JACOBS
UNIVERSITY

Herbert Jaeger

Machine Learning

Lecture Notes

V 1.5, April 17, 2018

BSc Program IMS

Contents

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" run 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 is as different from any of the former as cars are from horses.

Human learning is incredibly scintillating and elusive. It is as complex and impossible to be fully understood as you can't fully understand yourself. Think of *all* the things you can *do*, all of your body motions from tying your shoes to playing the guitar; 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 — and almost everything of that wild collection is the result of a fabulous mixing learning of some kind 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 · physiology and neurophysiology · body growth · brain development · motion control · exploration, curiosity, play · creativity · social interaction · drill and exercise and rote learning · reward and punishment, pleasure and pain · the universe, the earth, the atmosphere, water, food, caves· evolution · dreaming · remembering · forgetting · aging · other people, living · other people, long dead · machines, tools, buildings, toys· words and sentences· concepts and meanings · letters and books and schools · traditions ...

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 is automated image caption (technical report: Kiros et al. (2014)). At a website of one of the "deep learning" pioneers, Geoffrey Hinton from the University of Toronto, you can find stunning examples of caption phrases that have been automatically generated by a neural network which was given a photographic image as input (check out the static demo at `http://www.cs.toronto.edu/~nitish/nips2014demo`). Figure 1 shows some screenshots. Other fascinating examples of deep learning are face recognition (e.g., Parkhi et al. (2915)), online text translation (e.g., Bahdanau et al. (2015)), inferring a Turing machine (almost) from input-output examples (Graves, 2016), or playing the game of Go at and beyond the level of human grand-masters (Silver et al, 2016).

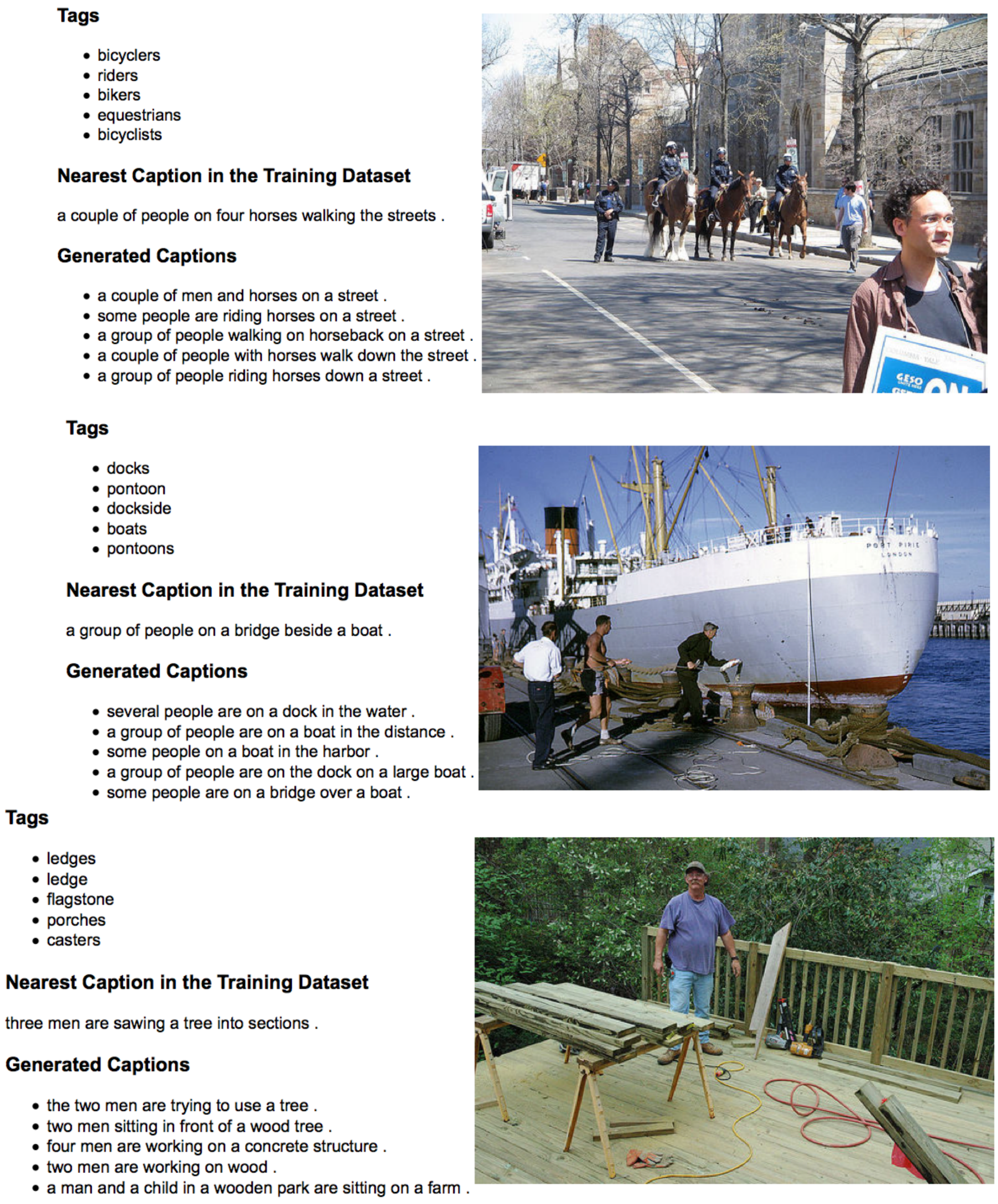
So, apparently machine learning algorithms come close to human performance in several tasks or even surpass humans, and these performance achievements have been *learnt* by the algorithms, — thus, machines today can learn like humans??!? The answer is NO. ML researchers are highly aware of this. Outside ML however, naive beholders (from the popular press, politics, or other sciences) often conclude that since learning machines can perform 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 at this point. 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 speed. Beyond that brute force component, computer chess programs are just clever graph search algorithms. Similarly, the celebrated ML demos rely on brute force. Their force is however not burnt on graph search as in classical AI algorithms, but it is used to represent and estimate complex probability distributions. Progress has accelerated in the last few years because increasingly large training datasets have become available and technically manageable, and fast enough computing systems (concretely, GPU clusters) have become cheap enough for university research teams.

This is not to say that powerful ML "just" means large datasets and fast machines. These are necessary but not sufficient. In addition to these technological preconditions, 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 the remainder of this course.

Take-home message: The astonishing learning feats of today's ML are based on statistical modeling techniques, raw processing power and a lot of a researcher's personal experience and trial-and-error optimization. It's technology and maths, not psychology. Dismiss any romantic ideas about ML. ML is stuff for sober engineers. But they are allowed to become very excited about that stuff, and that stuff can move mountains.

2 ML as Modeling Probability Distributions

In this section I want to explain, still on a largely informal level, that large parts of ML can be understood as the art of estimating probability distributions from data. I will also use this occasion to highlight some typical challenges that ML projects often face.



Tags

- bicyclers
- riders
- bikers
- 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 neural-network based image caption demo at http://www.cs.toronto.edu/~nitish/nips2014demo.

2.1 Introducing TICS: a recent highlight example of machine learning

I will use the demo task highlighted in Figure 1 as an example. Let us follow the sacred tradition of ML and first introduce an acronym that is not understandable to outsiders: TICS = the Toronto Image Caption System.

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 the screenshots in Figure 1). In fact, TICS not only just outputs a list of caption suggestions, but this list is 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 make this more formal. TICS captions are based on a finite vocabulary of English words (and their grammatical inflection forms, like "try", "tried", "trying" etc) that the system designers fixed at design time. For simplicity let us assume that this vocabulary contains 10,000 words (that would be a typical size for today's ML systems that handle simple natural language texts; I did not find the vocabulary size documented in the scientific report). Let us furthermore assume for even more simplicity that the length of captions that TICS can generate is bounded, say to a maximum of 20 words. This means that there are $N = 10,000^{20} + 10,000^{19} + \ldots + 10,000$ different possible word sequences. Let $C = \{s_1, \ldots, s_N\}$ denote the set of these sequences — the set of potential captions. C is a very large set. When TICS (after training) is run on an input test image x_{test}, the system implicitly uses a learnt probability distribution over C to return the five word sequences that have the highest probability.

Some technical-mathematical remarks are in place at this point. C is a finite set (though large). Therefore, a probability distribution P over C is, mathematically speaking, just an N-dimensional probability vector, that is, a vector $(P(s_1), \ldots, P(s_N))$ satisfying (i) $P(s_i) \geq 0$ for all $1 \leq i \leq N$ and (ii) $\sum_i P(s_i) = 1$, where $P(s_i)$ is the probability of the sequence s_i. Simple as this set-up looks, it harbors two problems.

Problem 1: Since N is huge, the total probability mass of 1 necessarily must be distributed over the N components s_i in very tiny portions. If you want to code these probabilities on your digital computer, numerical underflow will occur: most probabilities $P(s_i)$ will be smaller than the numerical precision of your computer and thus become represented as zero. *Solution 1*: use log probabilities instead of raw probabilities. That is, instead of storing $P(s_i)$, store $\log P(s_i)$. In fact, it is common practice in ML formalisms and ML programming to work with "log-probs" instead of with the raw probilities. *Solution 2*: don't calculate probabilities at all. For practical use, it is not the probabilities per se but the *ratios* of probabilities that are important. For instance, it is enough for figure caption generation to know that the most probable caption candidate is 3 times as probable as the second-best, which in turn is 1.3 times more probable than the third, etc. In fact many ML algorithms do not calculate or output raw or log probabilites but only probability ratios.

Problem 2: It is impossible to completely compute a vector of N numbers when N is as large as in TICS — memory overflow and time overflow prevent this. *Solution*: the TICS algorithm must be set up in a clever way such that it can find the most probable candidate (and its next-best companions) without first computing all probabilities and then selecting the highest. This can be a tricky affair.

Back to our main business of outlining the probability story behind TICS. We have seen that in essence (though technical realisation may differ), TICS upon input of a test image x_{test} computes a probability distribution on C. For different test images x_k, x_l these distributions over candidate captions will differ: if the image x_k shows a flying bird, the caption bird in the sky will receive a higher probability than the caption car on a highway; for x_l showing a car on a road the second caption will get a higher probability. Thus, TICS has to compute *conditional probabilities* $P(\mathsf{Caption} = s_i \,|\, \mathsf{Picture} = x_j)$ of captions s_i given input pictures x_j.

Conditional probabilities abound in ML. One could even say that (almost) every probability that arises in real-world modeling tasks is a conditional probability, since (almost) all interesting modeling tasks aim at elucidating how probabilities of some *effects* vary when *causes* are changing. That is, the most interesting probabilities encountered in applications look like $P(\mathsf{Effect} = y \,|\, \mathsf{Cause} = x)$. In our case, the causes are input images and the effects are figure captions.

At this point - don't read on unless you have (re-)familiarized yourself with the concept of conditional and joint probabilities. You find a short rehearsal in Appendix A.

2.2 From discrete to continuous probability distributions

You will have heard buzzwords like "big data" or "deep learning", highlighting current developments in data analytics and machine learning. A characteristic of these developments is that very large datasets are being analyzed. This has become possible only in the last few years due to technological advances (availability of large datasets, cloud computing, large RAM's, use of GPUs) and the development of novel ML algorithms that can cope with large datasets. The TICS demo is a good example. Often the data that one wishes to process come in a mix of discrete and continuous formats. Again TICS is a good example:

- Images are continuous-valued data. More specifically, an RGB image that has K pixels can be formally seen as a real-valued vector of size $3K$, where each value in this vector is the red, green or blue intensity of one of the image's pixels. If the image is merely 600×800 pixel sized (small for today's standards), this alread turns an image into a $3*600*800 = 1,440,000$ dimensional vector. Assuming that color intensities are coded as reals between 0 and 1, the "image space" in which mathematical formalism places images is the hypercube $I = [0,1]^{1,440,000}$. A LARGE! dimension indeed, and quite a different animal from the typically 2- or 3-dimensional vectors you encountered in mathematics classes.

- Bounded-length captions made from a finite vocabulary form a finite set and hence are, mathematically speaking, discrete data. However, the number of possible different data points is astronomical. Again, such a very large (yet finite) set is quite a different animal than the pocket-sized finite sets that have been drawn on the blackboard by your math teacher in set theory classes.

Similarly, in other typical "big data" or "deep learning" applications, when data are discrete, they usually come in sets of very large size. For practical reasons (limited processing times and limited computer memory space) one cannot easily handle such sets.

Another difficulty with discrete data arises when they have to be combined with continuous data. The machinery behind TICS, for instance, has to combine information about images (continuous) with information about word sequences (discrete) in order to produce meaningful results.

On the other hand, when one works with purely continuous data spaces, the existence of very many data points does not present headaches. Mathematically, continuous data spaces are typically set up as *vector spaces*. A vector space contains infinitely many possible data points from the outset, and the mathematics of vector spaces is just made to deal with infinite point sets – for instance, lines, subspaces, and objects called *manifolds* that we'll inspect soon.

Thus, a standard strategy in modern machine learning (especially in deep learning) is to transform any discrete data that one has to deal with, into vector data at the beginning of a day's work. Such a discrete-to-continuous transformation can be done in many ways, and which method one uses depends on the task at hand. Here are two examples:

- A simple case: assume the discrete data are records of "yes/no" answers given in some questionnaire. A discrete data point is then a list of N "yes/no" alternatives, where N is the number of questions in the questionnaire. This can be turned into a real vector by just replacing "yes" by the number 1.00 and "no" by the number 0.00, leading to a binary numerical vector in $x \in \mathbb{R}^N$. Similarly, any symbolic data record can be transformed into a numerical vector format by replacing symbols with numbers according to a look-up table.

