{R}^k$  if one wishes to emphasize the dependance of  $\mathcal{N}$ 's functionality on its weights.

The learning task is defined by a loss function  $L : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^{\geq 0}$ . As we have seen before, a convenient and sometimes adequate choice for L is the quadratic loss  $L(\mathcal{N}(\mathbf{u}), \mathbf{y}) = \|\mathcal{N}(\mathbf{u}) - \mathbf{y}\|^2$ , but other loss functions are also widely used. Chapter 6.2 in the deep learning bible (I. Goodfellow, Bengio, and Courville 2016) gives advice about which loss functions should be used in which task settings.

Given the loss function, the immediate goal of training an MLP is to find a weight vector  $\theta_{opt}$  which minimizes the empirical loss, that is

$$\theta_{\text{opt}} = \underset{\theta \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L(\mathcal{N}_{\theta}(\mathbf{u}_{i}), \mathbf{y}_{i}), \qquad (157)$$

where  $\mathcal{H}$  is a set of candidate weight vectors. We have seen variants of this formula many times by now in these lecture notes! And it goes almost without saying (only "almost" because I can't say this often enough) that some method of regularization combined with some sort of cross-validation must be used to shield solutions obtained from the training error minimization (157) against overfitting.

"Function approximation" sounds dry and technical, but many kinds of learning problems can be framed as function approximation learning. Here are some examples:

Pattern recognition: inputs **u** are vectorized representations of any kind of "patterns", for example images, soundfiles, texts or customer profiles. Outputs **y** are hypothesis vectors of the classes that are to be recognized.

- Time series prediction: inputs are vector encodings of a past history of a temporal process, outputs are vector encodings of future observations of the process. Examples are stock market timeseries or weather data recordings.
- Denoising, restoration and pattern completion: inputs are patterns that are corrupted by noise or other distortions, outputs are cleaned-up or repaired or completed versions of the same patterns. Important applications can be found for instance in satellite sensing, medical imaging or audio processing.
- Data compression: Inputs are high-dimensional patterns, outputs are low-dimensional encodings which can be restored to the original patterns using a decoding MLP. The encoding and decoding MLPs are trained together.
- Process control: In control tasks the objective is to send control inputs to a technological system (called "plant" in control engineering) such that the plant performs in a desired way. The algorithm which computes the control inputs is called a "controller". Control tasks range in difficulty from almost trivial (like controlling a heater valve such that the room temperature is steered to a desired value) to almost impossible (like operating hundreds of valves and heaters and coolers and whatnots in a chemical factory such that the chemical production process is regulated to optimal quality and yield). The MLP instantiates the controller. Its inputs are target signals for the desired plant behavior, plus optionally observation data from the current plant performance. The outputs are the control actions which are sent to the plant.

Note that animal brains can be seen as controllers that generate their own targets (i.e. what the animal *wants* to do next), send control input to the plant (i.e. activation signals to the animal's own muscles) and receives feedback about the effects of these control inputs through sensor signals of all sorts.

This list should convince you that "function approximation" is a worthwhile topic indeed, and spending effort on learning how to properly handle MLPs is a good professional investment for any engineer or data analyst.

# 12.1 MLP structure

Figure 85 gives a schematic of the architecture of a vanilla MLP. It consists of several *layers* of units. Layers are numbered  $0, \ldots, K$ , where layer 0 is comprised of the input units and layer K of the output units. The number of units in layer m is  $L^m$ . The units of two successive layers are connected in an all-to-all fashion by synaptic links (arrows in Figure 85). The link from unit j in layer m - 1 to unit i in layer m has a weight  $w_{ij}^m \in \mathbb{R}$ . The layer 0 is the *input layer* and the layer K is the *output layer*. The intermediate layers are called *hidden* layers. When an



MLP is used for a computation, the *i*-th unit in layer *m* will have an *activation*  $x_i^m \in \mathbb{R}$ .

Figure 85: Schema of an MLP with K - 1 hidden layers of neurons.

From a mathematical perspective, an MLP  $\mathcal{N}$  implements a function  $\mathcal{N}$ :  $\mathbb{R}^{L^0} \to \mathbb{R}^{L^K}$ . Using the MLP and its layered structure, this function  $\mathcal{N}(\mathbf{u})$  of an argument  $\mathbf{u} \in \mathbb{R}^{L^0}$  is computed by a sequence of transformations as follows:

- 1. The activations  $x_j^0$  of the input layer are set to the component values of the  $L_0$ -dimensional input vector **u**.
- 2. For layers 0 < m < K, assume that the activations  $x_j^{m-1}$  of units in layer m-1 have already been computed (or have been externally set to the input values, in the case of m-1=0). Then the activation  $x_i^m$  is computed from the formula

$$x_i^m = \sigma \left( \sum_{j=1}^{L^{m-1}} w_{ij}^m x_j^{m-1} + w_{i0}^m \right).$$
(158)

That is,  $x_i^m$  is obtained from linearly combining the activations of the lower layer with combination weights  $w_{ij}^m$ , then adding the bias  $w_{i0}^m \in \mathbb{R}$ , then wrapping the obtained sum with the activation function  $\sigma$ . The activation function is a nonlinear, "S-shaped" function which I explain in more detail below. It is customary to interpret the bias  $w_{i0}^m$  as the weight of a synaptic link from a special *bias unit* in layer m - 1 which always has a constant activation of 1 (as shown in Figure 85).

Equation 158 can be more conveniently written in matrix form. Let  $\mathbf{x}^m = (x_1^m, \ldots, x_{L^m}^m)'$  be the activation vector in layer m, let  $\mathbf{b}^m = (w_1^m, \ldots, w_{L^m}^m)'$  be the vector of bias weights, and let  $\mathbf{W}^m = (w_{ij}^m)_{i=1,\ldots,L^m; j=1,\ldots,L^{m-1}}$  be the connection weight matrix for links between layers m-1 and m. Then (158) becomes

$$\mathbf{x}^{m} = \sigma \left( \mathbf{W}^{m} \, \mathbf{x}^{m-1} + \mathbf{b}^{m} \right), \tag{159}$$

where the activation function  $\sigma$  is applied component-wise to the activation vector.

3. The  $L^k$ -dimensional activation vector **y** of the output layer is computed from the activations of the pre-output layer m = K-1 in various ways, depending on the task setting (compare Chapter 6.2 in I. Goodfellow, Bengio, and Courville 2016). For simplicity we will consider the case of *linear* output units,

$$\mathbf{y} = \mathbf{x}^K = \mathbf{W}^K \mathbf{x}^{K-1} + \mathbf{b}^K, \tag{160}$$

that is, in the same way as it was done in the other layers, except that no activation function is applied. The output activation vector  $\mathbf{y}$  is the result  $\mathbf{y} = \mathcal{N}(\mathbf{u})$ .

The activation function  $\sigma$  is traditionally either the hyperbolic tangent (tanh) function or the *logistic sigmoid* given by  $\sigma(a) = 1/(1 + \exp(-a))$ . Figure 86 gives plots of these two S-shaped functions. Functions of such shape are often called *sigmoids*. There are two grand reasons for using sigmoids:

- Historically, neural networks were conceived as abstractions of biological neural systems. The electrical activation of a biological neuron is bounded. Applying the tanh bounds the activations of MLP "neurons" to the interval [-1, 1] and the logistic sigmoid to [0, 1]. This can be regarded as an abstract model of a biological property.
- Sigmoids introduce nonlinearity into the function  $\mathcal{N}_{\theta}$ . Without these sigmoids,  $\mathcal{N}_{\theta}$  would boil down to a cascade of affine linear transformations, hence in total would be merely an affine linear function. No nonlinear function could be learnt by such a linear MLP.

In the area of deep learning, a drastically simplified "sigmoid" is today standardly used, the *rectifier* function defined by r(a) = 0 for a < 0 and r(a) = a for  $a \ge 0$ . The rectifier has somewhat less pleasing mathematical properties compared to the classical sigmoids but can be computed much more cheaply, and it helps mitigating a nasty problem of gradient descent optimization called the "vanishing gradient" problem (not treated in our course, if you are interested in a more detailed presentation of MLPs, take a look into my lecture notes on Neural Networks). This is of great value in deep learning scenarios, where the neural networks and the training samples both are often very large and the training process requires very many evaluations of the sigmoid, and where the vanishing gradient problem is apt to strike badly.



Figure 86: The tanh (blue), the logistic sigmoid (green), and the rectifier function (red).

In intuitive terms, the operation of an MLP can be summarized as follows. After an input vector  $\mathbf{u}$  is written into the input units, a "wave of activation" sweeps forward through the layers of the network. The activation vector  $\mathbf{x}^m$  in each layer m is directly triggered by the activations  $\mathbf{x}^{m-1}$  according to (159). The data transformation from  $\mathbf{x}^{m-1}$  to  $\mathbf{x}^m$  is a relatively "mild" one: just an affine linear map  $\mathbf{W}^m \mathbf{x}^{m-1} + \mathbf{b}^m$  followed by a wrapping with the gentle sigmoid  $\sigma$ . But when several such mild transformations are applied in sequence, very complex "foldings" of the input vector  $\mathbf{u}$  can be effected. Figure 87 gives a visual impression of what a sequence of mild transformations can do. Also the term "feedforward neural network" becomes clear: the activation wave spreads in a single sweep unidirectionally from the input units to the output units.

## 12.2 Universal approximation and "deep" networks

One reason for the popularity of MLPs is that they can approximate arbitrary functions  $f : \mathbb{R}^n \to \mathbb{R}^k$ . Numerous results on the approximation qualities of MLPs have been published in the early 1990-ies. Such theorems have the following general format:

**Theorem (schematic).** Let  $\mathcal{F}$  be a certain class of functions  $f : \mathbb{R}^n \to \mathbb{R}^k$ . Then for any  $f \in \mathcal{F}$  and any  $\varepsilon > 0$  there exists an multilayer perceptron  $\mathcal{N}$  with one hidden layer such that  $||f - \mathcal{N}|| < \varepsilon$ .



Figure 87: Illustrating the power of iterating a simple transformation. The *baker* transformation (also known as *horseshoe transformation*) takes a 2-dimensional rectangle, stretches it and folds it back onto itself. The bottom right diagram visualizes a set that is obtained after numerous baker transformations (plus some mild nonlinear distortion). — Diagrams on the right taken from Savi 2016.

Such theorems differ with respect to the classes  $\mathcal{F}$  of functions that are approximated and with respect to the norms  $\|\cdot\|$  that measure the mismatch between two functions. All practically relevant functions belong to classes that are covered by such approximation theorems. In a summary fashion it is claimed that MLPs are *universal function approximators*. Again, don't let yourself be misled by the dryness of the word "function approximator". Concretely, the universal function approximation property of MLPs would spell out, for example, to the (proven) statement that any task of classifying pictures can be solved to any degree of perfection by a suitable MLP.

The proofs for such theorems are typically constructive: for some target function f and tolerance  $\varepsilon$  they explicitly construct an MLP  $\mathcal{N}$  such that  $||f - \mathcal{N}|| < \varepsilon$ . However, these constructions have little practical value because the constructed MLPs  $\mathcal{N}$  are far too large for any feasible implementation. You can find more details concerning such approximation theorems and related results in my legacy ML lecture notes https://www.ai.rug.nl/minds/uploads/LN\_ML\_Fall11.pdf, Section 8.1.

Even when the function f that one wants to train into an MLP is very complex (highly nonlinear and with many "baker folds"), it can in principle be approximated with 1-hidden-layer MLPs. However, when one employs MLPs that have many hidden layers, the required overall size of the MLP (quantified by total number of weights) is dramatically reduced (Bengio and LeCun 2007). Even for super-complex target functions f (like photographic image caption generation), MLPs of feasible size exist when enough layers are used (one of the subnetworks in the TICS system described in Section 1.2.1 used 17 hidden layers, but today's deep networks easily have 50-100 layers). This is the basic insight and motivation to consider *deep* networks, which is just another word for "many hidden layers". Unfortunately it is not at all easy to train deep networks. Traditional learning algorithms had made non-deep ("shallow") MLPs popular since the 1980-ies. But these shallow MLPs could only cope with relatively simple learning tasks. Attempts to scale up to larger numbers of hidden layers and more complex data sets largely failed, due to numerical instabilities, too slow convergence, or poor model quality. Since about 2006 an accumulation of clever "tricks of the trade", plus the availability of powerful, yet affordable, GPU hardware has overcome these hurdles. This area of training deep neural networks is now one of the most thriving fields of ML and has become widely known under the name *deep learning*.

# 12.3 Training an MLP with the backpropagation algorithm

In this section I give an overview of the main steps in training a non-deep MLP for a mildly nonlinear task — tasks that can be solved with one or two hidden layers. When it comes to unleash deep networks on gigantic training corpora for hypercomplex tasks, the basic recipes given in this section will not suffice. You would need to train yourself first in the art of deep learning, investing at least a full university course's effort (this is an advertisment of Matthia Sabatelli's *Deep Learning* course in summer). However, what you can learn in this section is a necessary minimal outfit for surviving in the wilderness of deep learning.

# 12.3.1 General outline of an MLP training project

Starting from a task specification and access to training data (given to you by your boss or a paying customer, with the task requirements likely spelled out in rather informal terms, like "please predict the load of our internet servers 3 minutes ahead"), a basic training procedure for a elementary MLP  $\mathcal{N}$  goes like follows.

- 1. Get a clear idea of the formal nature of your learning task. Do you want a model output that is a probability vector? or a binary decision? or a maximally precise transformation of the input? how should "precision" best be measured? and so forth. Only proceed with using MLPs if they are really looking like a suitable model class for your problem. MLPs are raw power cannons. If you fire them on feeble datasets you make a professional error: simpler models (like random forests) will give better result at lower cost and better controllability and interpretability!
- 2. Decide on a loss function. Go for a simple quadratic loss if you want a quick baseline solution, but be prepared to invest into other loss functions if you have enough time and knowledge (Chapter 6.2 in I. Goodfellow, Bengio, and Courville 2016).

- 3. Decide on a regularization method. For elementary MLP training, suitable candidates are adding an  $L_2$  norm regularizer to your loss function; extending the size of the training data set by adding noisy / distorted exemplars (this is called *data augmentation*); varying the network size; or *early stopping* (= stop gradient descent training a overfitting-enabled ANN when the validation error starts increasing requires continual validation during gradient descent). Demyanov 2015 is a solid guide for ANN regularization.
- 4. Think of a suitable vector encoding of the available training data, including dimension reduction if that seems advisable (it *is* advisable in all situations where the ratio *raw data dimension / size of training data set* is high).
- 5. Fix an MLP architecture. Decide how many hidden layers the MLP shall have, how many units each layer shall have, what kind of sigmoid is used and what kind of output function and loss function. The structure should be rich enough that data overfitting becomes possible and your regularization method can kick in.
- 6. Set up a cross-validation scheme in order to optimize the generalization performance of your MLPs. Recommended: implement a semi-automated k-fold cross-validation scheme which is built around two subroutines, (1) "train an MLP of certain structure for minimal training error on given training data", (2) "test MLP on validation data". If, however, you are aiming for the high end with very large datasets and very large networks, classical cross-validation is not feasible due to the cost of training just a single model; the only escape then is early stopping.
- 7. Implement the training and testing subroutines. The training will be done with a suitable version of the *error backpropagation* algorithm, which will require you to meddle with some global control parameters (like learning rate, stopping criterion, initialization scheme, and more).
- 8. Do the job. Enjoy the powers, and behold the wickedness, of ANNs.

You see that "neural network training" is a multi-faceted thing and requires you to consider all the issues that *always* jump at you in supervised machine learning tasks. An ANN will *not* miraculously give good results just because it has "neurons inside". The actual "learning" part, namely solving the optimization task (157), is only a subtask, albeit a conspicuous one because it is done with an algorithm that has risen to fame, namely the backpropagation algorithm.

### 12.3.2 The error backpropagation algorithm

ANNs are trained by an iterative optimization procedure. In the classical, elementary case, this is pure gradient descent. I repeat the setup:

- **Given:** Labelled training data in vector/vector format (obtained possibly after preprocessing raw data)  $(\mathbf{u}_i, \mathbf{y}_i)_{i=1,\dots,N}$ , where  $\mathbf{u} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^k$ . Also given: a network architecture for models  $\mathcal{N}_{\theta}$  that can be specified by *l*-dimensional weight vectors  $\theta \in \mathcal{H}$ . Also given: a loss function  $L : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^{\geq 0}$ .
- **Initialization:** Choose an initial model  $\theta^{(0)}$ . This needs an educated guess. A widely used strategy is to set all weight parameters  $w_i^{(0)} \in \theta^{(0)}$  to small random values. *Remark:* For training deep neural networks this is not good enough the deep learning field actually got kickstarted by a clever method for finding a good model initialization (Hinton and Salakuthdinov 2006).
- **Iterate:** Compute a series  $(\theta^{(n)})_{n=0,1,\dots}$  of models of decreasing empirical loss (aka training error) by gradient descent. Concretely, with

$$R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}) = \frac{1}{N} \sum_{i=1,\dots,N} L(\mathcal{N}_{\theta^{(n)}}(\mathbf{u}_i), \mathbf{y}_i)$$
(161)

denoting the empirical risk of the model  $\theta^{(n)}$ , and with

$$\nabla R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}) = \left(\frac{\partial R^{\mathsf{emp}}}{\partial w_1}(\theta^{(n)}), \dots, \frac{\partial R^{\mathsf{emp}}}{\partial w_l}(\theta^{(n)})\right)'$$
(162)

being the gradient of the empirical risk with respect to the parameter vector  $\theta = (w_1, \ldots, w_l)'$  at point  $\theta^{(n)}$ , update  $\theta^{(n)}$  by

$$\theta^{(n+1)} = \theta^{(n)} - \mu \nabla R^{\mathsf{emp}}(\mathcal{N}_{\theta^{(n)}}).$$

**Stop** when a stopping criterion chosen by you is met. This can be reaching a maximum number of iterations, or the empirical risk decrease falling under a predermined threshold, or some early stopping scheme.

Computing a gradient is a differentiation task, and differentiation is easy. With high-school maths you would be able to compute (162). However, the gradient formula that you get from math textbooks would be too expensive to compute. The *error backpropagation* algorithm (or simply "backprop" for the initiated, or even just BP if you want to have a feeling of belonging to this select community) is a specific algorithmic scheme to compute this gradient in a computationally efficient way.

The backpropagation algorithm became widely known as late as 1986 (historical remarks further below). In historical perspective it was the one and only decisive breakthrough for making ANNs practically usable. Every student of machine learning *must* have understood it in detail at least once in his/her life, even if later it's just downloaded from a toolbox in some more sophisticated fashioning. Thus, brace yourself and follow along!
Let us take a closer look at the empirical risk (161). Its gradient can be written as a sum of gradients

$$\nabla R^{\mathrm{emp}}(\mathcal{N}_{\theta}) = \nabla \left( \frac{1}{N} \sum_{i=1,\dots,N} L(\mathcal{N}_{\theta}(\mathbf{u}_{i}), \mathbf{y}_{i}) \right) = \frac{1}{N} \sum_{i=1,\dots,N} \nabla L(\mathcal{N}_{\theta}(\mathbf{u}_{i}), \mathbf{y}_{i}),$$

and this is also how it is actually computed: the gradient  $\nabla L(\mathcal{N}_{\theta}(\mathbf{u}_i), \mathbf{y}_i)$  is evaluated for each training data example  $(\mathbf{u}_i, \mathbf{y}_i)$  and the obtained N gradients are averaged.

This means that at every gradient descent iteration  $\theta^{(n)} \to \theta^{(n+1)}$ , all training data points have to be visited individually. In MLP parlance, such a sweep through all data points is called an *epoch*. In the neural network literature one finds statements like "the training was done for 120 epochs", which means that 120 average gradients were computed, and for each of these computations, N gradients for individual training example points  $(\mathbf{u}_i, \mathbf{y}_i)$  were computed.

When training samples are large — as they should be — one epoch can clearly be too expensive. Therefore one often takes resort to *minibatch* training, where for each gradient descent iteration only a subset of the total sample S is used.

The backpropagation algorithm is a subroutine in the gradient descent game. It is a particular algorithmic scheme for calculating the gradient  $\nabla L(\mathcal{N}_{\theta}(\mathbf{u}_i), \mathbf{y}_i)$  for a single data point  $(\mathbf{u}_i, \mathbf{y}_i)$ . Highschool-math calculations of this quantity incur a cost of  $O(l^2)$  (where l is the number of network weights). When l is not extremely small (it will almost never be extremely small — a few hundreds of weights will be needed for simple tasks, and easily a million for deep networks applied to serious real-life modeling problems), this cost  $O(l^2)$  is too high for practical exploits (and it has to be paid N times in a single gradient descent step!). The backprop algorithm is a clever scheme for computing and storing certain auxiliary quantities which cuts the cost down from  $O(l^2)$  to O(l).

Here is how backprop works.

1. BP works in two stages. In the first stage, called the *forward pass*, the current network  $\mathcal{N}_{\theta}$  is presented with the input  $\mathbf{u}$ , and the output  $\hat{\mathbf{y}} = \mathcal{N}_{\theta}(\mathbf{u})$  is computed using the "forward" formulas (159) and (160). During this forward pass, for each unit  $x_i^m$  that is not a bias unit and not an input unit, the quantity

$$a_i^m = \sum_{j=0,\dots,L^{m-1}} w_{ij}^m x_j^{m-1}$$
(163)

is computed – this is sometimes referred to as the *potential* of unit  $x_i^m$ ; it is its internal state before it is passed through the sigmoid.

2. Some bites of math in between. Applying the chain rule of calculus we have

$$\frac{\partial L(\mathcal{N}_{\theta}(\mathbf{u}), \mathbf{y})}{\partial w_{ij}^{m}} = \frac{\partial L(\mathcal{N}_{\theta}(\mathbf{u}), \mathbf{y})}{\partial a_{i}^{m}} \frac{\partial a_{i}^{m}}{\partial w_{ij}^{m}}.$$
(164)

Define

$$\delta_i^m = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_i^m}.$$
(165)

Using (163) we find

$$\frac{\partial a_i^m}{\partial w_{ij}^m} = x_j^{m-1}.$$
(166)

Combining (165) with (166) we get

$$\frac{\partial L(\mathcal{N}_{\theta}(\mathbf{u}), \mathbf{y})}{\partial w_{ij}^{m}} = \delta_{i}^{m} x_{j}^{m-1}.$$
(167)

Thus, in order to calculate the desired derivatives (164), we only need to compute the values of  $\delta_i^m$  for each hidden and output unit.

3. Computing the  $\delta$ 's for output units. Output units  $x_i^K$  are typically set up differently from hidden units, and their corresponding  $\delta$  values must be computed in ways that depend on the special architecture. For concreteness here I stick with the simple linear units introduced in (160). The potentials  $a_i^K$  are thus identical to the output values  $\hat{y}_i$  and we obtain

$$\delta_i^K = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial \hat{y}_i}.$$
(168)

This quantity is thus just the partial derivative of the loss with respect to the *i*-th output, which is usually simple to compute. For the quadratic loss  $L(\mathcal{N}_{\theta}(\mathbf{u}), \mathbf{y}) = \|\mathcal{N}_{\theta}(\mathbf{u}) - y\|^2$ , for instance, we get

$$\delta_i^K = \frac{\partial \|\mathcal{N}_{\theta}(\mathbf{u}) - \mathbf{y}\|^2}{\partial \hat{y}_i} = \frac{\partial \|\hat{\mathbf{y}} - \mathbf{y}\|^2}{\partial \hat{y}_i} = \frac{\partial (\hat{y}_i - y_i)^2}{\partial \hat{y}_i} = 2\,(\hat{y}_i - y_i).$$
(169)

4. Computing the  $\delta$ 's for hidden units. In order to compute  $\delta_i^m$  for  $1 \le m < K$  we again make use of the chain rule. We find

$$\delta_i^m = \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_i^m} = \sum_{l=1,\dots,L^{m+1}} \frac{\partial L(\mathcal{N}_\theta(\mathbf{u}), \mathbf{y})}{\partial a_l^{m+1}} \frac{\partial a_l^{m+1}}{\partial a_i^m}, \quad (170)$$

which is justified by the fact that the only path by which  $a_i^m$  can affect  $L(\mathcal{N}_{\theta}(\mathbf{u}), \mathbf{y})$  is through the potentials  $a_i^{m+1}$  of the next higher layer. If we

substitute (165) into (170) and observe (163) we get

$$\delta_{i}^{m} = \sum_{l=1,...,L^{m+1}} \delta_{l}^{m+1} \frac{\partial a_{l}^{m+1}}{\partial a_{i}^{m}}$$

$$= \sum_{l=1,...,L^{m+1}} \delta_{l}^{m+1} \frac{\partial \sum_{j=0,...,L^{m}} w_{lj}^{m+1} \sigma(a_{j}^{m})}{\partial a_{i}^{m}}$$

$$= \sum_{l=1,...,L^{m+1}} \delta_{l}^{m+1} \frac{\partial w_{li}^{m+1} \sigma(a_{i}^{m})}{\partial a_{i}^{m}}$$

$$= \sigma'(a_{i}^{m}) \sum_{l=1,...,L^{m+1}} \delta_{l}^{m+1} w_{li}^{m+1}.$$
(171)

This formula describes how the  $\delta_i^m$  in a hidden layer can be computed by "back-propagating" the  $\delta_l^{m+1}$  from the next higher layer. The formula can be used to compute all  $\delta$ 's, starting from the output layer (where (168) is used — in the special case of a quadratic loss, Equation 169), and then working backwards through the network in the *backward pass* of the algorithm.