- A more advanced case: In TICS, each word w_i from the used caption vocabulary was transformed into an 300-dimensional vector $v_i \in \mathbb{R}^{300}$. The tricky part is *how* this transformation $w_i \mapsto v_i$ was computed. The guiding idea is to find a mapping that encodes semantic similarity. For instance, code vectors v, v' for words $w = \mathsf{airplane}$ and $w' = \mathsf{aircraft}$ should be similar, wheras the code vector v'' for $w'' = \mathsf{rose}$ should be dissimilar to both v and v'. Similarity of vectors can be quantified conveniently by metric distance, 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 was achieved in a preprocessing stage where semantic similarity of two words w, w' was assessed by measuring how often they occur in similar locations in phrase contexts. To this end, large collections of English texts were 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?). Check out Mikolov et al. (2013) for a standard reference on this technique – this paper has been (Google-scholar) cited more than 3,000 times in merely three years!

I emphasize that while a discrete-to-continuous transformation is often done, it is not always done. One can also go the other way: unify formalism and algorithms by making everything discrete. The simplest method for discretizing continuous vector data $v_i \in \mathbb{R}^n$ is *binning*: The region $\mathcal{D} \subseteq \mathbb{R}^n$ in which the observed vectors v_i fall is partitioned into a finite number of hypercube subregions $\mathcal{D} = \mathcal{H}_1 \dot\cup \ldots \dot\cup \mathcal{H}_K$ (where $\dot\cup$ denotes disjoint union). A continuous vector falling into the hypercube $\mathcal{H}_i$ is then transformed to the integer i (and integers are discrete objects).

Generally speaking, the continuous vs. discrete data representation choice leads to a landmark divide in machine learning techniques. Continuous data are the typical choice for neural network based techniques, where the background mathematics is linear algebra and calculus. On the other hand, discrete data formats are often processed with algorithms that are based on logic calculi and/or graph-oriented statistical methods, so-called graphical models.

At any rate, for the remainder of this introductory section, we'll be looking at vector data only.

2.3 A curse

In order to output the five highest-probability captions $c_1, \ldots, c_5$ given an input image x, the TICS system must compare conditional probabilities of the kind $P(\mathsf{Caption} = c \,|\, \mathsf{Image} = x)$. Since we have agreed to work with vector data, these probability comparisons are actually comparisons between pdf's, not probability values, so more correctly, TICS must compare pdf values of the kind $f_{\mathsf{Caption}|\mathsf{Image}}(c \,|\, x)$. Recall (see Appendix A) that these conditional pdf's are the

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 2: Two images and their annotations from the training data set. Taken from Young et al. (2014).

quotients

$$f_{\mathsf{Caption}|\mathsf{Image}}(c\,|\,x) = \frac{f_{\mathsf{Caption},\mathsf{Image}}(c,x)}{f_{\mathsf{Image}}(x)},$$

TICS must be able to compute – explicitly or implicitly – the joint probability density $f_{\mathsf{Caption},\mathsf{Image}}(c,x)$ and the marginal pdf $f_{\mathsf{Image}}(x)$. We will now take a closer look at the latter.

So, let us address the question of describing a pdf in the image space $I = [0,1]^{1,440,000}$.

In order to visualize the following arguments, let us dramatically simplify the story. Assume that the image space I were only two-dimensional, that is, assume $I = [0,1]^2$ instead of $I = [0,1]^{1,440,000}$ (a two-dimensional image vector would correspond to a two-pixel grayscale "image"). Thus, images are just points in the 2-dimensional unit square. Assume furthermore that the training set only consists of the two images shown in Figure 2. Then the training data can be visualized as in Figure 3.

Now consider a test image x_{test} presented to TICS (red cross in Figure 3). For concreteness assume that this test image is the harbor scene picture from Figure 1. The task for TICS is to generate some captions for this test image x_{test}. The only information that TICS can use to solve this task is the training dataset.

How on earth should that be possible? The images from the training dataset (the businessman and the butcher images) seem unrelated to the harbor image. How should TICS learn anything useful from those different pictures to bring to bear on the test image caption finding task? To use a core technical term from machine learning: how on earth should TICS be able to *generalize* from the training data to test data?

You might think that TICS should work as follows to solve this task. In order to generate meaningful captions for x_{test}, TICS should search through the training data, find training images x_i that are "similar" to x_{test}, then generate captions that are somehow interpolations between the known training captions for those similar-to-x_{test} training images.

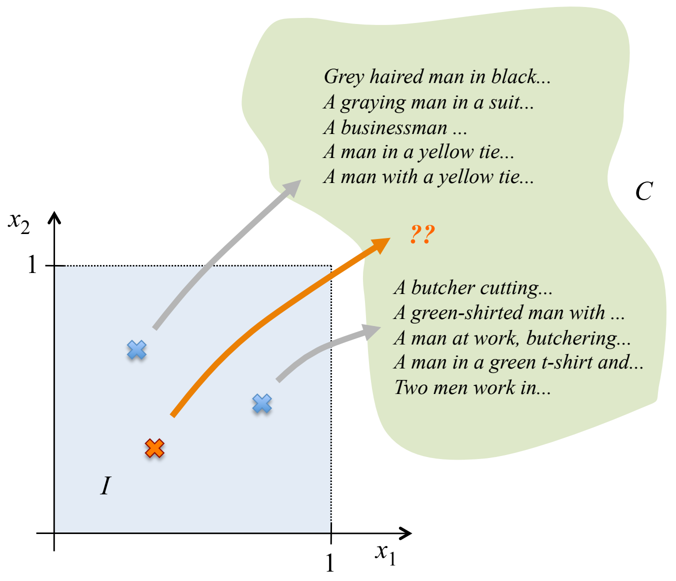

Grey haired man in black...
A graying man in a suit...
A businessman ...
A man in a yellow tie...
A man with a yellow tie...
C
x2
1
??
A butcher cutting...
A green-shirted man with ...
A man at work, butchering...
A man in a green t-shirt and...
Two men work in...
I
1
x1

Figure 3: Training data for TICS (highly simplified). The image dataspace I is the N-dimensional unit hypercube (here $N = 2$, shown in light blue shading), the two training images are points in this space (blue crosses indicating our 2-image demo training dataset), and the associated 5 captions per image are elements of a word sequence space C (indicated by greenish cloud). The test image is marked by an orange cross.

In Figure 3 this strategy wouldn't work because there are no apparent "close-by" training images for the test image x_{test}. So you might suspect that this trimmed-down caricature with a 2-dimensional image space and only two training examples is an unfair rendering of the actual situation. But for the full-size TICS system the situation is even far worse than this visualization suggests. This is because the actual image space is 1,440,000-dimensional, but there are only 30,000 training images. There are 1,440,000 / 30,000 = 48 times more dimensions than data points! In our caricature this ratio is much better: 2 dimensions, 2 images — as many images as dimensions. Thus, the actual TICS training data are very thinly spread over the image data space I (much more thinly than in the caricature). Even worse: the longest diagonal in the 1,440,000-dimensional unit hypercube has a length of $\sqrt{1440000} = 1200$. Not only can the training images "cover" only a subspace in I whose dimension is a mere 1/48-th of the embedding space dimension, but training images will be metrically spaced far apart from each other. We will not dig deeper into the geometry of high-dimensional metric spaces at this point (humans can't really image such spaces, all our intuitions are confined to 3-dimensional spaces). The message that I want to bring home here: The training images are *exceedingly thinly* spread over the data space I, they have large distances from each other, and a test image will likewise typically have a large distance from *all* of the training images. TICS wouldn't be able to detect "similar" training examples for x_{test} simply because there aren't any — at least, if "similarity" is expressed in terms of metric closeness in data space.

In spite of all these apparent impossibilities, TICS works. How has the impossible been rendered possible? Answer: by a clever combination of differential geometry and statistics. I will now attempt to give an intuitive account of the geometrical statistics (or statistical geometry) underlying TICS. The picture that will emerge is by and large the current mainstream view on the nature of high-dimensional statistical modeling held in the "deep learning" subfield of ML.

Before we start, I need to sketch a concept from analytical geometry, *manifolds.* Consider the n-dimensional real vector space $\mathbb{R}^n$ (for our TICS, this would be I with n being 1440000). 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 the m-dimensional Euclidean coordinate system (Figure 4**A**).

1-dimensional manifolds are just lines embedded in some higher-dimensional $\mathbb{R}^n$ (Figure 4**B**), 2-dimensional manifolds are surfaces, etc. Manifolds can be wildly curved, knotted (as in Figure 4**C**), or fragmented (as in Figure 4**B**). Humans cannot visually imagine manifolds of dimension greater than 2.

Back to TICS. A key to why machine learning is not impossible is the fact that n-dimensional data points generated by real-world data generating environments typically lie on an m-dimensional submanifold embedded in $\mathbb{R}^n$. In our case of images, $n = 1440000$, but the TICS system actually is set up in a way that forces the images to lie on a $m = 4096$ dimensional manifold embedded in I. Compared to the raw data space dimension $n = 1440000$, this is a dramatically much lower-dimensional space. However, the 4096-dimensional manifold is curled and curved into the embedding space in an exceedingly complicated way (Figure 4**C** only gives a very faint impression of the unwieldy shape of real-world data manifolds).

The reason why real-world data vectors typically lie on relatively low-dimensional manifolds is that the real world is full of rules and regularities and symmetries and constraints. These regularities reduce the degrees of freedom by which real-world data vectors can vary. In the case of image vectors, for instance, some of these constraints would be

- neighboring pixels tend to have similar color,
- sudden changes in color tend to follow line segments,
- in a part of an image showing a face there will be a portion of that part that shows a nose,
- trees have many similar leaves,
- ... (you get the idea)

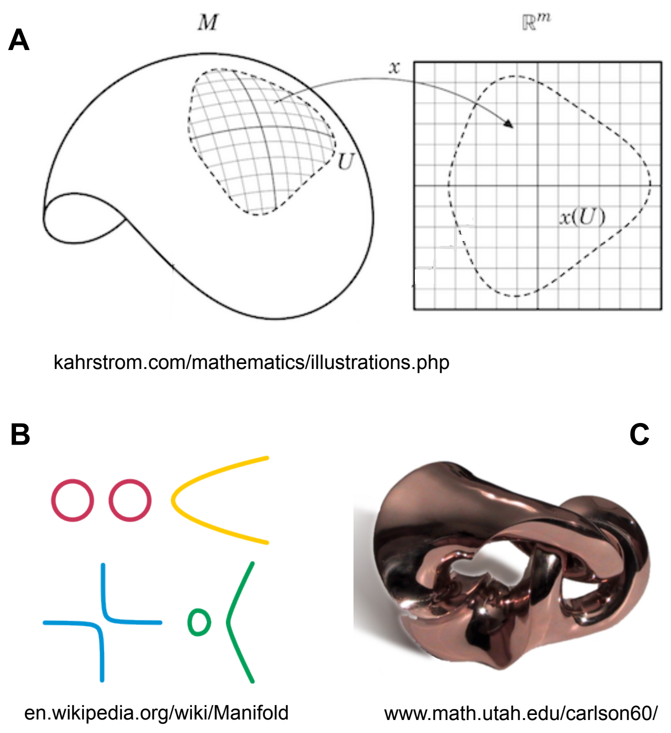

A
M
$\mathbb{R}^m$
x
U
x(U)
kahrstrom.com/mathematics/illustrations.php
B
C
en.wikipedia.org/wiki/Manifold
www.math.utah.edu/carlson60/

Figure 4: **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 next beneficial circumstance that helps to make TICS work is that there are more training data points than dimensions of the data manifold. Concretely in TICS there are about 30,000 caption-labelled images in the Flickr dataset that constituted the main training information. This relative abundance of training points enabled TICS to trace out the curled-up geometry of the $m = 4096$ dimensional target manifold – at least, approximately, and in some regions. Figure 5**A** shows how curved manifolds can be identified by a sufficient number of data points. In fact, when TICS was trained, another 70,000 images (without captions) were used in addition to the core Flickr dataset just for detecting the "image manifold".

But, real-world data contain random noise components and will not exactly fall on the manifold that one wishes to determine (schematic in Figure 5**B**). Determining the manifold thus implicitly requires the learning system to model the

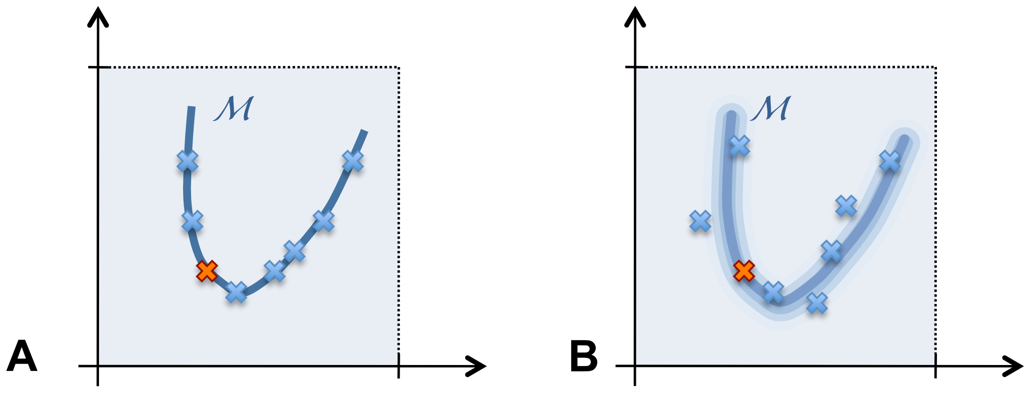


Figure 5: **A** When there are "enough" data points on a manifold available and when one assumes suitable smoothness criteria, a manifold can be determined ("learnt") from the datapoints (idealized schematic). **B** Real-world data are always randomly scattered around the ideal manifold.

probability distribution of data points.

(Sneak preview: We will later learn about *feedforward neural networks*, also often called *multilayer perceptrons* (MLPs). TICS uses such an MLP network to learn the manifold from training images. Such MLPs consist of a stack of layers of "neurons", where the lowest layer receives the input and the highest layer returns the output. In the TICS MLP, the input layer functions like a retina and has 1440000 neurons which code the pixel-color-channel values of the input image. The output layer has 4096 neurons whose activation values span a 4096-dimensional coordinate system, analog to the 2-dimensional coordinate system from Figure 4**A**.)

Summing up what we have seen so far: TICS contains an image processing submodule (an MLP) which transforms an input image x into a point f_x in a 4096-dimensional coordinate system. The transformed point f_x is called a *feature vector* representing x, and the 4096-dimensional coordinate system is called *feature space* F. In geometrical terms, this transformation from raw images x to feature vectors f_x achieves two important objectives:

- a dimension reduction from 1440000 to 4096 (factor of 350), which leads to a much better coverage density of the low-dimensional feature space with feature vectors, compared to the thin scattering of raw images in image space;

- an "un-curling", or "flattening-out" of the 4096-dimensional manifold. In the original 1440000-dimensional image space, this manifold is highly "curled", whereas the final 4096-dimensional feature representation can be considered as a "flat" coordinate system. The word "flat" should not be understood in the same way as we perceive a tabletop surface as being flat. Rather more abstractly, it means that images that should get similar captions have a small metrical distance on this manifold and vice versa.

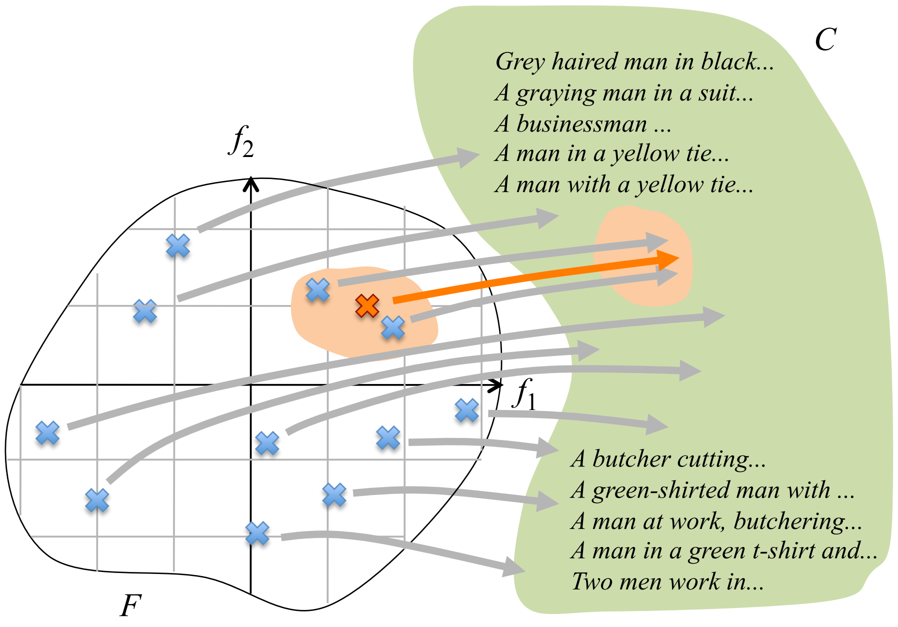

C
Grey haired man in black...
A graying man in a suit...
A businessman ...
A man in a yellow tie...
A man with a yellow tie...
f_2
f_1
A butcher cutting...
A green-shirted man with ...
A man at work, butchering...
A man in a green t-shirt and...
Two men work in...
F

Figure 6: The lower-dimensional feature space F is more densely covered by training image feature representations f_x, compared to the thinly spread-out population of raw images x in the original image space I. Each of the training image feature vectors f_x is associated with a set of 5 captions in C (only two are shown). Orange cross: a test image. Orange blobs: neighborhoods of test image in feature space and in caption space.

Figure 6 attempts to visualize this situation. TICS is now in a much better condition to solve the caption-generating problem. Intuitively speaking, the relatively dense coverage of feature vectors f_x in F makes it possible to meaningfully interpolate between them. A test image feature vector f_{test} (orange in Figure 6) can now be meaningfully related to "neighboring" training images f_x, and captions for f_{test} can be obtained by "interpolating" between the captions of those neighbors (light orange areas in Figure 6).

The technical details of how this "interpolating" is done are beyond the scope of this course. I can only give a simplified outline. Mathematically, TICS has to learn a family of conditional distributions $\{p(\mathsf{Caption} = c \mid \mathsf{Image} = x) \mid x \text{ is a real-world image}\}$. Note that we only need to learn caption-generation for images taken from the real world, not for arbitrary vectors $x_{\text{nonsense}} \in [0,1]^{1440000}$. Only those real-world images lie on (or close to) our 4096-dimensional manifold in I, and only for real-world images the feature transformation makes sense.

A core component of the TICS machine is thus a *dimension reduction* module. This module reduces the 1440000-dimensional picture input vector to *feature* vector which is only 4096-dimensional. All caption-relevant information in the raw picture is condensed into 4096 numbers which constitute this feature vector. The transformation from 1440000-dimensional raw input to a 4096-dimensional encoding of the picture's "essential meaning" is achieved by a deep neural network whose input layer has 1440000 units and whose output layer has only 4096 units. Much of the power and mystic of deep learning concerns the art to design and train such networks. After this dimension reduction, another submodule of TICS is trained to transform the 4096-dimensional feature vector into the caption hypotheses. In many deep learning architectures, this post-processing submodule which transforms a low-dimensional feature vector to the desired output is itself realized by a neural network, or by additional neural processing layers on top of the dimension-reducing subnetwork. Often however this post-processing module is more complex than just a few more neural layers. In TICS, the post-processing of the 4096-dimensional feature vector f_x utilizes the combined action of a so-called recurrent neural network, an elementary grammar model, and a vector-space based coding of word meaning, all of which are trained from data.

The TICS paper of Kiros et al. (2014) gives no account of the total number of parameters that need to be estimated by a training procedure for TICS, nor an account of the computer runtime required by the training algorithm. Similar state-of-the-art neural-network based models typically have several hundred thousands of parameters and the training time is in the order of a few days on GPU hardware or medium-sized PC clusters.

TICS exemplifies the current state of the art in machine learning. It also highlights some challenges and phenomena that permeate almost every corner of ML, and which give ML its characteristic "flavor":

High-dimensional data require dimension reduction. Many ML applications deal with images, texts, speech, video, time series, institutional data repositories which are high-dimensional. As we have seen, high-dimensional data points are by necessity thinly spread in their data space. This makes a head-on direct modeling of their distribution all but impossible, and *dimension reduction* techniques are an important topic in ML. In TICS this dimension reduction was effected for the image part of the data by the manifold learning; for the caption part several mechanisms (not reported here) were combined.

The core of ML is estimating and representing distributions. Almost all ML applications deal with data that have a random component. The degree of randomness in image caption generation is somehow intermediate; stock price modeling is infested by extreme randomness; control of car engines is almost, but not quite, deterministic. A well-trained ML system will necessarily incorporate a model of the data distribution. In many applications this model is implicit (probabilities are not always explicitly computed when the trained model is put to work). ML methods differ widely in how, concretely, these distributions are mathematically-algorithmically represented. In TICS two neural networks plus some other formalisms were combined. For the design and analysis of ML systems, probability theory and statistics are key.

Good models are smoother than reality — but how much smoother is best? We have seen in TICS that the super-curled-up distribution of images in image space was flattened to a distribution of feature vectors in a 4096-dimensional feature space. This space was low-dimensional enough to be "covered" by the available training data points, enabling some sort of interpolation for test images. If a larger set of training data had been available, a feature space of dimension higher than 4096 could have been "covered", resulting in a more detailed model. Similarly, with fewer training points, only a smaller-size model could have been estimated. In any case, the obtained models (distributions) are simpler than the real distribution. Finding the right model size (in TICS: 4096 for the image representation part) is of paramount importance for optimal outcomes. Closely related to the issue of model size is the issue of smoothing the model. The technical term is *regularization*. Figure 5**B** visualizes that the image manifold learnt by TICS "smoothes out" the training data points. We will learn much more about this later in this course.

ML is data hungry almost without limits. Given that ML is about real-world modeling and that the real world is almost unboundedly complex, and considering what I just said about smoothing, it is clear that more data will give more detailed models and hence better application performance. An important enabling factor for the recent successes of ML was the sheer availability of very large training datasets (plus, the hardware to process it). Further progress along this line can be expected.

3 The Dimensions of Machine Learning

ML as a field which perceives itself as a field is relatively young (say, 30 years); it is interdisciplinary and has historical and methodological connections to neuroscience, cognitive science, linguistics, mathematical statistics, AI, signal processing and control, and pattern recognition (the latter is traditional academic subdiscipline of computer science); it uses mathematical methods from statistics (of course), information theory, systems theory (in the sense of signal processing and control), dynamical systems (in the mathematical sense), 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 render the individual author's specific view and knowledge of the field and be partially blind to other perspectives. This is quite different from other areas in CS, say for example formal languages / theory of computation / computational complexity where a standard body of themes with a widely shared standard repertoire of methods and some standard textbooks 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 1970-1980ies, a main research motif/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.

There is no established division of ML into subfields. My personal map of ML 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. The descriptive data structures distilled from raw data are AI knowledge bases, relational databases, or other types of symbolic data collections, in each case enriched with probabilistic information. Neural networks are rarely used. A main motif 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 `https://suchanek.name/work/publications/`

`sigmodrec2013akbc.pdf` 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 case study is the PaleoDeepDive project described in Peters et al. (2014). This large-scale project aimed at making paleontological knowledge easily searchable and more realiable. Palaeontology is the science of extinct animal species. Its "raw data" are fossil bones. It is obviously difficult to reliably classify a collection of freshly excavated bones as belonging to a particular species – first, because one usually only has a few bone fragments instead of 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 belong to the same species. In the PaleoDeepDive project, the web was crawled to retrieve virtually all scientific documents relating to paleontology which were in pdf format – including a large number of such documents that had been published in pre-digital times and were just image scans. Then, 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, motor control to name a few. Human-user interpretability of the distribution representation is not important. Like Segment 2, this field is decidedly application-driven. Under the catchterm "deep learning" a subfield of this area has received a lot of attention recently.

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 punishment signals. The outcome of research often is *agent architectures*: complex, multi-module "box-and-wiring 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.

A common subdivision of ML, partly orthogonal to my private 4-section partition, is based on three fundamental abstractions 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 are images and the y are captions. The learning objective is to obtain a mechanism which, when fed with new test inputs x_{test}, returns outputs y_{test} that generalize in a meaningful way from the training sample. The underlying mathematical task is to estimate the joint distribution $P_{X,Y}$ or the conditional distributions $P_{Y \mid X}$ from the training sample (check the end of Appendix A for a brief explanation of this notation). The learnt input-output mechanism is "good" to the extent that upon input x_{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 *pattern classification* (the y are correct class labels for the input patterns x), *timeseries prediction* (the y are correct continuations of initial timeseries x), and *system identification* (this is an application type in control engineering and signal processing, where the x are input signals fed to some system and the y are measurements/observables indicative of the reactions of the system to the given input). 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 and the training objective is to learn an estimate of the distribution P_X. In our TICS demo we saw a special variant of this in the image distribution submodule, where the distribution P_{Image} of images in image space was represented in the form of a lower-dimensional manifold. Unsupervised learning can become very challenging when data points are high-dimensional and/or when the distribution has a complex shape. Simple tables or pdfs are then useless representation formats, and part of the art of unsupervised learning is to develop manageable representation formats in the first place. Unsupervised learning is closely related to *data compression* and dimension reduction: one usually desires a representation of P_X which is lower-dimensional, or simpler in some sense, than the original data format. Abstractly speaking, data compression is possible to the extent that the learning system can *discover* regularities / rules / symmetries / redundancies in the raw training data. Unsupervised learning thus needs (or leads to) some sort of "insight" into the generating mechanism for x. 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.

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 $P(A_n = a_n \,|, S_1 = s_1, \ldots, S_{n-1} = s_{n-1}; A_1 = a_1, \ldots, A_{n-1} = a_{n-1})$, 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. A celebrated 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 in interplanetary space.

The distinction between supervised and unsupervised learning is not clear-cut. 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 original 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.

4 Learning to classify patterns

The bread and butter machine learning application is *pattern classification*, which is closely connected to *pattern recognition*. In fact, before the term "machine learning" came into general use, the term "pattern recognition" stood in its place: in the 1970-ies there were university chairs, conferences, journals that had this term in their naming (some still exist). In pattern classification, the objective is to assign a correct *class label* to an input *pattern*. Here is a small choice of typical pattern recognition applications:

task name	**input patterns**	**labels**
optical character recognition	grayscale images showing a character from some alphabet	a, ..., z, 1, ..., 9 etc.
speech recognition	microphone audiosignals	words from a given vocabulary
fault diagnosis	recorded sensor data from a technical device, e.g. car or wind turbine or even a complete power grid	type and cause of malfunction
medical diagnosis	patient symptoms and laboratory data	name of possible illness causing the symptoms

The labels must come from a finite set – they are discrete data. The input patterns can be virtually any kind of data, continuous (like images) or discrete (like patient symptoms written down by a doctor) or a mixture, or even complex data structures like labelled trees.

There is a subtle difference between what is usually meant by pattern classification vs. recognition. The former is just what it says: pattern in, class label out. The latter is more involved and typically includes mechanisms to *localize* an instance of a class within a pattern, which furthermore requires to *segment* the raw input data. For instance, *visual scene analysis* is a pattern recognition tasks where the input is a camera picture of a scene, and the desired output is a list of named objects together with their localizations in the picture, and the object's shape boundaries. The TICS demo involves some visual scene analysis as a subfunctionality.