When the logistic sigmoid  $\sigma(a) = 1/(1 + \exp(-a))$  is used, the computation of the derivative  $\sigma'(a_i^m)$  takes a particularly simple form. Observing that for this sigmoid it holds that  $\sigma'(a) = \sigma(a) (1 - \sigma(a))$  leads to

$$\sigma'(a_i^m) = x_i^m \left(1 - x_i^m\right).$$

Although simple in principle, and readily implemented, using the backprop algorithm appropriately is something of an art, even in basic shallow MLP training. I hint at some difficulties:

- The stepsize  $\mu$  must be chosen sufficiently small in order to avoid instabilities. But it also should be set as large as possible to speed up the convergence. We have seen the inevitable and often brutal conflict between these two requirements in Section 11. In neural networks it is not possible to provide an analytical treatment of how to set the stepsize optimally. Generally one uses adaptation schemes that modulate the learning rate as the gradient descent proceeds, using larger stepsizes in early epochs. Novel, clever methods for online adjustment of stepsize have been one of the enabling factors for deep learning. For shallow MLPs stepsizes can be fixed without much thinking in the order from 0.001 to 0.01.
- Gradient descent on nonlinear performance landscapes may sometimes be crippling slow in "plateau" areas still far away from the next local minimum where the gradient is small in all directions, for other reasons than what we have seen in quadratic performance surfaces.

• Gradient-descent techniques on performance landscapes can only find a local minimum of the risk function. This problem can be addressed by various measures, all of which are computationally expensive. Some authors claim that the local minimum problem is overrated.

The error backpropagation "trick" to speed up gradient calculations in layered systems has been independently discovered several times and in a diversity of contexts outside neural network research. Schmidhuber 2015, Section 5.5, provides a historical overview. In its specific versions for neural networks, backpropagation apparently was first described by Paul Werbos in his 1974 PhD thesis, but remained unappreciated until it was re-described in a widely read collection volume on neural networks (Rumelhart, Hinton, and Williams 1986). From that point onwards, backpropagation in MLPs developed into what today is likely the most widely used computational technique in machine learning.

A truly beautiful visualization of MLP training has been pointed out to me by Rubin Dellialisi: playground.tensorflow.org/.

Simple gradient descent, as described above, is cheaply computed but may take long to converge and/or run into stability issues. A large variety of refined iterative loss minimization methods have been developed in the deep learning field, which often combine an acceptable speed of convergence with stability. Some of them refine gradient descent, others use information from the local curvature (and are therefore called "second-order" methods) of the performance surface. The main alternatives are nicely sketched and compared at https://www.neuraldesigner. com/blog/5\_algorithms\_to\_train\_a\_neural\_network (retrieved May 2017, local copy at https://www.ai.rug.nl/minds/uploads/NNalgs.zip).

### 13 Open doors in ML — leading where?

This section is optional reading and will not be queried in exercises or exams.

The breathtaking achievements of current deep learning (DL) research may suggest that ML has reached essential maturity, and further progress will mainly be coming from further growth of computing power and data resources, not needing revolutionary new ideas. After all, the best current DL system already surpass human cognitive performance (face recognition in crowds! winning chess, Go, complex strategic computer games against human world champions!), and DL applications are broadly permeating all fields of engineering, science, finance, administration, military, and cars and fridges and wrist watches and lamplight switches!

Wrong wrong wrong. Almost all really fundamental questions about "learning" and "cognition" remain almost as unsolved and ill-understood as they were fifty years ago. The more ingenious and inventive DL researchers are, the more aware they are of the hollows in their understanding. DL pioneers of stellar calibre, close to retirement age (Yoshua Bengio) or beyond (Geoffrey Hinton), burn their brains in a final rush to find answers, in their remaining lifetime, to the questions that have been driving them for decades.

First, a brief list of some of the most visible and important shortcomings of current ML methods. These deficits are acutely felt by all leading thinkers in the field, and heavy efforts are invested in Google Deep Mind, OpenAI et al. to find ways to improve on these shortfalls (human performance highlighted in blue font):

- **Data inefficiency.** High-end "cognitive" ML solutions (like GPT3 for text generation) need vast amounts of training data, orders of magnitude more than a human ever sees in a lifetime.
- Memorizing rather than understanding. ML systems are binge learners, like overcharged students before an exam: they swallow large amounts of training data and preserve a lot of superficial memory traces, but don't understand what they have gulped down. Again, GPT3 is a good example. When queried about the size of an elephant's egg, GPT may answer that it has a diameter of 50 cm; and when GPT3 is asked to tell a longer story, the system is liable to lose track of a storyline, confabulating incoherently. A human would understand that the elephant question is a joke and give a laugh for an answer, or (if we are dealing with a person who has no sense of humor) would reject the question as making no sense. And an experienced criminal, when interrogated by the police, can tell a very long, very coherent story and the even more experienced police officer can detect even the slightest inconsistencies between what the culprit said two hours ago and what comes out of his mouth now.
- Lack of explainability. When a trained neural network outputs a result on a query input, it is unclear on what grounds this output was derived. NNs are very black boxes. To achieve *explainable AI* is a burning issue, because human-understandable explanations are often critically needed. Examples: decisions made by self-driving cars (legal liabilities!), medical dignosis support systems (life vs. death; a blind recommendation of a pill is not good enough), financial investement recommendations (investors want to have good reasons to spend millions), crime prediction or creditworthyness assessment tools (citizens should get explanations when they are pre-emptively detained or denied a credit). Human experts can usually explain why they come up with an answer to a question, or at least, what they believe are their reasons.
- Slow and brittle learning from rewards. Some of the most stunning DL successes, in particular in game playing, arise from reinforcement learning (RL, which we unfortunately did not cover in this course), that is, learning from past rewards to maximize future rewards. However, RL learning is excruciatingly slow and generalizes poorly to informal real-world applications. For

example, the seemingly simple learning task of optimizing the switching cycles of the traffic lights at a single street crossing marks the limits of what is currently achievable with RL in real-world settings. I have learnt this from an external PhD project that I am supervising. A 2-year effort of a professional developer team was needed to train a RL traffic light controller (Müller et al. 2021), which in the end was not fielded because of deficits in sensor reliability and termination of funding. A little girl who has burnt her finger on a candle just once, will not put her finger into flames thereafter — not only not into candle flames, but also not into flames from matches, torches, chimney fires.

- Weak generalization and robustness. Today's ML methods work best (often: work only) when the test data come from the same distribution as the training data. With "out of distribution" test data, very bad things can happen — recall our adversarial attack example from Section 1.2.1, or wait for the next news article on a Tesla crashing into a van that the autopilot mistook for blue sky over drivable area. Humans can drive cars safely. Well, ... provided that testosteron or alcohol level is not peaking, but that is another story.
- Limited adaptation, single-purpose performance. ML systems can be amazingly good at performing exactly the task they were trained on. But it is difficult to design ML learners that can continue to learn after having been dispatched to the end-user, carrying on to adapt themselves to new task variations. This would obviously be a highly desirable feature, for instance when a speech recognizer could adapt to a new dialect or a drunken speaker (assuming drunken speaker recordings were not in the training corpus). In the neural network world this problem has been realized since long under the name of *catastrophic forgetting*. Only in the last few years, halfway satisfactory solutions to the *continual learning* problem begin to appear (briefly introduced in Section 11). However, these solutions function only in limited scenarios where the new task variants are of the same sort as the originally learnt one — same input data format, same output data format, for instance. You and I continue to learn all throughout our lives, — for instance, you have learnt about ML in the last weeks and I have learnt that there is no blackboard in the Aletta Jacobs lecture halls. Furthermore, we are not single-purpose but general-purpose performers. Not only can we recognize faces, but also speak, sing, walk, make breakfast and funny grimaces, play guitar (some of us), and all the rest. Try to come up with a complete list for yourself: how long would that list be and how crazily diverse? It seems we can do almost *everything*!

Why, oh why, is machine learning so far away from what we humans can learn? What is the secret behind the magnificent efficiency, robustness, adaptability, versatility and creativity of human learning? If I knew the answer, I would know how to make gold from lead and brains from C++ code. All I can offer are some personal speculations about what might be the big blind spot in our view on "learning", and even more speculatively, which scientific revolutions may be lying ahead — maybe just around the next corner.

In this course I tried to draw the essential picture of ML as it is understood today. All modern ML grows from one root idea, which is elegant, simple, convincing and powerful, and which I tried to carry across in this course in ever so many repetitions and variations. This idea is that

- learning means extracting useful information from data;
- real-world data are always stochastic;
- all "information" in stochastic data sources is contained in the joint distribution of the random variables that can be observed (the visible ones) or inferred (the hidden ones) — this view of "information = properties of a probability distribution" is also the core of information theory and statistical physics;
- therefore, all machine learning models should be, and can be, understood as modeling probability distributions;
- which directly leads to our modern perception of the main challenges of ML, namely
  - finding representation formats for complex distributions that can be handled on computers — like decision trees, Bayesian networks or neural networks,
  - finding ways to estimate such representations from data, such that the estimated distributions come close to the true one — this leads to the core strategies of maximum-likelihood modeling paired with crossvalidation, or Bayesian modeling, and generally connects modern ML closely to classical statistics.

The powers of this line of thinking become apparent not only in the successes of ML, but also have left a deep imprint on cognitive and neuroscience, where many theories that try to explain human cognition and brains are expressed in statistical frameworks. A small choice of important examples:

• the *Bayesian brain* approaches (also known as, or closely related to, the paradigms of the *predictive brain* or the *free energy model* of adaptive cognition, see this Wikipedia article), which cast cognitive processing and neural processes as adaptive mechanisms that help the brain's owner to extract important and reliable information from noisy sensor data, solving the generalization problem by applying genetically evolved or individually acquired Bayesian priors;

- the *neural sampling* hypothesis which interprets the apparent stochasticity of neural spikes as an MCMC sampling mechanism for computing approximate inferences about probability distributions (Buesing et al. 2011);
- rigorous statistical models of neural mechanisms that enable reinforcement learning in animals and humans (Xie and Seung 2004).

Beyond connecting ML to cognitive and neuroscience, the statistical view on learning also provides a basis for scientific exchange across other disciplinary boundaries, in particular statistical physics, signal processing & control, information and communication theory, and optimization theory — and, of course, mathematical statistics proper. Researchers in all of theses fields share the intuition and the mathematical techniques to describe reality through probability distributions.

But — do the best known brains (still, that's ours!) actually process information on the basis of statistical inference and learning representations of distributions? Or — are our brains so good exactly because *they don't*?

A first indication that this might indeed be the case is the somewhat disturbing fact that humans are actually quite bad at statistical reasoning, as documented in the landmark paper of Tversky and Kahneman 1974 (47K Google Scholar cites, as of January 2023). Famously, in elementary diagnostic reasoning (inferring causes Y from symptoms X), students of medicine typically fail at properly factoring in the base rate P(Y) when they (should) apply Bayes' rule P(Y|X) = P(X|Y)P(Y)/P(X) to get the conditional probability P(Y|X) of causes given symptoms, which is why they tend to over-estimate the probability of a serious, but rare, illness. Good medical schools therefore have mandatory courses on medical decision making. Tversky and Kahneman present numerous other, well-documented scenarios that reveal humans as statistically challenged.

Another indication comes from a strand in cognitive science that explains human conceptual systems and reasoning as something that has grown from bodily experience and physical action in physical environments. I am a great fan of this view and would love to give you a sparkling, full account of this tradition, but that would mean an entire separate course. I must make do with a few links to standard references. If you are interested, check out

- the book of Port and Gelder 1995, "Mind as Motion: Explorations in the Dynamics of Cognition", which collects papers that model cognition as non-linear dynamical processes (as opposed to symbolic reasoning);
- or the textbook of Pfeifer and Scheier 1999, "Understanding Intelligence", which gives a systematic introduction to that approach to cognitive science, AI and robotics that has also been branded under the names like embodied cognitive science, behavior-based robotics / AI, situated agents, or simply new AI, and whose core assumption is that "higher" cognitive functions emerge "bottom-up", on the timescales of evolution and of individual learning and

development, from elementary reflexes like and sensor-motor coordinations like grasping and walking;

- or works from developmental psychology, where the epochal pioneer Jean Piaget (introductory: Piaget 1951, "The biological problem of intelligence") founded the *constructivist* approach to cognition, which explains adult concepts and reasoning as the result of a learning process that starts on day 1 when the newborn child jerks its arm and receives a sensory feedback as the hand strikes the baby's mouth, through many stages of differentiation — all documented with ingeniously attentive observations of minute steps of developmental progress in Piaget's own children — until in early youth the abilities to reason logically and judge ethically become established. The foundations laid by Piaget were subsequently refined in many ways; important proponents are Karmiloff-Smith 1994 who adds the insight that babies do not start their development from an entirely blank slate but have been equipped by evolution with some structuring predispositions; or infant psychologist Thelen 2000, "Grounded in the world: Developmental origins of the embodied mind", who started to collaborate with mathematicians and physicists to obtain rigorous and insightful dynamical systems models of developmental progress;
- or the more foundational-philosophical principle of *affordances* introduced in a lifetime's work by philosopher-psychologist Gibson 1979, "*Perceiving, Acting, and Knowing*", which I would roughly summarize as the principle that our cognitive concepts are grounded in what we can *do* with the conceptualized objects — like, any chair that we see directly "affords" the options to touch it, sit on it, offer it to somebody else etc., and that's the essence of one's "chair" concept;
- or at a grander scale of abstraction, the philosophical school of *evoluationary epistemology* (Bradie 1986) whose central tenet is that our human conceptual and reasoning apparatus has biologically evolved and can be continuously traced back to the *prerational intelligence* of amoeba, see the 1500 page, 1year workshop proceedings "*Prerational Intelligence: Adaptive Behavior and Intelligent Systems Without Symbols and Logic*" (Cruse, Dean, and Ritter 2013) (I was a member of that workshop for three months);
- or the PhD thesis book of Drescher 1991, "Made-up Minds: A Constructivist Approach to Artificial Intelligence", in which the author (who then left academia, what a pity) develops and implements a computer simulation version of Piaget's developmental theory — this work passed under most radars for two decades and is now being re-discovered by Google Deep Mind researchers;
- or, finally (my favorite!), the work of cognitive scientists / cognitive linguists

Lakoff and R. E. Núñez 2000, "Where mathematics comes from: How the embodied mind brings mathematics into being", who analysed the most abstract concepts that our brains have, namely the concepts of mathematics itself (summary article: Lakoff and R. Núñez 2013), and trace their origins back to our abilities and practices of purposeful motor action: walking from here to there becomes the ultimate root of the mathematical "line" concept, putting things on a heap is found to be the source of the "set" concept, etc., all worked out in much greater detail than I can highlight here.