We will now leave TICS behind – too complicated – and introduce a far simpler, yet not trivial pattern classfication task, which we will use as our demo object for a while. It is an optical character recognition (OCR) task where the input patterns are 15×16 pixel grayscale images of handwritten digits 0 – 9, normalized in size and aspect ratio to fill the image panel. The data come from a benchmark dataset donated by Robert Duin, orginally retrieved from `http://ftp.ics.uci.edu/pub/ml-repos/machine-learning-databases/mfeat/mfeat-pix`, now also locally copied to `http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/DigitsBasicRoutines.zip` — this zip-file contains the original data and documentation, plus a few pieces of Matlab code for importing the data into Matlab and visualizing them. Duin and coworkers used this dataset in a study where they compared numerous ML methods for pattern classification (Kittler et al., 1998). Figure 7 gives an impression of these digit images.

Technically speaking, the elements of this sample are two-dimensional arrays of size 15×16, or more mathematically viewed, vectors x_i of length 240. The values of the vector components indicate greyscale values. They are integers ranging from 0 to 6. There are 200 instances of each digit, leading to a total sample size of 2000. These instances are ordered, that is the first 200 elements of the dataset are "0" images, the next 200 elements are "1" images, etc.

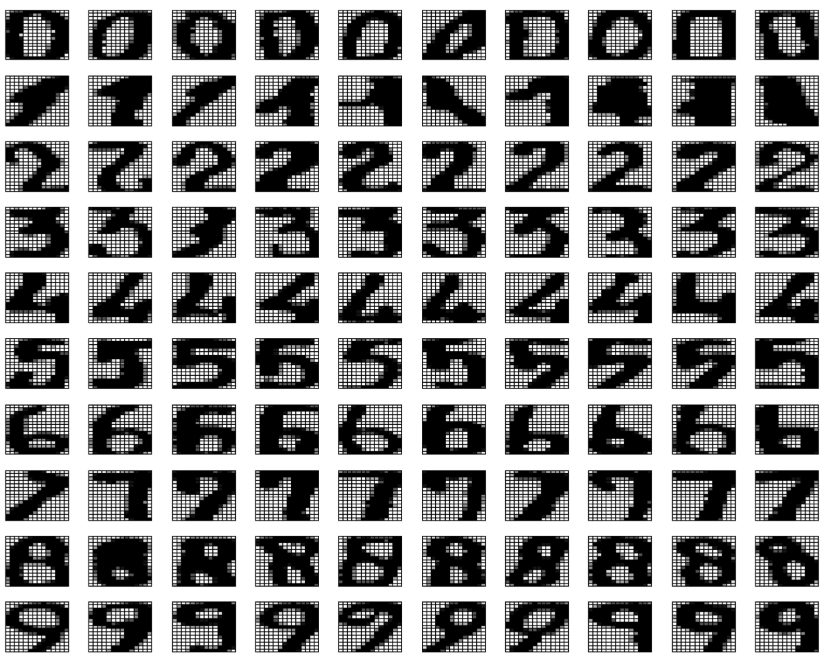


Figure 7: Some examples from the digits benchmark dataset.

In the coming weeks we will describe various ways how a *classifier* can be *learnt* (or *trained*) from these data. A classifier in our case is a computer program which upon input of a digit test image x outputs, in the simplest case, a single class label from the set $\{0, \ldots, 9\}$. More advanced classifiers would output not just their "best guess" but a *hypothesis vector* $h(x) \in (\mathbb{R}^{\geq 0})^{10}$ containing degrees of "belief" for the ten candidate labels. If one normalizes $h(x)$ to sum to 1, it can be interpreted (with due caution) as a probability vector.

In the early times of optical character recognition, such classifiers were hand-designed by engineers, who used their knowledge about the geometry of line segments in number pictures to create a cascade of optical filters in order to assemble a multi-module OCR system. This approach was prevailing until the 1980-ies (my personal impression). However, it did not work well – the messiness of real-life handwriting just defies clean reverse engineering. Today, OCR systems are not hand-designed, but learnt from example data using methods of ML.

The two main players in the ML game to train a pattern classifier are (i) *training data*, and (ii) a *learning algorithm.*

The training data consist of a collection $(x_i^{\text{train}}, y_i^{\text{train}})_{i=1,\ldots,N}$ of already classified pattern – label pairs, called a *labelled sample.* In our case, the x_i are image vectors from our digits dataset, and the y_i are the correct class labels (they are given together with the images in Duin's dataset).

The learning algorithm gets the training data as input and outputs a classifier $\mathcal{C}$. A learning algorithm transforms sample data into an algorithm!

After the classifier $\mathcal{C}$ has been trained, it is shipped to end-users who then feed it with new *test patterns* x_j^{test}, trusting that upon input of a test pattern $\mathcal{C}$ will output the correct label (or a good hypothesis vector).

Importantly, – veeeery importantly, – the test patterns are not contained in the training sample. A learning algorithm is useful to the extent that it can *generalize* well from the given training data to new test data. We will soon learn to appreciate that all the magic, and all the pain, of machine learning culminates in the question how to design learning algorithms that generalize well.

Since we don't have easy access to fresh test data formatted in the same way as the images in our dataset, we pretend that only half of the 2000 images are our training dataset. The remaining half we pretend are given to us only after training the classifier, and we can use them for testing the generalization capabilities of our classifiers.

Following the original publication Kittler et al. (1998), we thus split our dataset into two halves, using the first half (taking the first 100 instances of each digit class) for training and the second half for testing. Thus, concretely, our training data is a labelled sample

$$(x_i^{\text{train}}, y_i^{\text{train}})_{i=1,\ldots,1000}$$

and our test data is $(x_j^{\text{test}}, y_j^{\text{test}})_{j=1,\ldots,1000}$.

I said in the previous section that machine learning algorithms often are designed as all-continuous or as all-discrete devices. In pattern classification tasks, discrete-flavored classification algorithms often take the form of *decision trees.* We will not dig deeper into discrete-flavored algorithms in this lecture, but since decision trees are widely used in data mining I will give them a glimpse in passing.

A decision tree (or more specifically, a *classification tree*) is a classification algorithm whose structure can be visualized as a tree. Figure 8 gives an example for a (binary) classification tree for classifying fruit. A (binary) classification tree classifies by entering the top node with an input pattern, then checking some binary attribute of that pattern. The yes/no outcome of that attribute question leads one step down in the tree. At this next node, again a binary attribute test is carried out, etc., until a leaf node is hit, which contains the desired class label. The core challenge for training such classification trees from a labelled training sample is to identify the attribute questions that give the most reliable and compact classification tree. A big advantage of classifiers based on classification trees is that once a test case input pattern has been classified, the decision path through the tree makes it understandable to the human user *why* the classification was made – the classifier is *interpretable.* In contrast, neural network based classifiers, which we will be getting to know in this lecture, are not interpretable.

If you are interested to learn about decision trees in more detail, check out the slideset indicated in the caption of Figure 8 or read the decision tree section in the classical, super well-written textbook Mitchell (1997).

In a continuous-algorithm scenario, a classification algorithm usually is built around a *decision function.* If patterns are vectors $x \in \mathbb{R}^n$, and if there are k

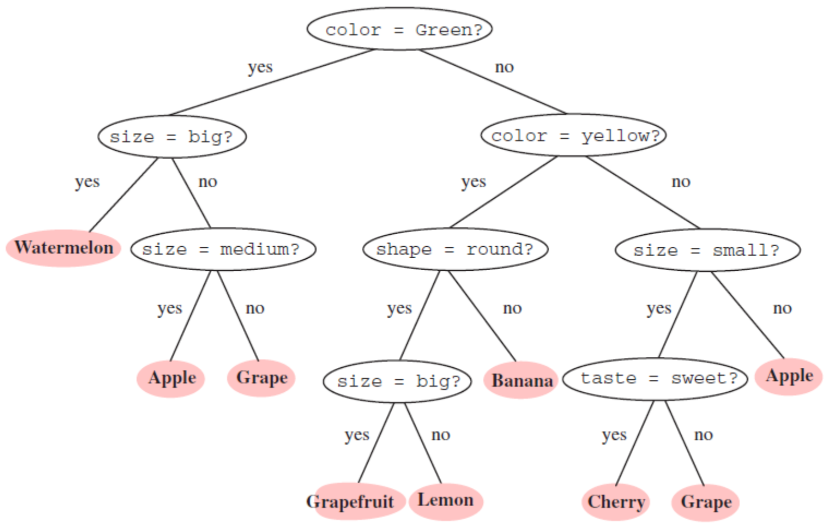


Figure 8: A binary classification tree. (Image taken from `http://www.cse.msu.edu/~cse802/DecisionTrees.pdf`, a slideset of a pattern analysis course from Anil K. Jain at Michigan State University. Local copy of slides: `http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/DecisionTrees.pdf`)

classes, then a decision function is a function

$$D : \mathbb{R}^n \to \mathbb{R}^k. \tag{1}$$

Such a decision function is the main outcome of a learning algorithm. A decision function can be a very complex algorithm, for instance a neural network. If a new test pattern $x^{\text{test}} \in \mathbb{R}^n$ comes in, it is fed to D, resulting in a hypothesis vector $h(x^{\text{test}}) = (h_1, \ldots, h_k)' \in \mathbb{R}^k$. In a final postprocessing step, this hypothesis vector is

1. either turned into a single "best guess" outcome, by returning the index i of the maximal h_i in the hypothesis vector, or

2. the hypothesis vector is transformed into a probability vector, which contains "belief" values for the classification.

There is a general standard method known by every machine learning engineer to transform any k-dimensional vector $h \in \mathbb{R}^k$ into a probability vector, called the *softmax* function:

$$\mathsf{Softmax} : \mathbb{R}^k \to \mathbb{R}^k, \quad (h_1, \ldots, h_k)' \mapsto (s_1, \ldots, s_k)',$$
$$\text{where} \quad s_i = \frac{e^{\alpha\, h_i}}{\sum_{j=1,\ldots,k} e^{\alpha\, h_j}}. \tag{2}$$

The parameter $\alpha > 0$ modulates the "sharpness" of the softmax function. Small values of α lead toward uniformly distributed softmax outcomes, large values emphasize the larger values in the argument vector h (Figure 9).

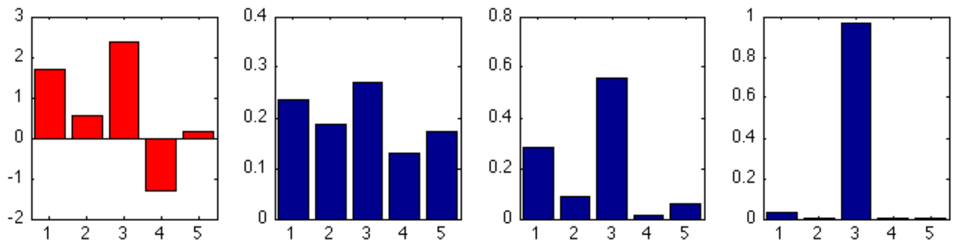


Figure 9: Softmax visualization. A 5-dimensional vector (red barchart) is submitted to the softmax with $\alpha = 0.2, 1, 5$ (from left to right, blue).

We now consider (and answer) the question which decision functions are "good" or even optimal. The natural quality criterion for a classification algorithm is that it should yield as few misclassifications on test patterns as possible. We now make this formal, and embark on our first serious exploration into the statistical foundations of machine learning.

Let us first fix some notation. Let us denote the space from which patterns can be taken as $\mathbf{P}$. Since we are focussing on continuous data, typically $\mathbf{P} \subseteq \mathbb{R}^n$ (remember: in our TICS intro example we had $\mathbf{P} = [0,1]^{1440000}$ and in our handwritten digit example $\mathbf{P} = [0,1]^{240}$). The class labels come from a finite set $\mathbf{C} = \{c_1, \ldots, c_k\}$. We furthermore introduce a random variable X for the patterns and another RV Y for the labels.

Our starting point is the true joint distribution of patterns and labels. This joint distribution is given by all the probabilities of the kind

$$P(X \in A, Y = c_i), \tag{3}$$

where A is some subvolume of $\mathbf{P}$ and $c_i \in \mathbf{C}$. The subvolumes A can be n-dimensional hypercubes within $\mathbf{P}$, but they also can be arbitrarily shaped "volume bodies", for instance balls or donuts or whatever. We write $P_{X,Y}$ for this distribution. Note that the probabilities $P(X \in A, Y = c_i)$ are numbers between 0 and 1, while the entire distribution $P_{X,Y}$ can be understood as the collection of all such probabilities. (Probability theory would give us a rigorous formal-axiomatic way to characterize the nature of this strange object, $P_{X,Y}$, but that is beyond the scope of this lecture. For us it is enough to view $P_{X,Y}$ as the totality of all the concrete probabilities of the kind (3).)

The joint distribution $P_{X,Y}$ is our "ground truth" – it is the real-world statistical distribution of pattern-label pairs of the kind we are interested in. In the digits example, it would be the distribution of pairs of handwritten digits (preprocessed into the shape shown in Figure 7), and a human-expert provided digit label. Test digit images would be randomly "drawn" from this distribution. Note that the ground truth distribution $P_{X,Y}$ cannot be precisely known – it is a real-world distribution and we have no full control of it. The best we can do is to *estimate* this distribution from training data, as we will see further below.

A decision function $D : \mathbf{P} \to \mathbb{R}^k$ separates the pattern space $\mathbf{P}$ into k disjoint *decision regions* $R_1, \ldots, R_k$ by

$$R_i = \{x \in \mathbf{P} \mid D(x) = (h_1, \ldots, h_k)' \text{ is maximal in the } i\text{th component}\}. \tag{4}$$

(I use notation v' to denote the transpose of a vector or matrix v.) A test pattern x^{test} is classified via d as class i if it falls into the decision region R_i. On the boundaries between two adjoining decision regions i and j, the hypothesis vector has (at least) two maximal entries that have identical values $h_i = h_j$.

Now we are prepared to analyze and answer our original question, namely which decision functions yield the lowest possible number 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 $P_{X \mid Y=c_i}$. 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

$$\begin{aligned}
&\sum_{i=1}^{k} P(X \in R_i, Y = c_i) = \\
&\quad = \sum_{i=1}^{k} P(X \in R_i \mid Y = c_i)\, P(Y = c_i) \\
&\quad = \sum_{i=1}^{k} P(Y = c_i) \int_{R_i} f_i(x)\, dx \\
&\quad = \sum_{i=1}^{k} \int_{R_i} P(Y = c_i)\, f_i(x)\, dx.
\end{aligned} \tag{5}$$

Note that the integral is taken over a region that possibly has curved boundaries, and the integration variable x is a vector. The boundaries between the decision regions are called *decision boundaries*. For patterns 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 7 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 (5) obviously becomes maximal if the decision regions are given by

$$R_i = \{x \in \mathbf{P} \,|\, i = \underset{j}{\operatorname{argmax}}\, P(Y = c_j)\, f_j(x)\}. \tag{6}$$

Take a look at Appendix B if you are unfamiliar with the argmax operator.

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

$$D_{\text{opt}} : \mathbf{P} \to \mathbb{R}^k,\ x \mapsto \left(\begin{array}{c} P(Y = c_1)\, f_1(x) \\ \vdots \\ P(Y = c_k)\, f_k(x) \end{array} \right). \tag{7}$$

This is not the only optimal decision function. If $\tau : \mathbb{R} \to \mathbb{R}$ is any strictly monotonic growing function, then

$$\tau \circ D_{\text{opt}} : \mathbf{P} \to \mathbb{R}^k,\ x \mapsto \left(\begin{array}{c} \tau(P(Y = c_1)\, f_1(x)) \\ \vdots \\ \tau(P(Y = c_k)\, f_k(x)) \end{array} \right)$$

leaves the decision regions unchanged and hence gives the same classification results as d_{opt}.

A learning algorithm that finds the optimal decision function (or some function approximating it) must learn (implicitly or explicitly) 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*. Figures 10 and 11 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.

Back to our digits classification example. The best results obtained by Kittler et al. (1998) had a misclassification rate of a little worse than 0.02 (that is, 2 percent) on the test set. Presumably today’s more advanced ML methods would lead to better results. You’ll later be asked to try your skills on this dataset in a homework exercise. If you reach 5% that will be very good.

5 The Curse of Dimensionality and Feature Extraction

We have briefly met this infamous curse in Section 2.3: image vectors from training data were unimaginably thinly spread in the pattern space $\mathbf{P}$ (called I in Section

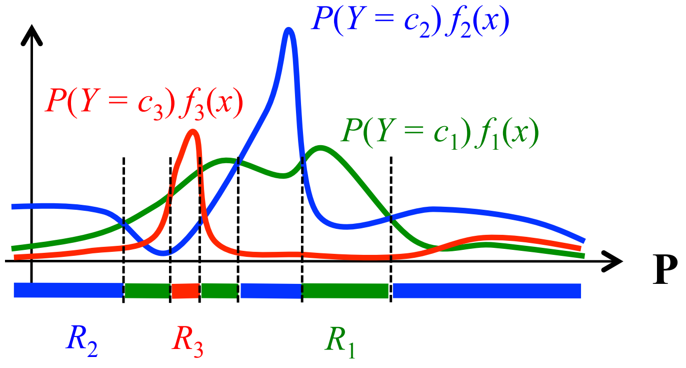


Figure 10: Optimal decision regions R_i. A case with a one-dimensional pattern space $\mathbf{P}$ and $k = 3$ classes is shown. Broken lines indicate decision boundaries. Decision regions need not be connected!

2). This “dilution of training example density” gets worse by a degree that is exponential in the pattern space dimension. Generally speaking, learning models (of whatever kind and for whatever purpose) from data usually benefits from, or even plainly requires, an estimation of the distribution of input patterns.

Estimating a distribution P_X for high-dimensional RVs X is both a very frequent (sub)task in ML, and it is intrinsically difficult to the extent that it seems almost impossible. That's the curse of dimensionality.

A common antidote to this poison is to use *feature extraction* in order to achieve a dimension reduction.

Formally, a feature is a function $f : \mathbf{P} \to \mathbb{R}$ which computes a scalar characteristic of patterns. If one bundles together m such features $f_1, \ldots, f_m$ one obtains a *feature map* $(f_1, \ldots, f_m)' =: \mathbf{f} : \mathbf{P} \to \mathbb{R}^m$ which maps patterns 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 $\mathbf{P}$ 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.

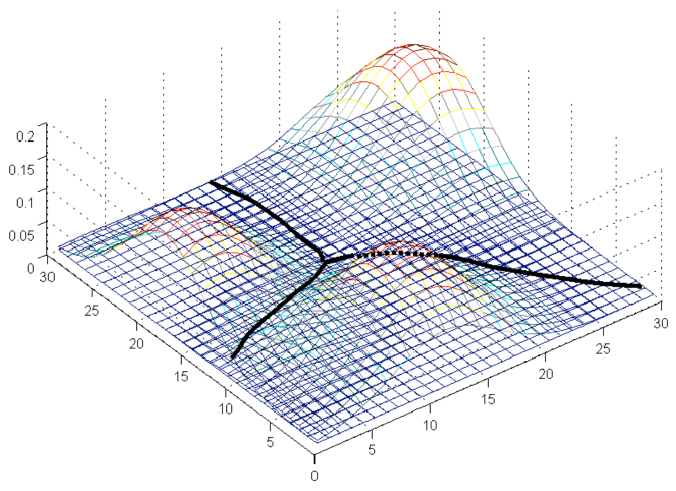


Figure 11: Optimal decision regions example for 2-dimensional patterns coming in three classes. The three "hills" are the functions $P(Y = c_k)\, f_i(x)$.

- 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 systems meets the raw data. The "has wings" feature for helicopter vs. winged aircraft classification basically amounts to actually solving the 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 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.

To sharpen our intuitions about features, let us hand-design some features of our digit images which might do good service for digit classification. We assume that the $n = 240$-dimensional image vectors x have been normalized to a real-valued pixel value range in $[0, 1]$ with $0 =$ white and $1 =$ black.

Mean brightness. $f_1(x) = 1'_n \, x \, / \, n$ (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 class-differentiating.

Radiality. An image x is assigned a value $f_2(x) = 1$ 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 $f_2(x) = 0$. 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 more informative about classes.

Prototype matching. For each of the 10 classes c_j, define a *prototype vector* π_j as the mean image vector of all 100 training samples of that class: $\pi_j = 1/100 \sum_{x \text{ is training image of class } j} x$. Then define 10 features f_3^j by match with these prototype vectors: $f_3^j(x) = \pi'_j \, x$. 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. Generally speaking, human insight on the side of the data engineer is a success factor for ML systems that can hardly be over-rated. 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 two subsections we will inspect two particularly simple and wide-spread methods for the automated construction of features, *K-means clustering* and *principal component analysis*.

5.1 Vector quantization with K-means clustering

Vector quantization is a general strategy to obtain a meaningful low-dimensional feature representation of high-dimensional data points. The idea is illustrated in Figure 13. A collection of training data points $(x_i)_{i=1,\ldots,N} \in \mathbb{R}^n$ is grouped by some geometric method into *clusters* $C_1, \ldots, C_K$ of points that in some way "belong together" in each cluster – for instance, they lie close to each other. Each cluster C_i is then represented by a *codebook* vector c_i, for instance the vector pointing to the center of gravity of the cluster.

The codebook vectors can be used in various ways to compress n-dimensional test data points x^{test} into lower-dimensional formats. The classical method, which also motivates naming the c_i "codebook" vectors, is to represent x^{test} simply by the index i of the codebook vector c_i which lies closest to x^{test}, that is, which has the minimal distance $\alpha_i = \|x^{\text{test}} - c_i\|$. This method is clearly very economical and it is widely used in data compression and data transmission.

Another way to employ the distances α_i for dimension reduction is to represent x^{test} by the 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.

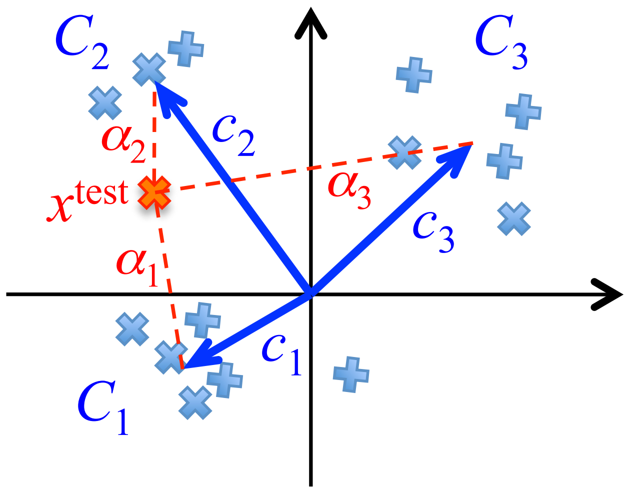


Figure 12: Vector quantization (schematic): within a training set of data points (blue crosses), a spatial grouping into clusters C_i is detected and each group becomes represented by a codebook vectors c_i (blue arrows). The figure shows three groups. A test data point (red cross) is then coded in terms of the distances α_i of that point to the codebook vectors.

There are many ways how the intuition of a cluster as a collection of points that "belong together" can be made precise. Formal specifications of "belonging together" and algorithms to group training data points into clusters can become very involved when one admits curved boundaries of clusters. Clustering methods are an ever-evolving research field. We will only consider the method which arguably is the simplest, most intuitive, fastest, and most widely used one: *K-means clustering*.

We can be brief, because K-means clustering is almost self-explaining. The rationale for defining clusters is “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:

Given: a training data set $(x_i)_{i=1,\ldots,N} \in \mathbb{R}^n$, and a number K of clusters that one maximally wishes to obtain.
Initialization: randomly assign the training points to K sets S_j $(j = 1, \ldots, K)$.
Repeat: For each set S_j, compute the mean $\mu_j = |S_j|^{-1} \sum_{x \in S_j} x$. Create new sets S'_j by putting each data point x_i into that set S'_j where $\|x_i - \mu_j\|$ 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

$$J = \sum_{j=1}^{K} \sum_{x \in S_j} \|x - \mu_j\|^2 \tag{8}$$

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 straight-line boundaries; each cluster forms a *Voronoi cell*. K-means cannot find clusters defined by curved boundaries. Figure 13 shows an example of a clustering run using K-means.

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 regard. They reveal a basic structuring of a set of patterns $\{x_i\}$ 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 $\{x_i\}$ are customer profiles, finding a good grouping into subgroups may help to design targetted marketing strategies.

5.2 Principal component analysis

Like clustering, principal component analysis (PCA) is a basic data analysis technique that has many uses besides dimension reduction. But we will here focus on this 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 high-dimensional patterns $x \in \mathbb{R}^n$ and encode it robustly in the feature vectors $y = \mathbf{f}(x)$.

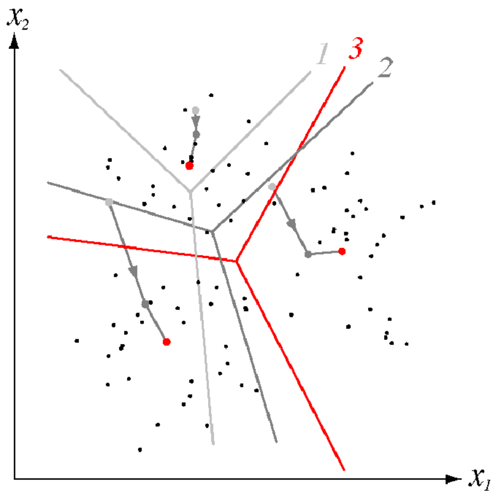


Figure 13: Running K-means with $K = 3$ on two-dimensional training points. Thick dots mark cluster means μ_j, lines mark cluster boundaries. The algorithm terminates after three iterations, whose boundaries are shown in light gray, dark gray, red. (Picture taken from Chapter 10 of the textbook Duda et al. (2001)).

But, what does it mean to "preserve information"? A clever answer is this: a feature representation $\mathbf{f}(x)$ preserves information about x to the extent that x can be *reconstructed* from $\mathbf{f}(x)$. 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 $\mathbf{f}(x)$ to x, that is we wish to achieve

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

And here comes a fact of empowerment: when $\mathbf{f}$ and $\mathbf{d}$ are confined to *linear* functions and when the similarity $x \approx \mathbf{d} \circ \mathbf{f}(x)$ 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 re-discovered 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 14). Assume the patterns are 3-dimensional vectors, and assume we are given a sample of $N = 200$ raw patterns $x_1, \ldots, x_{200}$. We will go through the steps of a PCA to reduce the dimension from $n = 3$ to $m = 2$.

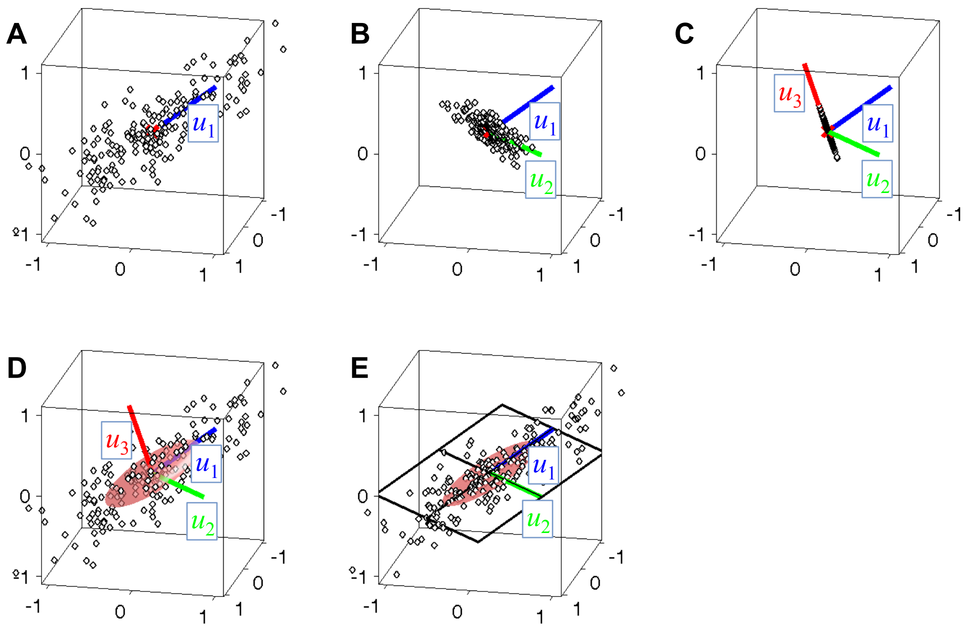


Figure 14: Visualization of PCA. **A.** Centered data points and the first principal component vector u_1 (blue). The origin of $\mathbb{R}^3$ is marked by a red cross. **B.** Projecting all points to the orthogonal subspace of u_1 and computing the second PC u_2 (green). **C.** Situation after all three 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).