While there are many differences between these lines of thinking and research, they all grow forward from one common idea: that our conceptual and reasoning system has gradually grown from our bodily interaction with our physical environment, both in the evolution of our species and again and further in the development of every individual. As a consequence, if one wants to understand intelligence, one should not *start* "top-down" from symbols, concepts, logic, and adult rational reasoning, but instead reconstruct these precious items "bottomup" as an *end* product that arises from the basic physical-bodily conditions of our existence. Seems a plausible enough story, no? And yet, classical AI and cognitive science wanted to do it the other way round for decades, and still often cling to the "logic first" imperative. But following up on that theme would lead us too far astray here.

Returning to our main topic. The bottom-up approach to intelligence has no difficulties explaining the weaknesses and errors that we humans make all the time when trying to think logically, talk grammatically, or make statistical inferences. The question however remains — can a proper understanding of human cognition help us to revolutionize machine learning? Isn't it so, to the contrary, that trying to build human-like machine learning systems would just give us ML systems that are just as weak as humans when they would try to think, talk or predict? Or, is it so that these weaknesses are the other side of a coin, whose face side would be the the positive powers of human cognition: all that astounding efficiency, robustness, adaptability, versatility and creativity? We don't know and I must leave this question for you to think about. It's a big one.

All I can do is to report on a recent development in machine learning, which tries to flip that coin of learning on its other side. This recent work is based on a 30 year long, specialized research line in Bayesian networks, which in turn turn tried to come to terms with an eternal riddle of philosophy. I will tell this story staring from that philosophical end.

That eternal riddle is the riddle to understand *causality* — intuitively, philosophically, natural-science-ly, statistically.

Intuitively we recognize causality very well what we meet it. When I hit a nail with a hammer, this hammer hit causes the nail to sink in a little further. It's not the other way round: you would call me crazy if I would claim that the nail's little sinking motion caused the hammer to hit it. Yet, both events are tightly — even completely — *correlated*. They coincide. And causation seems always to connect

causes with effects directly, both happening simultaneously. The archetype of cause-effect pairings is when a causing *force* is applied to effect a *change* in the location, speed, or shape of an object. But the change happens exactly at the same time while the force is applied — when there is no force, there is no change. Yet, somehow, the force comes "first" and its effect is "dependent", "secondary". But this is not a temporal "first" versus "second".

A brief digression: when you check out the Wikipedia article on *Granger* causality, you will find a statistical method, which is widely used in financial forecasting, that claims (or tries) to pinpoint causality by exploiting temporal precedence of causes to their effects. Clive Granger was awarded the Nobel prize in economics. Though it is called Granger causality, critics and Granger himself emphasize that this method does not capture the essence of causality, but only a certain phenomenon of systematic temporal precedence. Nonetheless, Granger causality analyses have been applied in a less (self)critical fashion, also in neuroscience where these statistical methods have been used to establish that activity in certain brain regions "causes" response activity in other regions, leading to a directed functional connectivity granger causality").

Returning again to the main thread, I want to confuse you further. There does seem to be something temporal about causation. When you play a video of any everyday life scene backwards, you will find that it looks completely weird. You can very clearly tell whether the movie was played forward or backward: there is an arrow of time which always flies forward. Yet, no physical law is violated in the backward version: all fundamental laws of physics are time-reversible (well, this needs some differentiation: while all models of classical physics are time reversible, in subatomic particle and quantum physics one needs to reverse parity and charge together with time to obtain the time-mirrored equivalent laws). Besides the fundamental laws of physics, also the second law of thermodynamics has been solicited as explaining why real-world time seems to differ in the forward versus the reverse direction. This law states directly that there is an arrow of time. It roughly says, the entropy of a system can only grow, or in more intuitive terms: the world progresses from order to disorder. However, this is a statistical law, not a fundamental one, and can explain the arrow of time only by assuming that the state of the universe at its big bang beginning was highly ordered, such that there is always enough leeway to continue becoming more disordered. The problem of the arrow of time becomes a problem of cosmology. Furthermore, this statistical view cannot explain why you think that it is the hammer that drives the nail, and not the nail that pulls the hammer. The arrow of time problem is not satisfactorily solved by today's physics. Check out https://en.wikipedia.org/wiki/T-symmetry for more information on the physics of time reversal, https://en.wikipedia.org/wiki/Arrow\_of\_time for an introductory (somewhat shallow) discussion of this arrow not only in physics but also in other domains of science and life, and Callender 2011 for in introduction to the philosophy of time.

Cutting things short and maybe a bit unfairly, let me summarize the above as follows: while intuitively we see a close connection between causation and the arrow of time, physicists know that they don't know what exactly this link is, and philosophers know that they do not even know what they don't know. Causation is as ungraspable as time, space, and randomness. It is an eternal riddle.

After having set the stage in this big way, I come to this 30-year long research line in Bayesian networks that I announced above. This part of the story is a tribute to Judea Pearl. You know him as the pioneer of Bayesian networks. As you have learnt in this course, Bayesian networks (BNs) are our best (in many ways, only) tool when it comes to reason about joint probability distributions of many interacting random variables. Specifically, BNs allow us to calculate conditional probabilities P(Y|X) of causes given symptoms, where symptoms X are observed and causes Y are inferred. Wait... "causes"?? This is the word that is customarily used in the BN literature. Furthermore, BNs are also known as "causal" networks, and the arrows in a BN are said to reflect "causal" influences: a modeler drawing the graph structure of a BN should draw an arrow  $Y \to X$ when he/she thinks there is a "causal" influence from Y to X. This is all right because and inasmuch a human modeler has a clear intuitive grasp on causation. But above I painted a picture that shows that our intuitive understanding does not amount to scientific or philosophical understanding. Thus, it seems that calling BNs and the arrows within "*causal*" is neither good science, nor good philosophy — just a selling point and a helper to let modelers think of heuristically okay'ish graph structures.

The first person to become frustrated by this conceptual blank in BNs was Pearl himself. Well, I guess so; I don't know him in person. But it is a fact that after establishing the foundations of BNs as you saw them in our course, ever after 1991 he directed his efforts to detecting true causality in statistical models (early publications: Pearl and Verma 1992; Pearl 1993). Apparently his preoccupation with causality started much earlier. Citing Pearl (after Russell 2011):

"I got my first hint of the dark world of causality during my junior year of high school. My science teacher, Dr. Feuchtwanger, introduced us to the study of logic by discussing the 19th century finding that more people died from smallpox inoculations than from smallpox itself. Some people used this information to argue that inoculation was harmful when, in fact, the data proved the opposite, that inoculation was saving lives by eradicating smallpox.

And here is where logic comes in," concluded Dr. Feuchtwanger, "To protect us from cause-effect fallacies of this sort." We were all enchanted by the marvels of logic, even though Dr. Feuchtwanger never actually showed us how logic protects us from such fallacies.

It doesn't, I realized years later as an artificial intelligence researcher. Neither logic, nor any branch of mathematics had developed adequate tools for managing problems, such as the smallpox inoculations, involving cause-effect relationships." Baysian networks, as you know them, do not allow one to pin down causality. Citing Russell 2011: "A Bayesian network such as Smoking  $\rightarrow$  Cancer fails to capture causal information; indeed, it is mathematically equivalent to the network Cancer  $\rightarrow$  Smoking." No model of a joint distribution has information about true causal dependencies between the modeled random variables. All that can be inferred from any such model are statistical dependencies of various sorts (correlations, conditional probabilities), but as you surely have heard, Correlation Does Not Imply Causation, and neither does the stronger brother of correlation, statistical dependence. In this situation, Pearl extended the mathematical apparatus of statistics and added a new operator to statistics, today called the do-operator. Here I can only give you an intuition of this operator; the mathematical theory behind it is not super easy and describing it would turn your reading experience into real work (if you want to dig deeper: the tutorial by Schölkopf, Locatello, et al. 2021 gives a brilliantly written introduction). So, — what does the do-operator do?

One way to understand it is to take a look at clinical studies that aim at establishing the true causal efficacy of a new drug. To achieve this goal, a *doubleblind* study must be carried out. The new drug, or a placebo, are administered to two balanced populations of patients. If the drug-administered population shows a statistically significant improved health status compared to the placebo group, a (positive) causal effect of the drug is established. While this scheme is sound, it suffers from statistical variance, and from imperfections in balancing the treatment versus placebo groups. The statistical efficiency of this scheme would be very much enhanced if one could give the drug to the very same pool of patients as the placebo. For this, however, one would need two worlds, which at the beginning of the study would be identical, and where in world A one would give the patient pool the drug, and in world B the placebo, and then compare the outcomes. But we have only one world.

Here comes the do-operator. It creates these two worlds. Staying in the medical domain, consider the classical probability P(X = x | Y = y), which describes the probability that a health status x is observed after administering a treatment y. Estimating the conditional distribution  $P_{X|Y}$  from data — the standard basic task of statistics and machine learning — is based on datasets  $(x_i, y_i)_{i=1,...,N}$ , where the cases i correspond to individual patients. Unfortunately, each patient i is either given the drug, or the placebo. This bars the way to retrieving causality from these data, because causal influences are confounded with the assignments of patients i to the treatment versus placebo populations. We would however get causality information if we could observe each patient i in two conditions, one observation giving the health outcome  $x_i$  when the drug  $y_i$  was given, and the other observation  $x'_i$  when the placebo  $y'_i$  was given. The do-operator gives us these two sorts of probabilities. The formal expression P(X = x | do(Y = y)) represents the probability of getting health outcome x if all patients received treatment y, and

P(X = x | do(Y = y')) is the probability of getting outcome x under administering the placebo y' to all patients. In the parlance of causal statistics, the two "dosettings" do(Y = y) and do(Y = y') are called *interventions*. If we would know these two kinds of interventional probabilities, we could assert a causal effect of the treatment (if there is any).

Obviously, these interventional statistics are not contained in the data. Each patient *i* is observed only under one of the two conditions: the dataset contains either  $(x_i, y_i)$ , or  $(x'_i, y'_i)$ , but not both. To get "data" for estimating  $P(X = x | \operatorname{do}(Y = y))$  and  $P(X = x | \operatorname{do}(Y = y'))$ , for each patient *i* we would need counterfactual information that complements the actually observed half of the picture, for instance: "if patient *i would have* gotten placebo *y'*, we would have observed outcome *x'*. The new statistical theory, which was worked out and stepwise refined by Pearl and others in his followship, has identified a number of additional assumptions — often satisfiable in reasonably good approximation — and computational techniques, which together allow us to derive the interventional distributions  $P_{X|\operatorname{do}(Y)}$  from a classical Bayesian network which was estimated from "one-world-only" data  $(x_i, y_i)_{i=1,\ldots,N}$ . These techniques involve the construction of modified versions of the original BN, in which certain links are deleted, and involved estimations of auxiliary conditional probabilities.

You might ask, why not simply collect data like it is done in double-blind clinical studies, and then have direct access to causality through these data? This is not an option in general, for several reasons. First, in clinical studies the independent variable (drug versus placebo) is binary, which mandates two balanced patient populations. If one wants to study the causal impact of an independent variable with n different values, one would need n separate patient populations. And if one wants to investigate the causal relationships between k variables, each with n values, one would need  $n^k$  separate data-donating populations — a dead end. Furthermore, BNs are typically estimated from data that do not come from controlled experiments, but from opportunistically collected ad-hoc observations, motto: "use any data you can get" (as was the case for the Space Shuttle example in Section 10, and is the case for all the magnificient DL super-machines). The do-operator methods of causal analysis enable the assessment of the direction and strength of causal influence between all modeled random variables in all of these scenarios! (BIG exclamation mark!)

So far, causal analysis techniques have become widely adopted mostly in medicine and in particular, in epidemology (as far as I can tell, I am not an expert and this is second-hand but reliable information). A recent Master's project that I was lucky enough to accompany from a distance (it was carried out at the Harvard Medical School by RUG student Richard Fischer) provides a beautiful case study and analysis of causal analysis, and if you are interested in these matters, is a recommendable reading with transparent methods explanations (Fischer 2023). This thesis also illustrates the remaining problems of these techniques, among them numerical instabilities and the need for numerous clever algorithm design decisions from the side of the data analyst. It's not (yet) a download-and-be-happy tool for non-expert users.

Before we proceed to the hopes for revolutionizing ML and DL at large, let us step back and consider the big question of causality once again. Causal analysis in the spirit of Pearl does not give a philosophical answer to the eternal question of the essence of causality. It is a computational technique that lets us identify a specific kind of causality in data. What is the nature of this specific kind of causality? The key lies in the little word "do". The do-operator formalizes the *decision* to assign a drug or placebo to a patient, or in more general terms, to set the value of an independent variable by a wilful act of an experimenter. This is analog to how experiments in the natural sciences should be carried out: the values of independent variables are systematically varied while otherwise the experimental set-up is not changed. In our statistical terminology, these *controlled laboratory conditions* amount to repeated observations of the same "individual" *i* (the unchanged experiment set-up) under application of different "treatments". The workings of the do-operator are also analog to how human children and adults learn about causal effects of their actions: in our daily experimentation with the world around us we infer what our acts causally effect by comparing the outcomes when we do something versus when we don't do it. We observe that the nail does not sink in while we do not use the hammer, and that it sinks in when we do. The philosophical side of this: we learn about a kind of causality that is connected to our wilful actions, "wilful" meaning that we decide when and what to do or not to do. This view on "causality" is philosophically worked out in Kutach 2011 (warning: not an easy read, it's a philosopher's text!).

And now, finally, we return to ML, and DL in particular. We started our thought odyssee by asking the double question, what is it in purely statistical distribution modeling that makes it so weak with regards to efficiency, generalization, robustness etc., and what is it in human cognition and learning that makes us so strong with regards to the same (at the price of being weak statisticians)? One possible answer is that our human conceptual system is not organized around probabilities, correlations and statistical dependencies, but around chains of causality, and furthermore, that the sort of causality that we use roots in our experience of the consequences of wilful action.

This leads to a revolutionary research programme for putting ML on a new theoretical foundation. Instead of looking at data in search of statistical structure, look at them in search for causal structures. The hope and promise of this view is that, if a ML model is structured by representational modules that interact along *causal* influence chains, not through *statistical dependency* connections, information from new data not contained in the original training corpus can be incorporated much faster and more robustly. For instance, if a child has learnt that when she hits a small solid object with a heavier one, the impact will cause the smaller object to be driven away from the impact direction, and this child then gets a hammer into her hands for the first time in her life, she will predict that the nail will go down. A purely statistical learner, in turn, will be quite helpless because hammers were not present in the training data, and a good amount of new hammering data would be needed to establish a sufficiently significant correlation between the hammer's and the nail's behavior.

This is an intriguing perspective. Pearl's original do-operator formalism is tied to the mathematical BN framework, and furthermore, causal analysis starts from a trained BN and the do-operator does not instruct us how we can improve training the BN in the first place (as far as I can judge, I must repeat that I am no BN expert). Thus, Pearl's ingenious studies do not directly show us how we could put ML at large on a new foundation. Enter Bernhard Schölkopf. Renown for his contributions on statistical learning theory, kernel methods in general and support vector machines in particular, since more than a decade Schölkopf has taken up the challenge of making causality principles fertile for ML in general. He has teamed up with deep neural network researchers. Two recent, extensive, visionary tutorials written by him together with other ML experts (among them, DL pioneer Joshua Bengio) will, as far as I can forecast, become a milestone in spreading the idea (Schölkopf, Locatello, et al. 2021; Schölkopf and Kügelgen 2022). Deep neural network models that are organized around causality are beginning to sprout (for example, Tangemann et al. 2021). Do not expect an immediate total overthrow of machine learning, however. Causal analysis is subtle and difficult enough even in the restricted BN setting, and transfer to other branches of ML is far from straightforward. I conclude by citing the concluding remarks of Schölkopf and Kügelgen 2022: "Most of the discussed fields are still in their infancy [...]. With the current hype around machine learning, there is much to say in favor of some humility towards what machine learning can do, and thus towards the current state of AI — the hard problems have not been solved yet, making basic research in this field all the more exciting."

## Appendix

## A Elementary mathematical structure-forming operations

#### A.1 Pairs, tuples and indexed families

If two mathematical objects  $\mathcal{O}_1, \mathcal{O}_2$  are given, they can be grouped together in a single new mathematical structure called the *ordered pair* (or just *pair*) of  $\mathcal{O}_1, \mathcal{O}_2$ . It is written as

$$(\mathcal{O}_1, \mathcal{O}_2).$$

In many cases,  $\mathcal{O}_1, \mathcal{O}_2$  will be of the same kind, for instance both are integers. But the two objects need not be of the same kind. For instance, it is perfectly possible to group integer  $\mathcal{O}_1 = 3$  together with a random variable (a function!)  $\mathcal{O}_2 = X_7$ in a pair, getting  $(3, X_7)$ .

The crucial property of a pair  $(\mathcal{O}_1, \mathcal{O}_2)$  which distinguishes it from the *set*  $\{\mathcal{O}_1, \mathcal{O}_2\}$  is that the two members of a pair are *ordered*, that is, it makes sense to speak of the "first" and the "second" member of a pair. In contrast, it makes not sense to speak of the "first" or "second" element of the set  $\{\mathcal{O}_1, \mathcal{O}_2\}$ . Related to this is the fact that the two members of a pair can be the same, for instance (2, 2) is a valid pair. In contrast,  $\{2, 2\}$  makes no sense.

A generalization of pairs is *N*-tuples. For an integer N > 0, an *N*-tuple of *N* objects  $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_N$  is written as

 $(\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_N).$ 

1-tuples are just individual objects; 2-tuples are pairs, and for N > 2, N-tuples are also called *lists* (by computer scientists that is; mathematicians rather don't use that term). Again, the crucial property of N-tuples is that one can identify its *i*-th member by its position in the tuple, or in more technical terminology, by its *index*. That is, in an N-tuple, every index  $1 \le i \le N$  "picks" one member from the tuple.

The infinite generalization of N-tuples is provided by *indexed families*. For any nonempty set I, called an *index set* in this context,

 $(\mathcal{O}_i)_{i\in I}$ 

denotes a compound object assembled from as many mathematical objects as there are index elements  $i \in I$ , and within this compound object, every individual member  $\mathcal{O}_i$  can be "addressed" by its index *i*. One simply writes

 $\mathcal{O}_i$ 

to denote the *i*th "component" of  $(\mathcal{O}_i)_{i \in I}$ . Writing  $\mathcal{O}_i$  is a shorthand for applying the *i*th projection function on  $(\mathcal{O}_i)_{i \in I}$ , that is,  $\mathcal{O}_i = \pi_i((\mathcal{O}_i)_{i \in I})$ .

#### A.2 Products of sets

We first treat the case of products of a finite number of sets. Let  $S_1, \ldots, S_N$  be (any) sets. Then the product  $S_1 \times \ldots \times S_N$  is the set of all N-tuples of elements from the corresponding sets, that is,

$$S_1 \times \ldots \times S_N = \{(s_1, \ldots, s_N) \mid s_i \in S_i\}$$

This generalizes to infinite products as follows. Let I be any set — we call it an *index* set in this context. For every  $i \in I$ , let  $S_i$  be some set. Then the *product* set indexed by I is the set of functions

$$\prod_{i \in I} S_i = \{ \varphi : I \to \bigcup_{i \in I} S_i \mid \forall i \in I : \varphi(i) \in S_i \}.$$

Using the notation of indexed families, this could equivalently be written as

$$\prod_{i \in I} S_i = \{ (s_i)_{i \in I} \mid \forall i \in I : s_i \in S_i \}.$$

If all the sets  $S_i$  are the same, say S, then the product  $\prod_{i \in I} S_i = \prod_{i \in I} S$  is also written as  $S^I$ .

An important special case of infinite products is obtained when  $I = \mathbb{N}$ . This situation occurs universally in modeling stochastic processes with discrete time. The elements  $n \in \mathbb{N}$  are the points in time when the amplitude of some signal is measured. The amplitude is a real number, so at any time  $n \in \mathbb{N}$ , one records an amplitude value  $a_n \in S_n = \mathbb{R}$ . The product set

$$\prod_{n \in \mathbb{N}} S_n = \{ \varphi : \mathbb{N} \to \bigcup_{n \in \mathbb{N}} S_n \mid \forall n \in I : \varphi(n) \in S_n \} = \{ \varphi : \mathbb{N} \to \mathbb{R} \}$$

is the set of all right-infinite real-valued timeseries (with discrete time points starting at time n = 0).

#### A.3 Products of functions

First, again, the case of finite products: let  $f_1, \ldots, f_N$  be functions, all sharing the same domain D, with image sets  $S_i$ . Then the product  $f_1 \otimes \ldots \otimes f_N$  of these functions is the function with domain D and image set  $S_1 \times \ldots \times S_N$  given by

$$f_1 \otimes \ldots \otimes f_N : D \to S_1 \times \ldots \times S_N$$
$$d \mapsto (f_1(d), \ldots, f_N(d)).$$

Again this generalizes to arbitrary products. Let  $(f_i : D \to S_i)_{i \in I}$  be an indexed family of functions, all of them sharing the same domain D, and where the image set of  $f_i$  is  $S_i$ . The product  $\bigotimes_{i \in I} f_i$  of this set of functions is defined by

$$\bigotimes_{i \in I} f_i : D \to \prod_{i \in I} S_i$$
$$d \mapsto \varphi : I \to \bigcup_{i \in I} S_i \quad \text{given by } \varphi(i) = f_i(d).$$

### **B** Joint, conditional and marginal probabilities

*Note.* This little section is only a quick memory refresher of some of the most basic concepts of probability. It does not replace a textbook chapter!

We first consider the case of two observations of some part of reality that have discrete values. For instance, an online shop creating customer profiles may record from their customers their age and gender (among many other items). The marketing optimizers of that shop are not interested in the exact age but only in age brackets, say  $a_1 = \text{at most 10 years old}$ ,  $a_2 = 11 - 20$  years,  $a_3 = 21 - 30$ years,  $a_4 = \text{older than 30}$ . Gender is roughly categorized into the possibilities  $g_1 = f, g_2 = m, g_3 = o$ . From their customer data the marketing guys estimate the following probability table:

$P(X = g_i, Y = a_j)$	$a_1$	$a_2$	$a_3$	$a_4$	
$g_1$	0.005	0.3	0.2	0.04	(179
$g_2$	0.005	0.15	0.15	0.04	
$g_3$	0.0	0.05	0.05	0.01	

The cell (i, j) in this  $3 \times 4$  table contains the probability that a customer with gender  $g_i$  falls into the age bracket  $a_j$ . This is the *joint probability* of the two observation values  $g_i$  and  $a_j$ . Notice that all the numbers in the table sum to 1.

The mathematical tool to formally describe a category of an observable value is a random variable (RV). We typically use symbols  $X, Y, Z, \ldots$  for RVs in abstract mathematical formulas. When we deal with concrete applications, we may also use "telling names" for RVs. For instance, in Table (172), instead of  $P(X = g_i, Y = a_j)$  we could have written  $P(\text{Gender} = g_i, \text{Age} = a_j)$ . Here we have two such observation categories: gender and age bracket, and hence we use two RVs X and Y for gender and age, respectively. In order to specify, for example, that female customers in the age bracket 11-20 occur with a probability of 0.3 in the shop's customer reservoir (the second entry in the top line of the table), we write  $P(X = g_1, Y = a_2) = 0.3$ .

Some more info bits of concepts and terminology connected with RVs. You should consider a RV as the mathematical counterpart of a procedure or apparatus to make observations or measurements. For instance, the real-world counterpart of the **Gender** RV could be an electronic questionnaire posted by the online shop, or more precisely, the "what is your age?" box on that questionnaire, plus the whole internet infrastructure needed to send the information entered by the customer back to the company's webserver. Or in a very different example (measuring the speed of a car and showing it to the driver on the speedometer) the real-world counterpart of a RV **Speed** would be the total on-board circuitry in a car, comprising the wheel rotation sensor, the processing DSP microchip, and the display at the dashboard.

A RV always comes with a set of possible outcomes. This set is called the sample space of the RV, and I usually denote it with the symbol S. Mathematically,

a sample space is a set. The sample space for the Gender RV would be the set  $S = \{\mathsf{m}, \mathsf{f}, \mathsf{o}\}$ . The sample space for Age that we used in the table above was  $S = \{\{0, 1, \ldots, 10\}, \{11, \ldots, 20\}, \{21, \ldots, 30\}, \{31, 32, \ldots\}\}$ . For car speed measuring we might opt for  $S = \mathbb{R}^{\geq 0}$ , the set of non-negative reals. A sample space can be larger than the set of measurement values that are realistically possible, but it must contain *at least* all the possible values.

Back to our table and the information it contains. If we are interested only in the age distribution of customers, ignoring the gender aspects, we sum the entries in each age column and get the *marginal probabilities* of the RV Y. Formally, we compute

$$P(Y = a_j) = \sum_{i=1,2,3} P(X = g_i, Y = a_j).$$

Similarly, we get the marginal distribution of the gender variable by summing along the rows. The two resulting marginal distributions are indicated in the table (173).

	$a_1$	$a_2$	$a_3$	$a_4$	
$g_1$	0.005	0.3	0.2	0.04	0.545
$g_2$	0.005	0.15	0.15	0.04	0.345
$g_3$	0.0	0.05	0.05	0.01	0.110
	0.01	0.5	0.4	0.09	

Notice that the marginal probabilities of age 0.01, 0.5, 0.4, 0.09 sum to 1, as do the gender marginal probabilities.