The first step in PCA is to *center* the training patterns x_i, that is, subtract their mean $\mu = 1/N \sum_i x_i$ from each pattern, obtaining centered patterns $\bar{x}_i = x_i - \mu$. The centered patterns form a point cloud in $\mathbb{R}^n$ whose center of gravity is the origin (see Figure 14**A**).

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 $\bar{x}_i$ from the origin *in the direction of* u is given by the projection of $\bar{x}_i$ on u, that is, the inner product $u' \bar{x}_i$ (see Figure 15).

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

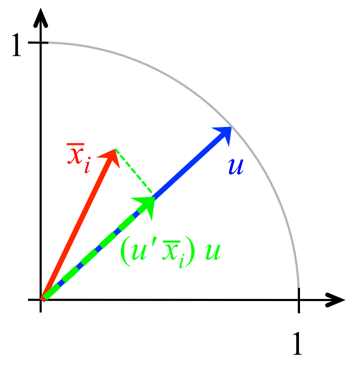


Figure 15: Projecting a point $\bar{x}_i$ on a direction vector u: the inner product $u'\,\bar{x}_i$ is the distance of $\bar{x}_i$ from the origin along the direction given by u.

direction of the largest extension of the point cloud is hence the direction vector given by

$$u_1 = \underset{u,\ \|u\|=1}{\text{argmax}}\ 1/N\ \sum_i (u'\,\bar{x}_i)^2. \tag{9}$$

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

Inspecting Figure 14**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 14**B**). That is, map pattern points $\bar{x}$ to $\bar{x}^* = \bar{x} - (u_1'\,\bar{x}) \cdot u_1$. Within this "flattened" pattern cloud, again find the direction vector of greatest variance

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

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 14**C** shows this situation, and Figure 14**D** visualizes the PCs plotted into the original data cloud.

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

$$f_k : \mathbb{R}^n \to \mathbb{R}, \quad x \mapsto u_k'\,\bar{x}, \tag{10}$$

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

$$x = \mu + \sum_{k=1,\ldots,n} f_k(x)\, u_k. \tag{11}$$

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

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m, \quad x \mapsto (f_1(x), \ldots, f_m(x))'. \tag{12}$$

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 x from their feature vectors $\mathbf{f}(x)$. In our PCA story, this decoding function is given by

$$\mathbf{d} : (f_1(x), \ldots, f_m(x))' \mapsto \mu + \sum_{k=1}^{m} f_k(x)\, u_k. \tag{13}$$

How "good" is this dimension reduction, that is, how similar are the original patterns x_i to their reconstructions $\mathbf{d} \circ \mathbf{f}(x_i)$?

If dissimilarity of two patterns $x_1, x_2 \in \mathbb{R}^n$ is measured in the square error sense by

$$\delta(x_1, x_2) := \|x_1 - x_2\|^2,$$

a full answer can be given. Let

$$\sigma_k^2 = 1/N \sum_i f_k(x_i)^2$$

denote the variance of the feature values $f_k(x_i)$ (notice that the mean of the $f_k(x_i)$, taken over all patterns, is zero, so σ_k^2 is indeed their variance). Then the mean square distance between patterns and their reconstructions is

$$1/N \sum_i \|x_i - \mathbf{d} \circ \mathbf{f}(x_i)\|^2 = \sum_{k=m+1}^{n} \sigma_k^2. \tag{14}$$

A derivation of this result is given in Appendix C.

Equation (14) gives an absolute value for dissimilarity. For applications (and theoretical insight) however the relative amount of dissimilarity compared to the mean variance of patterns is more instructive. It is given by

$$\frac{1/N \ \sum_i \|x_i - \mathbf{d} \circ \mathbf{f}(x_i)\|^2}{1/N \ \sum_i \|\bar{x}_i\|^2} = \frac{\sum_{k=m+1}^{n} \sigma_k^2}{\sum_{k=1}^{n} \sigma_k^2}. \tag{15}$$

Real-world patterns often exhibit a rapid (roughly exponential) decay of the feature variances as their index k grows. The ratio (15) 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 14 is not doing justice to the amount of compression savings that is often possible. Typical real-world 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.

5.3 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 $C = 1/N \ \sum_i \bar{x}_i \, \bar{x}_i'$, which can be easily obtained from the centered training data matrix $\bar{X} = [\bar{x}_1, \ldots, \bar{x}_N]$ 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 (graduate) online ML lecture notes, Section 4.4.1 (`http://minds.jacobs-university.de/sites/default/files/uploads/teaching/lectureNotes/LN_ML_Fall11.pdf`). Thus, the principal component vectors u_k and their associated data variances σ_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 V. Later in this course we will take a closer look at SVDs. 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 let it suffice 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 semi-definite matrix R, it returns a factorization

$$R = U \, \Sigma \, U',$$

where U is an $n \times n$ matrix whose columns are the normed orthogonal eigenvectors $u_1, \ldots, u_n$ of R and where Σ 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 U, and the variances σ_k^2, which appear as the eigenvalues of C, collected in Σ.

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 (15) to determine the "cutoff" m such that the ratio in this equation is about 0.02.

5.4 Summary of PCA based dimension reduction procedure

Data. A set $(x_i)_{i=1,\ldots,N}$ of n-dimensional pattern vectors.

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

Procedure.

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

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 to a m dimensional feature vector $\mathbf{f}(x)$, compute $\mathbf{f}(x) = U'_m\,\bar{x}$.

Usage for uncompression (decoding). In order to approximately restore x from its feature vector $\mathbf{f}(x)$, compute $x_{\text{restored}} = \mu + U_m\,\mathbf{f}(x)$. Equivalently, in a more compact notation, compute $x_{\text{restored}} = [U_m, \mu]\,[\mathbf{f}(x); 1]$, where I am using the Matlab-inspired notation $[u; v]$ for concatenating a vector u with a vector v.

5.5 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 the $N = 100$ training examples, we obtain 240 PCs u_k associated with the same number of variances σ_k^2. Only the first 99 of these variances are nonzero (because the 100 image vectors x_i span a 100-dimensional subspace in $\mathbb{R}^{240}$; after centering the $\bar{x}_i$ 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 16 shows some of these eigenvectors u_i rendered as 15×16 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.)

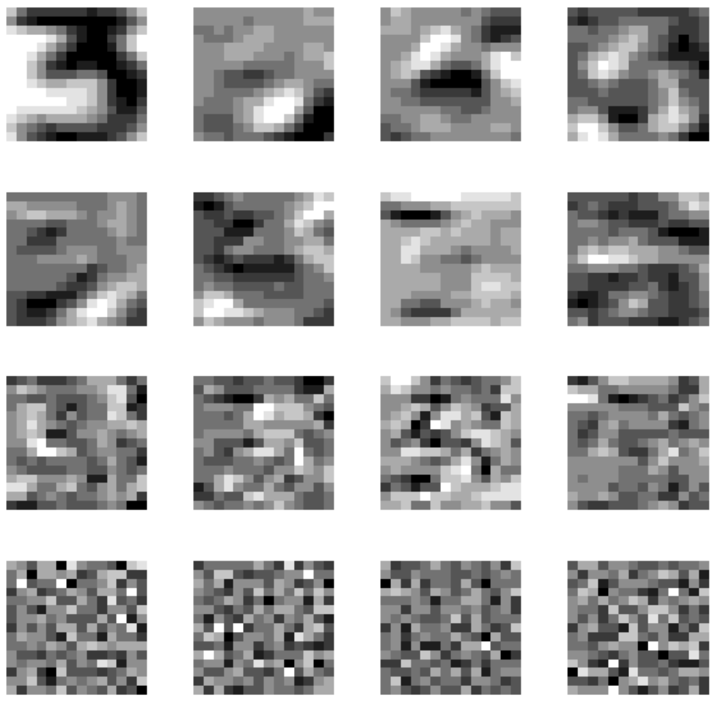


Figure 16: Visualization of a PCA computed from the "3" training images. Top left panel shows the mean μ, the next 7 panels (row-wise) show the first 7 PCs. Third row shows PCs 20–23, last row PCs 96-99. Grayscale values have been automatically scaled per panel such that they spread from pure white to pure black; they do not indicate absolute values of the components of PC vectors.

Figure 17 shows the variances $\hat{\sigma}_i^2$ of the first 99 PCs. You can see the rapid (roughly exponential) decay. Aiming for a dissimilarity ratio (Equation 15) of 0.1 gives a value of $m = 32$. Figure 18 shows the reconstructions of some "3" patterns from the first m PC features using (13).

5.6 Interim Summary: Dimension Reduction as First Stage in Classification Systems

We have seen that the core of a classification system is a decision function $d : \mathbb{R}^n \to \mathbb{R}^k$ which upon input of an n-dimensional pattern x returns a hypothesis vector

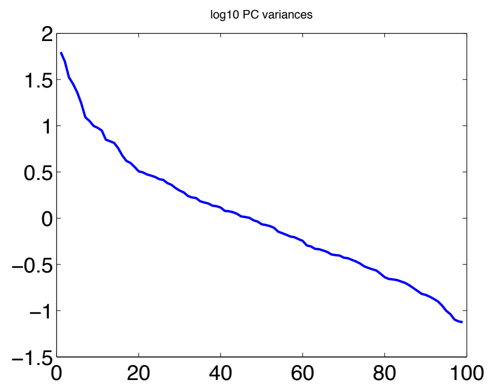


Figure 17: The (log10 of) variances of the PC features on the "3" training examples.

$D(x) = (h_1, \ldots, h_k)'$ which encodes the "degree of belief" that the classifier puts on the k classification options. The hypothesis vector can be further processed in simple but instructive ways. The most basic use of it is to identify the maximal value in it and returning its corresponding class index as "the" classification result. Often it is used, in a slightly more sophisticated way, to rank-order the classification options, by declaring which is the "most likely", the "second most likely", etc. classification outcome. Finally, one can normalize it into a probability vector which is interpreted as probabilities of the k classification alternatives. This should be done with due caution because only very few machine learning procedures yield decision functions that can be converted to probabilities in a statistically meaningful way (for experts: Roweis and Ghahramani (1999) give a unified review of such systems).

We have furthermore seen that a good decision function – one that generalizes well to test data – *must* be informed by the underlying joint distribution of patterns and their class labels (Equation 7). Learning a classification system from training data thus inevitably leads to the problem of estimating the underlying distribution from the training data sample. Learning algorithms in most cases do not *explicitly* compute a representation of this distribution – they do not usually have a substep in which a pdf or the like is generated. But they must have some kind of *implicit* encoding of that distribution.

When patterns x are high-dimensional, the curse of dimensionality strikes: it is inherently difficult to estimate a distribution from data points that are scattered unimaginably thinly in the possible data value space.

Therefore, classification systems are very often structured as two-stage processing systems. In a first stage, the high-dimensional raw input patterns $x \in \mathbb{R}^n$ are condensed into a much lower-dimensional feature vector $\mathbf{f}(x) \in \mathbb{R}^m$, where $m \ll n$.

A feature representation $\mathbf{f}(x)$ of a pattern x should preserve much information

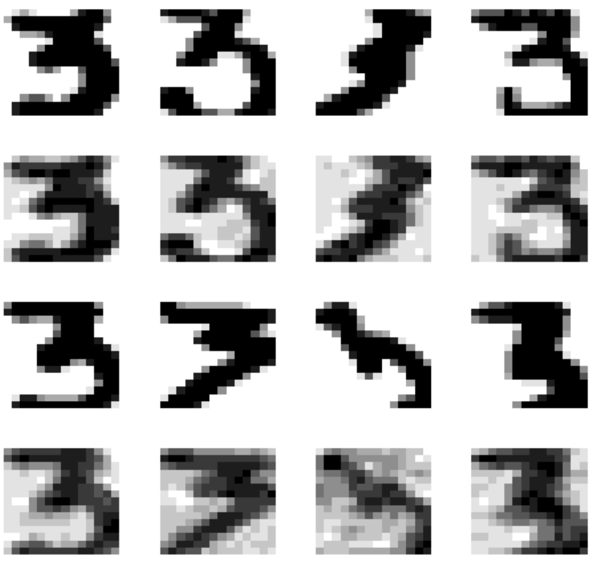


Figure 18: Reconstructions of digit “3” images from the first $m = 32$ features, corresponding to a re-constitution of 90% of the original image variance. First row: 4 original images from the training set. Second row: their reconstructions. Third row: 4 original images from the test set. Last row: their reconstruction.

contained in x which is relevant for solving the task at hand – here, classification. At the same time, task-irrelevant information in x should not be present in $\mathbf{f}(x)$ any more. A good feature mapping $\mathbf{f}$ should act as a data cleaning as well as a data compression operator.

We have discussed three standard methods for obtaining useful feature maps:

- Hand-made features are valuable for inserting human insight into the classifier. If the human experimenter has a good insight into what features of patterns are relevant for classification, and if these features can be computed from the raw patterns in an efficient way, this is a promising way to go. Many object recognition systems in robotics and medical data analysis are still partly or entirely based on hand-made features, although in the “deep learning” field there is a trend to replace them by automatically learnt features.