Finally, the conditional probability  $P(X = g_i | Y = a_j)$  that a customer has gender  $g_i$  given that the age bracket is  $a_j$  is computed through dividing the joint probabilities in column j by the sum of all values in this column:

$$P(X = g_i | Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(Y = a_j)}.$$
(174)

There are two equivalent versions of this formula:

$$P(X = g_i, Y = a_j) = P(X = g_i | Y = a_j)P(Y = a_j)$$
(175)

where the righthand side is called a *factorization* of the joint distribution on the lefthand side, and

$$P(Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(X = g_i | Y = a_j)},$$
(176)

demonstrating that each of the three quantities (joint, conditional, marginal probability) can be expressed by the respective two others. If you memorize one of these formulas – I recommend the second one – you have memorized the very key to master "probability arithmetics" and will never get lost when manipulating probability formulas.

The factorization (175) can be done in two ways:  $P(Y = a_j | X = g_i)P(X = g_i) = P(X = g_i | Y = a_j)P(Y = a_j)$ , which gives rise to *Bayes' formula* 

$$P(Y = a_j | X = g_i) = \frac{P(X = g_i | Y = a_j)P(Y = a_j)}{P(X = g_i)},$$
(177)

which has many uses in statistical modeling because it shows how one can revert the conditioning direction.

Joint, conditional, and marginal probabilities are also defined when there are more than two categories of observations. For instance, the online shop marketing people also record how much a customer spends on average, and formalize this by a third random variable, say Z. The values that Z can take are spending brackets, say  $s_1 = \text{less than 5 Euros}$  to  $s_{20} = \text{more than 5000 Euros}$ . The joint probability values  $P(X = g_i, Y = a_j, Z = s_k)$  would be arranged in a 3-dimensional array sized  $3 \times 4 \times 20$ , and again all values in this array together sum to 1. Now there are different arrangements for conditional and marginal probabilities, for instance  $P(Z = s_k | X = g_i, Y = a_j)$  is the probability that among the group of customers with gender  $g_i$  and age  $a_j$ , a person spends an amount in the range  $s_k$ . Or  $P(Z = s_k, Y = a_j | X = g_i)$  is the probability that in the gender group  $g_i$  a person is aged  $a_j$  and spends  $s_k$ . As a last example, the probabilities  $P(X = g_i, Z = s_j)$  are the marginal probabilities obtained by summing away the Y variable:

$$P(X = g_i, Z = s_j) = \sum_{k=1,2,3,4} P(X = g_i, Y = a_k, Z = s_j)$$
(178)

So far I have described cases where all kinds of observations were *discrete*, that is, they (i.e. all RVs) yield values from a finite set – for instance the three gender values or the four age brackets. Equally often one faces *continuous* random values which arise from observations that yield real numbers – for instance, measuring the body height or the weight of a person. Since each such RV can give infinitely many different observation outcomes, their probabilities cannot be represented in a table or array. Instead, one uses *probability density functions* (pdf's) to write down and compute probability values.

Let's start with a single RV, say H = Body Height. Since body heights are non-negative and, say, never larger than 3 m, the distribution of body heights within some reference population can be represented by a pdf  $f : [0,3] \to \mathbb{R}^{\geq 0}$ which maps the interval [0,3] of possible values to the nonnegative reals (Figure 88). We will be using subscripts to make it clear which RV a pdf refers to, so the pdf describing the distribution of body height will be written  $f_H$ .

A pdf for the distribution of a continuous RV X can be used to calculate the probability that this RV takes values within a particular interval, by integrating the pdf over that interval. For instance, the probability that a measurement of



Figure 88: A hypothetical distribution of human body sizes in some reference population, represented by a pdf.

body height comes out between 1.5 and 2.0 meters is obtained by

$$P(H \in [1.5, 2.0]) = \int_{1.5}^{2.0} f_H(x) dx, \qquad (179)$$

see the shaded area in Figure 88. Some comments:

- A probability density function is actually defined to be a function which allows one to compute probabilities of value intervals as in Equation 179. For a given continuous RV X over the reals there is exactly one function  $f_X$  which has this property, the pdf for X. (This is not quite true. There exist also continuous-valued RVs whose distribution is so complex that it cannot be captured by a pdf, but we will not meet with such phenomena in this lecture. Furthermore, a given pdf can be altered on isolated points which come from what is called a *null set* in probability theory and still be a pdf for the same distribution. But again, we will not be concerned with such subtelties in this lecture.)
- As a consequence, any pdf  $f : \mathbb{R} \to \mathbb{R}^{\geq 0}$  has the property that it integrates to 1, that is,  $\int_{-\infty}^{\infty} f(x) dx = 1$ .
- Be aware that the values f(x) of a pdf are not probabilities! Pdf's turn into probabilities only through integration over intervals.
- Values f(x) can be greater than 1 (as in Figure 88), again indicating that they cannot be taken as probabilities.

Joint distributions of two continuous RVs X, Y can be captured by a pdf  $f_{X,Y}$ :  $\mathbb{R}^2 \to \mathbb{R}^{\geq 0}$ . Figure 89 shows an example. Again, the pdf  $f_{X,Y}$  of a bivariate continuous distribution must integrate to 1 and be non-negative; and conversely, every such function is the pdf of a continuous distribution of two RV's.



Figure 89: An exemplary joint distribution of two continuous-valued RVs X, Y, represented by its pdf.

Continuing on this track, the joint distribution of k continuous-valued RVs  $X_1, \ldots, X_k$ , where the possible values of each  $X_i$  are bounded to lie between  $a_i$  and  $b_i$  can be described by a unique pdf function  $f_{X_1,\ldots,X_k} : \mathbb{R}^k \to \mathbb{R}^{\geq 0}$  which integrates to 1, i.e.

$$\int_{a_1}^{b_1} \dots \int_{a_k}^{b_k} f(x_1, \dots, x_k) \ dx_k \dots dx_1,$$

where also the cases  $a_i = -\infty$  and  $b_i = \infty$  are possible. A more compact notation for the same integral is

$$\int_D f(\mathbf{x}) \, d\mathbf{x},$$

where D denotes the k-dimensional box  $[a_1, b_1] \times \ldots \times [a_k, b_k]$  and **x** denotes vectors in  $\mathbb{R}^k$ . Mathematicians speak of k-dimensional *intervals* instead of "boxes". The set of points  $S = \{ \mathbf{x} \in \mathbb{R}^k \mid f_{X_1,\ldots,X_k} > 0 \}$  is called the *support* of the distribution. Obviously  $S \subseteq D$ .

In analogy to the 1-dim case from Figure 88, probabilities are obtained from a k-dimensional pdf  $f_{X_1,\ldots,X_k}$  by integrating over sub-intervals. For such a kdimensional subinterval  $[r_1, s_1] \times \ldots \times [r_k, s_k] \subseteq [a_1, b_1] \times \ldots \times [a_k, b_k]$ , we get its probability by

$$P(X_1 \in [r_1, s_1], \dots, X_k \in [r_k, s_k]) = \int_{r_1}^{s_1} \dots \int_{r_k}^{s_k} f(x_1, \dots, x_k) \, dx_k \dots dx_1.$$
(180)

In essentially the same way as we did for discrete distributions, the pdf's of marginal distributions are obtained by *integrating away* the RV's that one wishes

to expel. In analogy to (178), for instance, one would get

$$f_{X_1,X_3}(x_1,x_3) = \int_{a_2}^{b_2} f_{X_1,X_2,X_3}(x_1,x_2,x_3) \, dx_2. \tag{181}$$

And finally, pdf's of conditional distributions are obtained through dividing joint pdfs by marginal pdfs. Such conditional pdfs are used to calculate that some RVs fall into a certain multidimensional interval given that some other RVs take specific values. We only inspect a simple case analog to (174) where we want to calculate the probability that X falls into a range [a, b] given that Y is known to be c, that is, we want to evaluate the probability  $P(X \in [a, b] | Y = c)$ , using pdfs. We can obtain this probability from the joint pdf  $f_{X,Y}$  and the marginal pdf  $f_Y$ by

$$P(X \in [a,b] | Y = c) = \frac{\int_a^b f_{X,Y}(x,c) \, dx}{f_Y(c)}.$$
(182)

The r.h.s. expression  $\int_a^b f_{X,Y}(x,c) dx / f_Y(c)$  is a function of x, parametrized by c. This function is a pdf, denoted by  $f_{X|Y=c}$ , and defined by

$$f_{X|Y=c}(x) = \frac{f_{X,Y}(x,c)}{f_Y(c)}.$$
(183)

Let me illustrate this with a concrete example. An electronics engineer is testing a device which transforms voltages V into currents I. In order to empirically measure the behavior of this device (an electronics engineer would say, in order to "characterize" the device), the engineer carries out a sequence of measurement trials where he first sets the input voltage V to a specific value, say V = 0.0. Then he (or she) measures the resulting current many times, in order to get an idea of the stochastic spread of the current. In mathematical terms, the engineer wants to get an idea of the pdf  $f_{I|V=0.0}$ . The engineer then carries on, setting the voltage to other values  $c_1, c_2, ...$ , measuring resulting currents in each case, and getting ideas of the conditional pdfs  $f_{I|V=c_i}$ . For understanding the characteristics of this device, the engineer needs to know all of these pdfs.

Conditional distributions arise whenever cause-effect relationships are being modeled. The conditioning variables are causes, the conditioned variables describe effects. In experimental and empirical research, the causes are under the control of an experimenter and can (and have to) be set to specific values in order to assess the statistics of the effects – which are not under the control of the experimenter. In ML pattern classification scenarios, the "causes" are the input patterns and the "effects" are the (stochastically distributed) class label assignments. Since research in the natural sciences is very much focussed on determining Nature's cause-effect workings, and 90% of the applications in machine learning concern pattern classification (my estimate), it is obvious that conditional distributions lie at the very heart of scientific (and engineering) modeling and data analysis. In this appendix (and in the lecture) I consider only two ways of representing probability distributions: discrete ones by finite probability tables or probability tables; continuous ones by pdfs. These are the most elementary formats of representing probability distributions. There are many others which ML experts readily command on. This large and varied universe of concrete *representations* of probability distributions is tied together by an abstract mathematical theory of the probability distributions themselves, independent of particular representations. This theory is called *probability theory*. It is not an easy theory and we don't attempt an introduction to it. If you are mathematically minded, then you can get an introduction to probability theory in my graduate lecture notes "Principles of Statistical Modeling" (https://www.ai.rug.nl/minds/uploads/LN\_PSM.pdf). At this point I only highlight two core facts from probability theory:

- A main object of study in probability theory are distributions. They are abstractly and axiomatically defined and analyzed, without reference to particular representations (such as tables or pdfs).
- A probability distribution *always* comes together with random variables. We write  $P_X$  for the distribution of a RV X,  $P_{X,Y}$  for the joint distribution of two RVs X, Y, and  $P_{X|Y}$  for the conditional distribution (a truly involved concept since it is actually a family of distributions) of X given Y.

### C The argmax operator

Let  $\varphi: D \to \mathbb{R}$  be some function from some domain D to the reals. Then

 $\mathop{\rm argmax}_{a} \varphi(a)$ 

is that  $d \in D$  for which  $\varphi(d)$  is maximal among all values of  $\varphi$  on D. If there are several arguments a for which  $\varphi$  gives the same maximal value, – that is,  $\varphi$  does not have a unique maximum –, or if  $\varphi$  has no maximum at all, then the argmax is undefined.

## D Expectation, variance, covariance, and correlation of numerical random variables

Recall that a random variable is the mathematical model of an observation / measurement / recording procedure by which one can "sample" observations from that piece of reality that one wishes to model. We usually denote RVs by capital roman letters like X, Y or the like. For example, a data engineer of an internet shop who wants to get a statistical model of its (potential) customers might record the gender and age and spending of shop visitors – this would be formally captured

by three random variables G, A, S. A random variable always comes together with a sample space. This is the set of values that might be delivered by the random variable. For instance, the sample space of the gender RV G could be cast as  $\{m, f, o\}$  – a symbolic (and finite) set. A reasonable sample space for the age random variable A would be the set of integers between 0 and 200 – assuming that no customer will be older than 200 years and that age is measured in integers (years). Finally, a reasonable sample space for the spending RV S could be just the real numbers  $\mathbb{R}$ .

Note that in the A and S examples, the sample spaces that I proposed look very generous. We would not really expect that some customer is 200 years old, nor would we think that ever a customer spends  $10^{1000}$  Euros – although both values are included in the respective sample space. The important thing about a sample space is that it must contain all the values that might be returned by the RV; but it may also contain values that will never be observed in practice.

Every mathematical set can serve as a sample space. We just saw symbolic, integer, and real sample spaces. Real sample spaces are used whenever one is dealing with an observation procedure that returns numerical values. Real-valued RVs are of great practical importance, and they allow many insightful statistical analyses that are not defined for non-numerical RVs. The most important analytical characteristics of real RVs are expectation, variance, and covariance, which I will now present in turn.