- Clustering methods identify K accumulation areas in the training data point cloud and represent them by K codebook vectors. A new test pattern can then be represented by a K-dimensional feature vector which contains the metric distances of the test pattern to the codebook vectors. Often the clusters can be identified with the pattern classes that one wishes to find: it is reasonable to assume that patterns belonging to the same class are lying closer together to each other than to patterns of other classes. If there are k classes and the clustering has identified $K = k$ clusters, the metric distance vector should be highly informative about the correct classification of the test pattern.

We have seen only the most basic clustering method, called K-means clustering. It has some limitations, because it can only describe shapes of clusters that are determined by linear separating hyperplanes. Detecting curved shapes of clusters needs more advanced clustering techniques.

But even with plain K-means clustering one can do better than using $K = k$ clusters. When one uses a number $K > k$ of clusters which is greater than the number of classes, the point subclouds that correspond to a class c_i are further subdivided into subclasses. This may lead to better classification results.

- Computing a PCA of the training pattern sample, keeping only a few leading PC vectors $u_1, \ldots, u_m$ and defining features to be the projections of a test pattern on these few PCs (Equation 10) gives a feature vector $\mathbf{f}(x)$ from which x can be reconstructed with a predetermined level of accuracy. PC based features thus preserve information about the original pattern in a transparent and mathematically analysable way.

I mention that one may also decide to not do any feature extraction and work with the original patterns. This can be seen as the identity "feature map" $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n, x \mapsto x$ (thus $m = n$). Working with raw patterns has become a hallmark of modern "deep learning" algorithms.

Feature maps based on clustering and PCA are obtained by *unsupervised* learning algorithms: in order to carry out K-means clustering or PCA, only the training patterns but not their labels are used. Learning features by unsupervised methods is a common strategy in ML. It has pros and cons:

- *Pro:* These methods are universally applicable to any kind of vector patterns. One does not have to "think" a lot about using them because they are task-independent.

- *Con:* Unsupervised learning methods for feature learning are not informed by the task at hand. One would think that features that are ultimately useful for classification are not necessarily the same features that are good for other purposes, e.g. for time series prediction or noise filtering. Hence one has reason to doubt that they are the best features one could get for a specific task. In fact, modern machine learning methods have found ways to learn features that are directly optimized for the task at hand. Toward the end of this course we will see how neural networks can do that.

6 Using Linear Regression to Learn a Decision Function

Reducing the dimension of raw patterns $x \in \mathbb{R}^n$ to lower-dimensional feature vectors $\mathbf{f}(x) \in \mathbb{R}^m$ is only the first step in training a classification system. We obviously also need a mechanism to transform the feature vectors $\mathbf{f}(x)$ into a hypothesis vector $h(x) \in \mathbb{R}^k$, where k is the number of classes.

6.1 Classification seen as function approximation

One way to go would be to follow the lesson of Section 4 and attempt to learn the pdf's f_i of the conditional distributions $P_{X \mid Y=c_i}$ and the class probabilities $P(Y = c_i)$, which would in principle enable us to obtain optimal decision functions (compare Equation 7). However, even finding a mathematical formalism by which one can represent such complex high-dimensional pdf's in the first place, is difficult — let alone estimating the parameters of this mathematical formalism from training data.

Fortunately there is a way to learn a decision function that is more convenient than going the full way over the underlying probability distributions. Here is an outline of the main steps of this strategy:

Problem statement. *Given:* a training data set consisting of labelled pairs $(x_i, y_i)_{i=1,\ldots,N}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{c_1, \ldots, c_k\}$. After feature extraction, these data are transformed to a dimension-reduced training data set consisting of labelled pairs $(\mathbf{f}_i, y_i)_{i=1,\ldots,N}$, where $\mathbf{f}_i \in \mathbb{R}^m$ (we write $\mathbf{f}_i$ for $\mathbf{f}(x_i)$). *Wanted:* A decision function $d : \mathbb{R}^m \rightarrow \mathbb{R}^k$ which returns "good" class hypothesis vectors. We write D for a decision function acting on raw patterns x and d for a decision function acting on the feature vectors $\mathbf{f}$. All we have in our hands to construct such a decision function is the training data $(\mathbf{f}_i, y_i)$ — we are in a situation of supervised learning.

Turn class labels into indicator vectors. Represent the symbolic class labels c_i by binary vectors z_i of size k which are zero everywhere except at position i where they are set to 1 — *class indicator vectors.* For instance, the second label c_2 in a $k = 4$ sized class label set becomes $z_2 = (0100)'$. The "labels" are now numerical vectors. The training data set becomes $(\mathbf{f}_i, z_i)_{i=1,\ldots,N}$, where $z_i \in \mathbb{R}^k$.

Turn classification learning into a function approximation problem. At this point a clever idea can be played out. We want a decision function $d : \mathbb{R}^m \rightarrow \mathbb{R}^k$. Our indicator-vector based format $(\mathbf{f}_i, z_i)$ of training data already has the right dimensions: $(\mathbf{f}_i, z_i) \in \mathbb{R}^m \times \mathbb{R}^k$. We now use this version of training data directly to search for a function $d : \mathbb{R}^m \rightarrow \mathbb{R}^k$ which is optimized to return the correct class indicator vectors z_i upon input of $\mathbf{f}_i$, that is, we try to find a function d which yields $d(\mathbf{f}_i) \approx z_i$ for all training patterns. If we can find such a function, it can be used directly as a decision function!

The desired similarity $d(\mathbf{f}_i) \approx z_i$ can be measured in various ways. The most traditional and at the same time the computationally cheapest and simplest measure of similarity is the squared distance. That is, we want to find a function d which on average makes $\|d(\mathbf{f}_i) - z_i\|^2$ as small as possible:

$$d_{\text{opt}} = \operatorname*{argmin}_{d \in \mathcal{D}} \sum_{i=1}^{N} \|d(\mathbf{f}_i) - z_i\|^2, \tag{16}$$

that is, d is the least mean square error solution that attempts to map training feature vectors to their respective class indicator vectors.

Notice that we search for the minimizing function d within a set $\mathcal{D}$ of *admissible* candidate functions. It would be impossible to search for an optimal d_{opt} within a candidate space of "all" functions $d : \mathbb{R}^m \to \mathbb{R}^k$.

6.2 Take it easy – make it linear

This is the big picture. It can (and must) be made concrete by specifying the class $\mathcal{D}$ of functions in which one searches for an optimal d_{opt}. At this stage of the course we choose the simplest nontrivial option: we search within the *linear* functions. A linear function $d : \mathbb{R}^m \to \mathbb{R}^k$ is just a $k \times m$ sized matrix W. We thus arrive at the following linear version of (16):

$$W_{\text{opt}} = \operatorname*{argmin}_{W \in \mathbb{R}^{k \times m}} \sum_{i=1}^{N} \|W\,\mathbf{f}_i - z_i\|^2. \tag{17}$$

The solution W_{opt} can be computed by a straightforward procedure known as *linear regression.* Before we delve into this theme, I describe a final little twist that is standardly inserted into the picture, namely, adding a *bias.*

Linear functions have the property that they map the zero vector to the zero vector. That is, for a feature vector $\mathbf{f}_i = \mathbf{0}$ we would necessarily obtain an all-zero class indicator vector $W\,\mathbf{f}_i = \mathbf{0}$. But this will clearly be wrong if the training data contain feature vectors $\mathbf{f}_i$ that are zero, or approximately zero. For this and other reasons one prefers to use for $\mathcal{D}$ not the basic vanilla linear functions, but a generalized kind of linear functions known as *affine* functions. An affine function $d : \mathbb{R}^m \to \mathbb{R}^k$ is a linear function plus a constant offset,

$$d(\mathbf{f}_i) = W\,\mathbf{f}_i + b,$$

where $b \in \mathbb{R}^k$ is a constant *bias* vector from (this use of the word "bias" is unrelated to the "bias" concept known in statistics). There is an elegant trick to include bias vectors in the linear regression algorithm. Using the Matlab-inspired notation $[u; v]$ for concatenating a vector u with a vector v, it is easy to see that

$$W\,\mathbf{f}_i + b = [W\,b]\,[\mathbf{f}_i; 1].$$

Thus one automatically is led into the search space of affine functions if instead of using the original feature vectors $\mathbf{f}_i$ one uses extended feature vectors $[\mathbf{f}_i; 1]$ obtained by padding the original feature vector with a 1 at its end. Another way to effect the same thing is to always use a feature mapping $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ whose last feature f_m is the constant mapping $f_m(x) \equiv 1$. This trick is standardly employed. Assuming such feature maps containing the constant-1 feature, we can use the convenient search space of linear maps, solve (17) and be done.

We now derive a solution formula for the minimization problem (17). Most textbooks start from the observation that the objective function $\sum_{i=1}^N \|W\,\mathbf{f}_i - z_i\|^2$ which one wishes to minimize is a quadratic function in the weights W and then one uses 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 better reveals the underlying geometry of the problem.

We start by observing that $\sum_{i=1}^N \|W\,\mathbf{f}_i - z_i\|^2 = \sum_{i=1}^N \sum_{l=1}^k (w_l'\,\mathbf{f}_i - z_i(l))^2$, where w_l' is the l-th row in W and $z_i(l)$ is the l-th component in z_i. Since each row w_l' from W affects only the l-th component in the inner summation, minimizing the objective in (17) amounts to minimizing individually the l summands $\sum_{i=1}^N (w_l'\,\mathbf{f}_i - z_i(l))^2$. Here the target values $z_i(l)$ are just scalars, not vectors. In order to simplify notation, we henceforth consider a scalar version of (17),

$$w_{\text{opt}} = \operatorname*{argmin}_{w \in \mathbb{R}^m} \sum_{i=1}^N (w'\,\mathbf{f}_i - z_i)^2. \tag{18}$$

It is helpful to first get an intuitive geometric impression of the situation. Figure 19 **Top** shows a case where there are $N = 10$ patterns and $m = 4$ features. In this diagram, the 10 values of each of the four features f_j are rendered as a line. Similarly the 10 targets z_i are drawn as a line connecting 10 points. The linear combination of the four “feature lines” which gives the best approximation to y is shown in orange.

The key to really understand linear regression is to realize that the N values $f_j(x_1), \ldots, f_j(x_N)$ of a feature f_j can be regarded as a vector φ_j in $\mathbb{R}^N$. Similarly, the N target values $z_1, \ldots, z_N$ can be combined into an N-dimensional vector z.

Using these N-dimensional vectors as a point of departure, geometric insight gives us a nice clue how w_{opt} should be computed. To admit a visualization, we consider for illustration purposes a case where we have only $N = 3$ patterns, and where we extract only $m = 2$ features f_1, f_2. The latter give two $N = 3$ dimensional vectors φ_1, φ_2 (Figure 19 **Bottom**). The target values z_1, z_2, z_3 are combined in a 3-dimensional vector z.

Notice that in machine learning, one should always have more data points than features, that is, $N > m$. A rule of thumb is to have at least ten times more datapoints than features. The standard situation is thus that we have fewer vectors φ_j than dimensions $= N$. The vectors φ_j thus span an m-dimensional subspace in $\mathbb{R}^N$ (shaded area in Figure 19 **Bottom**).

Notice that the objective function $\sum_{i=1}^{N} (w' \mathbf{f}_i - z_i)^2$ from (18) is equal to $\|(\sum_{j=1}^{m} w_j \varphi_j) - z\|^2$ (easy exercise – do it!), which turns (18) into

$$w_{\text{opt}} = \underset{w \in \mathbb{R}^m}{\operatorname{argmin}} \|(\sum_{j=1}^{m} w_j \varphi_j) - z\|^2, \tag{19}$$

where we used $w = (w_1, \ldots, w_m)'$ We now take another look at the graphics in 19 **Bottom**. Equation 19 tells us that we have to find the linear combination $\sum_{j=1}^{m} w_j \varphi_j$ which comes closest to y. Any linear combination $\sum_{j=1}^{m} w_j \varphi_j$ is a vector which lies in the linear subspace $\mathcal{F}$ spanned by $\varphi_1, \ldots, \varphi_m$ (shaded area in the figure). The linear combination which is closest to z apparently is the projection of z on that subspace. This is the essence of linear regression!

We may assume that the vectors $\varphi_1, \ldots, \varphi_m$ 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, which would not change $\mathcal{F}$ and the achievable linear combination vectors.

All that remains is to compute which linear combination of $\varphi_1, \ldots, \varphi_m$ is equal to the projection of z on $\mathcal{F}$. Let us call this projection z_{opt}.

The rest is just mechanical linear algebra.

Let $\Phi = (\varphi_1, \ldots, \varphi_m)$ be the $N \times m$ sized matrix whose columns are formed by the N-dimensional vectors φ_j. Then $\Phi\,\Phi'$ is a positive semi-definite matrix of size $N \times N$ with a singular value decomposition $\Phi\,\Phi' = U_N\,\Sigma_N\,U_N'$. Since the rank of Φ is $m < N$, only the first m singular values in Σ_N are nonzero. Let $U = (u_1, \ldots, u_m)$ be the $N \times m$ matrix made from the first m columns in U_N, and let Σ be the $m \times m$ diagonal matrix containing the m nonzero singular values of Σ_N on its diagonal. Then

$$\Phi\,\Phi' = U\,\Sigma\,U'. \tag{20}$$

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 19 **Bottom**).