For the remainder of this appendix section we will be considering random variables X whose sample space is  $\mathbb{R}^n$  — that is, observation procedures which return scalars (case n = 1) or vectors. We will furthermore assume that the distributions of all RVs X under consideration will be represented by pdf's  $f_X$ :  $\mathbb{R}^n \to \mathbb{R}^{\geq 0}$ . (In mathematical probability theory, more general numerical sample spaces are considered, as well as distributions that have no pdf — but we will focus on this basic scenario of real-valued RVs with pdfs).

The *expectation* of a RV X with sample space  $\mathbb{R}^n$  and pdf  $f_X$  is defined as

$$E[X] = \int_{\mathbb{R}^n} x f_X(x) \, dx, \qquad (184)$$

where the integral is written in a common shorthand for

$$\int_{x_1=-\infty}^{\infty} \dots \int_{x_n=-\infty}^{\infty} (x_1, \dots, x_n)' f_X((x_1, \dots, x_n)) dx_n \dots dx_1.$$

The expectation of a RV X can be intuitively understood as the "average" value that is delivered when the observation procedure X would be carried out infinitely often. The crucial thing to understand about the expectation is that it does not depend on a sample, – it does not depend on specific data.

In contrast, whenever in machine learning we base some learning algorithm on a (numerical) training sample  $(x_i, y_i)_{i=1,\dots,N}$  drawn from the joint distribution  $P_{X,Y}$  of two RVs X, Y, we may compute the average value of the  $x_i$  by

mean
$$(\{x_1, \dots, x_N\}) = 1/N \sum_{i=1}^N x_i,$$

but this sample mean is NOT the expectation of X. If we would have used another random sample, we would most likely have obtained another sample mean. In contrast, the expectation E[X] of X is defined not on the basis of a finite, random sample of X, but it is defined by averaging over the true underlying distribution.

Since in practice we will not have access to the true pdf  $f_X$ , the expectation of a RV X cannot usually be determined in full precision. The best one can do is to *estimate* it from observed sample data. The sample mean is an *estimator* for the expectation of a numerical RV X. Marking estimated quantities by a "hat" accent, we may write

$$\hat{E}[X] = 1/N \sum_{i=1}^{N} x_i.$$

A random variable X is *centered* if its expectation is zero. By subtracting the expectation one gets a centered RV. In these lecture notes I use the bar notation to mark centered RVs:

$$\bar{X} := X - E[X].$$

The *variance* of a scalar RV with sample space  $\mathbb{R}$  is the expected squared deviation from the expectation

$$\sigma^2(X) = E[\bar{X}^2],\tag{185}$$

which in terms of the pdf  $f_{\bar{X}}$  of  $\bar{X}$  can be written as

$$\sigma^2(X) = \int_{\mathbb{R}} x^2 f_{\bar{X}}(x) \, dx.$$

Like the expectation, the variance is an intrinsic property of an observation procedure X and the part of the real world where the measurements may be taken from — it is independent of a concrete sample. A natural way to estimate the variance of X from a sample  $(x_i)_{i=1,\dots,N}$  is

$$\hat{\sigma}^2(\{x_1,\ldots,x_N\}) = 1/N \sum_{i=1}^N \left(x_i - 1/N \sum_{j=1}^N x_j\right)^2,$$

but in fact this estimator is not the best possible – on average (across different samples) it underestimates the true variance. If one wishes to have an estimator that is *unbiased*, that is, which on average across different samples gives the correct variance, one must use

$$\hat{\sigma}^2(\{x_1,\ldots,x_N\}) = 1/(N-1) \sum_{i=1}^N \left(x_i - 1/N \sum_{j=1}^N x_j\right)^2$$

instead. The Wikipedia article on "Variance", section "Population variance and sample variance" points out a number of other pitfalls and corrections that one should consider when one estimates variance from samples.

The square root of the variance of X,  $\sigma(X) = \sqrt{\sigma^2(X)}$ , is called the *standard* deviation of X.

The *covariance* between two real-valued scalar random variables X, Y is defined as

$$\operatorname{Cov}(X,Y) = E[\bar{X}\,\bar{Y}],\tag{186}$$

which in terms of a pdf  $f_{\bar{X}\bar{Y}}$  for the joint distribution for the centered RVs spells out to

$$\operatorname{Cov}(X,Y) = \int_{\mathbb{R}\times\mathbb{R}} x \, y \, f_{\bar{X}\,\bar{Y}}((x,y)') \, dx \, dy.$$

An unbiased estimate of the covariance, based on a sample  $(x_i, y_i)_{i=1,\dots,N}$  is given by

$$\widehat{\operatorname{Cov}}((x_i, y_i)_{i=1,\dots,N}) = 1/(N-1) \left( x_i - 1/N \sum_i x_i \right) \left( y_i - 1/N \sum_i y_i \right)$$

Finally, let us inspect the *correlation* of two scalar RVs X, Y. Here we have to be careful because this term is used differently in different fields. In statistics, the correlation is defined as

$$\operatorname{Corr}(X,Y) = \frac{\operatorname{Cov}(X,Y)}{\sigma(X)\sigma(Y)}.$$
(187)

It is easy to show that  $-1 \leq \operatorname{Corr}(X, Y) \leq 1$ . The correlation in the understanding of statistics can be regarded as a normalized covariance. It has a value of 1 if X and Y are identical up to some positive scaling factor, it has a value of -1 if X and Y are identical up to some negative scaling factor. When  $\operatorname{Corr}(X,Y) = 0$ , X and Y are said to be *uncorrelated*.

The quantity  $\operatorname{Corr}(X, Y)$  is also referred to as *(population) Pearson's correla*tion coefficient, and is often denoted by the greek letter  $\rho(X, Y) = \operatorname{Corr}(X, Y)$ .

In the signal processing literature (for instance in my favorite textbook Farhang-Boroujeny 1998), the term "correlation" is sometimes used in quite a different way, denoting the quantity

E[XY],

that is, simply the expectation of the product of the uncentered RVs X and Y. Just be careful when you read terms like "correlation" or "cross-correlation" or "cross-correlation matrix" and make sure that your understanding of the term is the same as the respective author's.

There are some basic rules for doing calculations with expectations and covariance which one should know: 1. Expectation is a linear operator:

$$E[\alpha X + \beta Y] = \alpha E[X] + \beta E[Y],$$

where  $\alpha\,X$  is the RV obtained from X by scaling observations with a factor  $\alpha.$ 

2. Expectation is idempotent:

$$E[E[X]] = E[X].$$

3.

$$\operatorname{Cov}(X, Y) = E[X Y] - E[X] E[Y].$$

# E Derivation of Equation 32

$$1/N \sum_{i} ||x_{i} - \mathbf{d} \circ \mathbf{f}(x_{i})||^{2} =$$

$$= 1/N \sum_{i} ||\bar{x}_{i} - \sum_{k=1}^{m} (\bar{x}'_{i} u_{k}) u_{k}||^{2}$$

$$= 1/N \sum_{i} ||\sum_{k=1}^{n} (\bar{x}'_{i} u_{k}) u_{k} - \sum_{k=1}^{m} (\bar{x}'_{i} u_{k}) u_{k}||^{2}$$

$$= 1/N \sum_{i} ||\sum_{k=m+1}^{n} (\bar{x}'_{i} u_{k}) u_{k}||^{2}$$

$$= 1/N \sum_{i} \sum_{k=m+1}^{n} (\bar{x}'_{i} u_{k})^{2} = \sum_{k=m+1}^{n} 1/N \sum_{i} (\bar{x}'_{i} u_{k})^{2}$$

$$= \sum_{k=m+1}^{n} \sigma_{k}^{2}.$$

### References

- D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. "A Learning Algorithm for Boltzmann Machines". In: *Cognitive Science* 9 (1985), pp. 147–169.
- [2] A. C. Antoulas and D. C. Sorensen. "Approximation of large-scale dynamical systems: an overview". In: *International Journal of Applied Mathematics and Computer Science* 11.5 (2001), pp. 1093–1121.
- [3] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: International Conference on Learning Representations (ICLR). 2015. URL: http://arxiv.org/abs/ 1409.0473v6.
- [4] J. A. Bednar and S. P. Wilson. "Cortical maps." In: The Neuroscientist 22.6 (2016), pp. 604–617.
- [5] Y. Bengio and Y. LeCun. "Scaling Learning Algorithms towards AI". In: Large-Scale Kernel Machines. Ed. by Bottou L. et al. MIT Press, 2007.
- [6] M. Bradie. "Assessing evolutionary epistemology". In: Biology and Philosophy 1.4 (1986), pp. 401–459.
- [7] L. Breiman. "Random forests". In: Machine Learning 45 (2001), pp. 5–32.
- [8] L. Buesing et al. "Neural Dynamics as Sampling: A Model for Stochastic Computation in Recurrent Networks of Spiking Neurons". In: *PLoS Comp. Biol.* 7.11 (2011), e1002211.
- C. Callender. "Introduction". In: The Oxford Handbook of Philosophy of Time. Ed. by C. Callender. Oxford University Press, 2011.
- [10] A. Clark. "Whatever Next? Predictive Brains, Situated Agents, and the Future of Cognitive Science." In: *Behavioral and Brain Sciences* 36.3 (2013), pp. 1–86.
- [11] G. E. Crooks. Field Guide to Continuous Probability Distributions, v 0.11 beta. online manuscript, retrieved April 2017, extended version also available in print since 2019. 2017. URL: http://threeplusone.com/fieldguide.
- [12] H. Cruse, J. Dean, and H. Ritter, eds. Prerational Intelligence: Adaptive Behavior and Intelligent Systems Without Symbols and Logic (3 volumes). Springer Science & Business Media, 2013.
- [13] A. Deisenroth, A. Faisal, and C. S. Ong. Mathematics for Machine Learning. Free online copy at https://mml-book.github.io/. Cambridge University Press, 2019.
- [14] A.P. Dempster, N.M. Laird, and D.B. Rubin. "Maximum likelihood from incomplete data via the EM-algorithm". In: *Journal of the Royal Statistical Society* 39 (1977), pp. 1–38.

- [15] S. Demyanov. "Regularization Methods for Neural Networks and Related Models". PhD thesis. Dept of Computing and Information Systems, Univ. of Melbourne, 2015.
- [16] J. Devlin et al. BERT: Pre-training of deep bidirectional transformers for language understanding. Tech. rep. arXiv manuscript https://arxiv.org/ pdf/1810.04805.pdf. 2018.
- [17] G. L. Drescher. Made-up Minds: A Constructivist Approach to Artificial Intelligence. MIT Press, Cambridge, Mass., 1991.
- [18] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern Classification (second edition). Wiley Interscience, 2001.
- [19] R. P. W. Duin and D. M. J. Tax. "Experiments with Classifier Combining Rules". In: Proc. First International Workshop on Multiple Classifer Systems, MCS 2000. Ed. by J. Kittler and R. Roli. LNCS 1857. http://ict.ewi.tudelft.nl/~duin/papers/mcs\_00\_classcomb.pdf. Springer, Berlin, 2000, pp. 16–29.
- [20] R. Durbin et al. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, 2000.
- [21] D. Durstewitz, J. K. Seamans, and T. J. Sejnowski. "Neurocomputational models of working memory". In: *Nature Neuroscience* 3 (2000), pp. 1184– 91.
- [22] S. Edelman. "The minority report: some common assumptions to reconsider in the modelling of the brain and behaviour". In: Journal of Experimental and Theoretical Artificial Intelligence (2015). URL: http://www. tandfonline.com/action/showCitFormats?doi=10.1080/0952813X. 2015.1042534.
- [23] S. Ermon. Probabilistic Graphical Models. Online lecture notes of a graduate course at Stanford University. 2019. URL: https://ermongroup.github. io/cs228-notes/.
- [24] B. Farhang-Boroujeny. Adaptive Filters: Theory and Applications (1rst edition). 2nd revised and extended edition 2013. Wiley, 1998.
- [25] R. Fischer. "Development and Evaluation of Causal Models With an Application to Orthopaedic Inquiries". https://www.ai.rug.nl/minds/ uploads/3924\_Fischer23.pdf. Master's thesis. Harvard Medical School and University of Groningen, Jan. 2023.
- [26] K. Friston. "A theory of cortical response". In: Philosophical Transactions of the Royal Society London, Series B 360 (2005), pp. 815–836.
- [27] K. Friston. "Learning and Inference in the Brain". In: Neural Networks 16 (2003), pp. 1325–1352.

- [28] S. Fusi and X.-J. Wang. "Short-term, long-term, and working memory". In: From Neuron to Cognition via Computational Neuroscience. Ed. by M. Arbib and J. Bonaiuto. MIT Press, 2016, pp. 319–344.
- [29] J. J. Gibson. "The Theory of Affordances". In: *Perceiving, Acting, and Knowing*. Ed. by R. Shaw and J. Bransford. Lawrence Erlbaum Ass., Hillsdale, New Jersey, 1979, pp. 127–143.
- [30] I. J. Goodfellow, J. Shlens, and C. Szegedy. "Explaining and Harnessing Adversarial Examples". In: *Proc. ICLR 2015*. arXiv:1412.6572v3. 2014.
- [31] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Open access version at http://www.deeplearningbook.org. MIT Press, 2016.
- [32] A. N. Gorban et al. Principal Manifolds for Data Visualization and Dimension Reduction. Springer, 2008.
- [33] A. Graves et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 7626 (2016), pp. 471–476.
- [34] A. Hart, J. Hook, and J. Dawes. "Embedding and approximation theorems for echo state networks". In: *Neural Networks* 128 (2020), pp. 234–247.
- [35] M. Hasegawa, H. Kishino, and T. Yano. "Dating the human-ape splitting by a molecular clock of mitochondrial DNA". In: J. of Molecular Evolution 22 (1985), pp. 160–174.
- [36] X. He. "Continual lifelong learning in neural systems: overcoming catastrophic forgetting and transferring knowledge for future learning". https://doi.org/10.33612 PhD thesis. School of Engineering and Science, University of Groningen, 2023.
- [37] G. E. Hinton and R. R. Salakuthdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.July 28 (2006), pp. 504–507.
- [38] E. Horvitz and M. Barry. "Display of information for time-critical decision making". In: Proc. 11th Conf. on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., 1995, pp. 296–305.
- [39] C. Huang and A. Darwiche. "Inference in Belief Networks: A Procedural Guide". In: Int. J. of Approximate Reasoning 11.1 (1994), p. 158.
- [40] J. P. Huelsenbeck and F. Ronquist. "MRBAYES: Bayesian inference of phylogenetic trees". In: *Bioinformatics* 17.8 (2001), pp. 754–755.
- [41] L. Hyafil and R. L. Rivest. "Computing optimal binary decision trees is NP-complete". In: *Information Processing Letters* 5.1 (1976), pp. 15–17.
- [42] G. Indiveri. Rounding Methods for Neural Networks with Low Resolution Synaptic Weights. arXiv preprint. Institute of Neuroinformatics, Univ. Zurich, 2015. URL: http://arxiv.org/abs/1504.05767.

- [43] H. Jaeger. "Echo State Network". In: Scholarpedia. Vol. 2. 2007, p. 2330. URL: http://www.scholarpedia.org/article/Echo%5C\_State%5C\_ Network.
- [44] H. Jaeger et al. "Optimization and Applications of Echo State Networks with Leaky Integrator Neurons". In: *Neural Networks* 20.3 (2007), pp. 335– 352.
- [45] E. T. Jaynes. Probability Theory: the Logic of Science. First partial online editions in the late 1990ies. First three chapters online at http://bayes.wustl.edu/etj/prob/book.pdf. Cambridge University Press, 2003.
- [46] D. Jones. Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications. technical report, published online. UCL Bioinformatics, 2010. URL: http://www.cs.ucl.ac.uk/staff/d.jones/ GoodPracticeRNG.pdf.
- [47] M. I. Jordan, Z. Ghahramani, et al. "An introduction to variational methods for graphical models". In: *Machine Learning* 37.2 (1999), pp. 183–233.
- [48] M. I. Jordan and D. M. Wolpert. "Computational motor control". In: The Cognitive Neurosciences, 2nd edition. Ed. by M. Gazzaniga. MIT Press, 1999.
- [49] A. Karmiloff-Smith. "Precis of: Beyond Modularity: A Developmental Perspective on Cognitive Science". In: *Behaviour and Brain Sciences* 17.4 (1994), pp. 693–707.
- [50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680.
- [51] R. Kiros, R. Salakhutdinov, and R. S. Zemel. "Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models". http://arxiv.org/abs/1411.2539 Presented at NIPS 2014 Deep Learning Workshop. 2014.
- [52] D. Kutach. "The Asymmetry of Influence". In: *The Oxford Handbook of Philosophy of Time*. Ed. by C. Callender. Oxford University Press, 2011. Chap. 8, pp. 247–275.
- [53] G. Lakoff and R. Núñez. "The metaphorical structure of mathematics: sketching out cognitive foundations for a mind-based mathematics". In: *Mathematical Reasoning*. Routledge, 2013, pp. 29–98.
- [54] G. Lakoff and R. E. Núñez. Where mathematics comes from: How the embodied mind brings mathematics into being. Basic Books, 2000.
- [55] S.L. Lauritzen. "The EM algorithm for graphical association models with missing data". In: Computational Statistics & Data Analysis 19.2 (1995), pp. 191–201.
- [56] D. Luchinsky and et al. "Overheating Anomalies during Flight Test due to the Base Bleeding". In: *Proc. 7th Int. Conf. on Computational Fluid Dynamics, Hawaii July 2012.* 2012.

- [57] B. Mau, M.A. Newton, and B. Larget. "Bayesian phylogenetic inference via Markov chain Monte Carlo methods". In: *Biometrics* 55 (1999), pp. 1-12. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.
   33.8433&rep=rep1&type=pdf.
- [58] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: Bull. of Mathematical Biophysics 5 (1943), pp. 115– 133.
- [59] N. Metropolis et al. "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092.
- [60] T. Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: Advances in Neural Information Processing Systems 26. Ed. by C. J. C. Burges et al. 2013, pp. 3111-3119. URL: http: //papers.nips.cc/paper/5021-distributed-representations-ofwords-and-phrases-and-their-compositionality.pdf.
- [61] A. Minnaar. Wor2Vec tutorial part I: the Skip-Gram model. Online tutorial. 2015. URL: http://mccormickml.com/2016/04/27/word2vecresources/%5C#efficient-estimation-of-word-representationsin-vector-space.
- [62] T. M. Mitchell. Machine Learning. McGraw-Hill, 1997.
- [63] S. R. T. Mouafo et al. "A tutorial on the EM algorithm for Bayesian networks: application to self-diagnosis of GPON-FTTH networks". In: Proc. 12th International Wireless Communications & Mobile Computing Conference (IWCMC 2016). 2016, pp. 369–376. URL: https://hal.archivesouvertes.fr/hal-01394337.
- [64] A. Müller et al. "Towards Real-World Deployment of Reinforcement Learning for Traffic Signal Control". In: Proc. 20th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2021, pp. 507– 514.
- [65] K. Murphy. An introduction to graphical models. Technical Report. http://www.cs.ubc.ca/~ Intel, 2001.
- [66] K. P. Murphy. "Dynamic Bayesian Networks: Representation, Inference and Learning". https://www.academia.edu/download/32395182/thesis. pdf. PhD thesis. Univ. of California, Berkeley, 2002.
- [67] R. M. Neal. Probabilistic Inference Using Markov Chain Monte Carlo Methods. Technical Report CRG-TR-93-1. Dpt. of Computer Science, University of Toronto, 1993.
- [68] R. M. Neal. Using Deterministic Maps when Sampling from Complex Distributions. Presentation given at the Evolution of Deep Learning Symposium in honor of Geoffrey Hinton. 2019. URL: http://www.cs.utoronto.ca/ ~radford/ftp/geoff-sym-talk.pdf.

- [69] K. Obermayer, H. Ritter, and K. Schulten. "A principle for the formation of the spatial structure of cortical feature maps". In: *Proc. of the National Academy of Sciences of the USA* 87 (1990), pp. 8345–8349.
- [70] O. M. Parkhi, A. Vedaldi, and A. Zisserman. "Deep Face Recognition". In: Proc. of BMVC. 2015. URL: http://www.robots.ox.ac.uk:5000/~vgg/ publications/2015/Parkhi15/parkhi15.pdf.
- [71] R. Pascanu and H. Jaeger. "A Neurodynamical Model for Working Memory". In: *Neural Networks* 24.2 (2011). DOI: 10.1016/j.neunet.2010.10.003, pp. 199–207.
- [72] J. Pearl. "[Commenatary on: Bayesian analysis in expert systems] graphical models, causality and intervention". In: *Statistical Science* 8.3 (1993), pp. 266–269.
- [73] J. Pearl and S. Russell. "Bayesian Networks". In: Handbook of Brain Theory and Neural Networks, 2nd Ed. Ed. by M.A. Arbib. MIT Press, 2003, pp. 157–160. URL: https://escholarship.org/uc/item/53n4f34m.
- [74] J. Pearl and T. S. Verma. "A statistical semantics for causation". In: Statistics and Computing 2.2 (1992). this article apparently is a workout of a 1991 conference contribution of the same authors, which however is not available online., pp. 91–95.
- S. E. Peters et al. "A Machine Reading System for Assembling Synthetic Paleontological Databases". In: *PLOS-ONE* 9.12 (2014), e113523. URL: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0113523.
- [76] R. Pfeifer and Ch. Scheier. Understanding Intelligence. MIT Press, 1999.
- [77] J. Piaget. "The biological problem of intelligence". In: Organization and pathology of thought: Selected sources. Ed. by D. Rapaport. engl. transl. of the introductory chapter of Piaget's La Naissance de l'Intelligence (1936). Columbia University Press, 1951. Chap. 7, pp. 176–192.
- [78] R. F. Port and T. van Gelder, eds. Mind as Motion: Explorations in the Dynamics of Cognition. MIT Press/Bradford books, 1995.
- [79] L.R. Rabiner. "A tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *Readings in Speech Recognition*. Ed. by A. Waibel and K.-F. Lee. Reprinted from Proceedings of the IEEE 77 (2), 257-286 (1989). Morgan Kaufmann, San Mateo, 1990, pp. 267–296.
- [80] F. Rosenblatt. "The Perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6 (1958), pp. 386–408.
- [81] S. Roweis and Z. Ghahramani. "A unifying review of linear Gaussian models". In: *Neural Computation* 11.2 (1999), pp. 305–345.

- [82] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing Vol. 1*. Ed. by D. E. Rumelhart and J. L. McClelland. Also as Technical Report, La Jolla Inst. for Cognitive Science, 1985. MIT Press, 1986, pp. 318– 362.
- [83] S. Russell. Judea Pearl. https://amturing.acm.org/award\_winners/ pearl\_2658896.cfm. Review of scientific achievements of Judea Pearl that led to the Turing Award. Accessed: January 2023. 2011.
- [84] M. A. Savi. "Nonlinear Dynamics and Chaos". In: Dynamics of Smart Systems and Structures. Ed. by V. Lopes Junior and et al. Springer International Publishing Switzerland, 2016, pp. 93–117.
- [85] J. Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: Neural Networks 61 (2015). Preprint: arXiv:1404.7828, pp. 85–117.
- [86] B. Schölkopf and J. von Kügelgen. From Statistical to Causal Learning. arXiv manuscript https://arxiv.org/abs/2204.00607. 2022.
- [87] B. Schölkopf, F. Locatello, et al. "Towards Causal Representation Learning". In: *Proceedings of the IEEE* 109.5 (2021). arxiv preprint at https: //arxiv.org/abs/2102.11107, pp. 612-634.
- [88] C. E. Shannon. "A mathematical theory of communication". In: The Bell System Technical Journal 27.3 (1948), pp. 379–423.
- [89] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489.
- [90] P. Smyth. "Belief Networks, Hidden Markov Models, and Markov Random Fields: a Unifying View". In: *Pattern Recognition Letters* 18.11-13 (1997), pp. 1261–1268.
- [91] F. Suchanek et al. "Advances in automated knowledge base construction". In: SIGMOD Records Journal March (2013). URL: http://suchanek. name/work/publications/sigmodrec2013akbc.pdf.
- [92] N. V. Swindale and H.-U. Bauer. "Application of Kohonen's self-organizing feature map algorithm to cortical maps of orientation and direction preference". In: *Proc. R. Soc. Lond. B* 265 (1998), pp. 827–838.
- [93] F. Takens. "Detecting strange attractors in turbulence". In: Dynamical Systems and Turbulence. Ed. by D.A. Rand and L.-S. Young. Lecture Notes in Mathematics 898. Springer-Verlag, 1981, pp. 366–381.
- [94] M. Tangemann et al. Unsupervised Object Learning via Common Fate. arxiv report https://arxiv.org/pdf/2110.06562. 2021.
- [95] J. Tenenbaum, T. L. Griffiths, and C. Kemp. "Theory-based Bayesian models of inductive learning and reasoning". In: *Trends in Cognitive Science* 10.7 (2006), pp. 309–318.
- [96] E. Thelen. "Grounded in the world: Developmental origins of the embodied mind". In: Infancy 1.1 (2000), pp. 3–28.
- [97] A. Tversky and D. Kahneman. "Judgment under Uncertainty: Heuristics and Biases: Biases in judgments reveal some heuristics of thinking under uncertainty". In: Science 185.4157 (1974), pp. 1124–1131.
- [98] M. Weliky, W. H. Bosking, and D. Fitzpatrick. "A systematic map of direction preference in primary visual cortex". In: *Nature* 379 (1996), pp. 725– 728.
- [99] X. Xie and H. S. Seung. "Learning in neural networks by reinforcement of irregular spiking". In: *Phys Rev E Stat Nonlin Soft Matter Phys* 69 (2004), p. 041909.
- H. Yin. "Learning nonlinear principal manifolds by self-organising maps". In: *Principal Manifolds for Data Visualization and Dimension Reduction*. Ed. by A. N. Gorban et al. Vol. 58. Lecture Notes in Computer Science and Engineering. Springer, 2008, pp. 68–95.
- [101] P. Young et al. "From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions." In: Transactions of the Association for Computational Linguistics 2 (2014), pp. 67– 78.
- [102] A. Zhang et al. Dive into deep learning. published on arXiv https:// arxiv.org/abs/2106.11342. 2021.