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

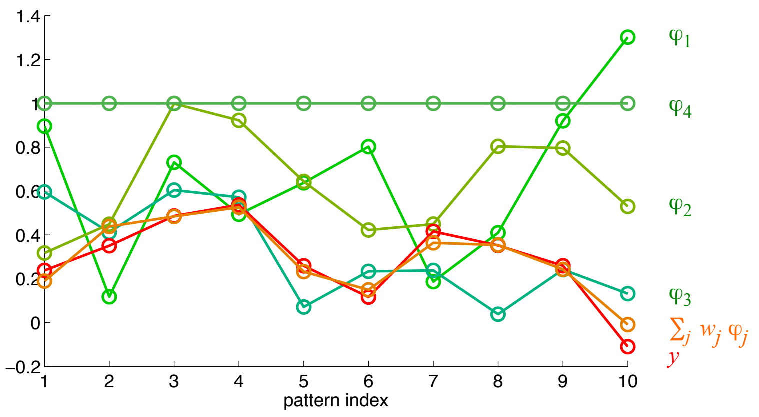
$$\varphi_j = \sum_{l=1}^{m} (u_l'\,\varphi_j)\; u_l = UU'\varphi_j, \tag{21}$$

where we introduced $U = (u_1, \ldots, u_m)$. Similarly, the projection z_{opt} of z on $\mathcal{F}$ is

$$z_{\text{opt}} = \sum_{l=1}^{m} (u_l'\,y)\; u_l = UU'z, \tag{22}$$

But also $z_{\text{opt}} = \Phi\,w_{\text{opt}}$. From (21) we get $\Phi = UU'\Phi$, which in combination with (22) turns $z_{\text{opt}} = \Phi\,w_{\text{opt}}$ into

$$UU'\,z = UU'\Phi\,w_{\text{opt}}.$$



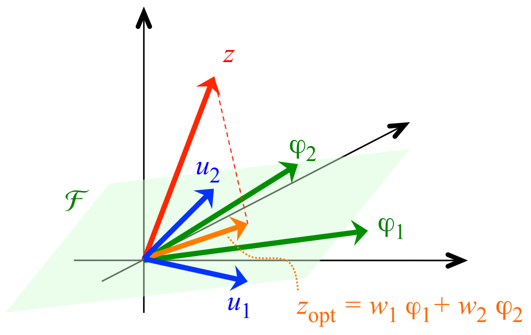


Figure 19: Two visualizations of linear regression. **Top.** This visualization shows a case where there are $N = 10$ patterns, each represented by $m = 4$ feature values $f_1(x_i), \ldots, f_4(x_i)$ (green circles). The feature f_4 is a constant-1 bias feature. The ten feature values $f_1(x_1), \ldots, f_1(x_{10})$ form a 10-dimensional vector φ_1, indicated by a connecting line. Similarly the other features can be regarded as a 10-dimensional vector each, as can be the ten target values $z_1, \ldots, z_{10}$ which give a vector z (shown in red). The linear combination $[\varphi_1\, \varphi_2\, \varphi_3\, \varphi_4]\, w_{\text{opt}}$ which gives the best approximation to z in the least mean square error sense is shown in orange. **Bottom.** The diagram shows a case where there the number of features is $m = 2$ and there are $N = 3$ sample patterns x_1, x_2, x_3 in the training set. The three values $f_1(x_1), f_1(x_2), f_1(x_3)$ of the first feature give a three dimensional vector φ_1, and the three values of the second feature give φ_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 z_1, z_2, z_3 similarly make for a vector z (red). The linear combination $z_{\text{opt}} = w_1\,\varphi_1 + w_2\,\varphi_2$ which has the smallest distance to z is given by the projection of z on this plane (shown in orange). The vectors u_1, u_2 shown in blue is a pair of orthonormal basis vectors which span the same subspace that is spanned by φ_1, φ_2.

A weight vector w_{opt} solves this equation if it solves

$$U'\,z = U'\Phi\,w_{\text{opt}}. \tag{23}$$

It remains to find a weight vector w_{opt} which satisfies (23). I claim that $w_{\text{opt}} = (\Phi'\Phi)^{-1}\,\Phi'\,z$ does the trick, that is, $U'\,z = U'\Phi\,(\Phi'\Phi)^{-1}\,\Phi'\,z$ holds.

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

$$U'\,z = U'\Phi\,(\Phi'\Phi)^{-1}\,\Phi'\,z \iff U\Sigma U'\,z = U\Sigma U'\Phi\,(\Phi'\Phi)^{-1}\,\Phi'\,z. \tag{24}$$

Replacing $U\Sigma U'$ by $\Phi\,\Phi'$ (compare Equation 20) turns the right equation in (24) into $\Phi\,\Phi'\,z = \Phi\,\Phi'\Phi\,(\Phi'\Phi)^{-1}\,\Phi'\,z$, which is obviously true. Therefore, $w_{\text{opt}} = (\Phi'\Phi)^{-1}\,\Phi'\,z$ solves our scalar optimization problem (19).

Returning to the original version of the linear regression problem (17) with vector targets $z_i \in \mathbb{R}^k$, it is easy to see that we obtain an optimal $k \times m$ weight matrix W_{opt} by

$$W'_{\text{opt}} = (\Phi'\Phi)^{-1}\,\Phi'\,Z, \tag{25}$$

where $Z = (z_i(j))_{i=1,\ldots,N;\,j=1,\ldots,m}$ is a matrix which collects all the target vectors z_i row-wise. We summarize our findings in a comprehensive algorithm:

Data. A set $(\mathbf{f}(x_i), z_i)_{i=1,\ldots,N}$ of m-dimensional feature vectors $\mathbf{f}(x_i)$ and k-dimensional target vectors z_i.

Result. A $k \times m$ dimensional weight matrix W_{opt} which solves the linear regression objective from Equation 17.

Step 1. Sort the feature vectors row-wise into an $N \times m$ matrix Φ and the targets into an $N \times k$ matrix Z.

Step 2. Compute the result by $W'_{\text{opt}} = (\Phi'\Phi)^{-1}\,\Phi'\,Z$.

Some further remarks:

- For an $a \times b$ sized matrix A, where $a \geq b$ 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 $(\Phi'\Phi)^{-1}$ may suffer from numerical instability when $\Phi'\Phi$ is close to singular; and it fails altogether when it is singular. A quick fix is to always add a small multiple of the $m \times m$ identity matrix before inverting, that is, replace (25) by

$$W'_{\text{opt}} = (\Phi'\Phi + \alpha^2\,I_{m\times m})^{-1}\,\Phi'\,Z. \tag{26}$$

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

- Without the ridge regression trick, there is another way to circumvent numerical issues that may arise from badly conditioned $\Phi'\Phi$. First compute the SVD $\Phi'\Phi = VDV'$. Then invert all nonzero diagonal values in D, obtaining a matrix $D^\dagger$. Then compute $W'_{\mathrm{opt}} = V\, D^\dagger\, V'\, \Phi'\, Z$.

  This still gives optimal weights. Computing such $V\, D^\dagger\, V'\, \Phi'$ is the basic procedure that is used in Matlab and other numerical programming packages when the built-in pseudoinverse function is called. In Matlab, you would compute linear regression weights in a numerically stable way by `W_optimal = (pinv(Phi) * Z)'`.

6.3 Interim summary: the worksuite of a pattern classification project

We have now everything in place to design complete learning algorithms for pattern classification. Here is how:

Training data. Given: a set $(x_i^{\mathrm{train}}, y_i^{\mathrm{train}})_{i=1,\ldots,N}$ of labelled patterns, where $x_i^{\mathrm{train}} \in \mathbb{R}^n$ and $y_i^{\mathrm{train}} \in \{c_1, \ldots, c_k\}$.

Fundamental assumption. The training data have been randomly sampled from a real-world joint distribution $P_{X,Y}$ of patterns and labels. Test patterns are sampled from the same distribution.

Objective. From the training data, learn a decision function $D : \mathbb{R}^n \to \mathbb{R}^k$ which upon input of test patterns x^{test} returns a decision vector $D(x^{\mathrm{test}}) = (h_1, \ldots, h_k)$. If the largest element in $(h_1, \ldots, h_k)$ is used to make a classification decision, these decisions should yield a low misclassification rate.

Stage 1: extract features. If the patterns x are high-dimensional – where "high-dimensional" can start to show its teeth already when n is greater than 10 – it is advisable to convert them into lower-dimensional feature vectors $\mathbf{f}(x) \in \mathbb{R}^m$. There is a large arsenal of techniques for feature extraction. We took a look at hand-made features, features obtained from clustering, and PCA based features. Determining the latter two kinds of features is in itself an (unsupervised) machine learning problem.

Stage 2: train a feature-based decision function $d : \mathbb{R}^m \to \mathbb{R}^k$. This is where the label information from the training data is exploited. There is again a large choice of methods for training decision functions. We only treated linear regression. A typical way to obtain the target values for linear regression in classification learning is to transform the class labels into binary class indicator vectors. Linear regression combined with this coding of classes has the advantage of being cheap to compute and easy to understand. With well-designed features it can work extremely well.

A final note. If one uses PCA for feature extraction and linear regression for computing the decision function, the overall classification algorithm $D : \mathbb{R}^n \to \mathbb{R}^k$ is just an affine linear mapping. This is because, firstly, the reduction from raw input patterns x to feature vectors $\mathbf{f}(x)$ is an affine linear map (exercise: show how the centering operation can be captured by an affine map). This is followed, secondly, by another affine linear mapping, namely the linear regression mapping. The concatenation of an affine linear map with another affine linear map yields an affine linear map again. Thus, altogether, $D : \mathbb{R}^n \to \mathbb{R}^k$ is an affine linear map, which can be encoded in a single $k \times (n+1)$ sized matrix whose last column is a bias vector (recommendable exercise: work out how this matrix is obtained from the mean pattern μ, the PC vector matrix U_m, and the regression weight matrix W). This means that the PCA + linear regression combo can only give us decision regions which have linear hyperplane boundaries. This will very likely not be the best possible classificator for digit images.

7 The bias-variance dilemma

We have seen that high dimensionality is a fundamental problem in modeling probability distributions – the curse of dimensionality. This curse is only one of two eternal fundamental problems in machine learning and statistics. The other eternal nemesis is the problem of overfitting versus underfitting data – the more educated term for the same is the *bias-variance dilemma*. I will use again the digits classification task as a demonstrator.

7.1 Training and testing errors

Let us take a closer look at what happens when we proceed as outlined in Section 6.3, using PCA projections to reduce the $n = 240$ original dimensions of the digit pics x_i to m-dimensional PC feature vectors $\mathbf{f}_i$, then use linear regression to obtain a linear decision function $d : \mathbb{R}^m \to \mathbb{R}^k$. Of each of the 200 images per class given in the public domain data set, we take the first $N = 100$ for training and the remaining 100 ones for testing.

The quality of the linear regression solution d can be quantified in several ways (using the notation from Section 6):

1. The training mean square error $\mathrm{MSE}^{\mathrm{train}} = 1/N \ \sum_{i=1}^{N} \ \|z_i^{\mathrm{train}} - d(\mathbf{f}_i^{\mathrm{train}})\|^2$.

2. The testing mean square error $\mathrm{MSE}^{\mathrm{test}} = 1/N \ \sum_{i=1}^{N} \ \|z_i^{\mathrm{test}} - d(\mathbf{f}_i^{\mathrm{test}})\|^2$.

3. The training misclassification rate

$$\varrho^{\text{train}} = 1/N \; |\{i \mid \text{maxInd}\; d(\mathbf{f}_i^{\text{train}}) \neq c_i^{\text{train}}\}|,$$

where maxInd v picks the index of the maximal element in a vector v.

4. The testing misclassification rate

$$\varrho^{\text{test}} = 1/N \; |\{i \mid \text{maxInd} d(\mathbf{f}_i^{\text{test}}) \neq c_i^{\text{test}}\}|$$

Figure 20 shows these diagnostics for all possible choices of the number $m = 1, \ldots, 240$ of PC features used. This plot visualizes one of the most important issues in (supervised) machine learning and deserves a number of comments.

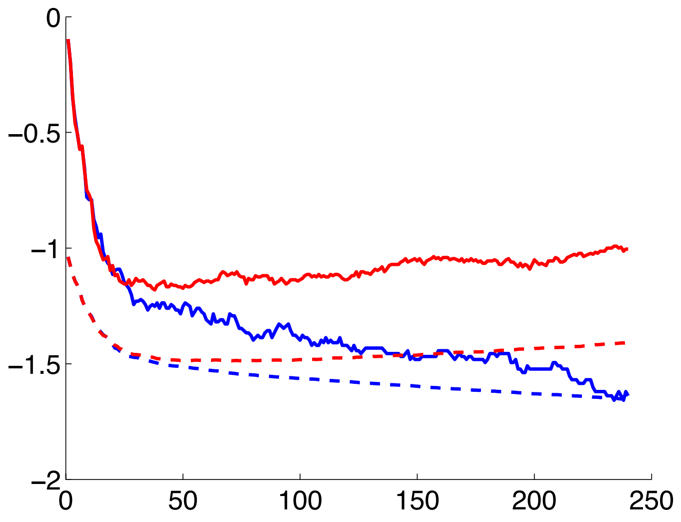


Figure 20: Dashed: Train (blue) and test (red) MSE obtained for $m = 1, \ldots, 240$ PCs. Solid: Train (blue) and test (red) misclassification rates. Y-axis is logarithmic base 10.

- As m increases (the x-axis in the plot), the number of parameters in the corresponding linear regression weight matrices W_{opt} 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 learning equation (17). This is evident from the monontonous decrease of the train MSE plot. 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 that have been trained with procedures that admit increasing 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 “regular” 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.

7.2 The menace of overfitting

This attempt of a geometric-intuitive explanation of why a too flexible model runs danger of poor test performance is admittedly hand-waving. ML research has invented a great variety of model types for supervised learning of classification and other tasks, and how exactly overfitting (mis-)functions in each of these model types will need a separate geometrical analysis. Because overfitting is such a fundamental challenge in machine learning, I illustrate its geometrical manifestations with three synthetic examples.

7.2.1 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 $y = f(x)$ 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 ν_i is drawn from a normal distribution with zero mean and standard deviation σ. Figure 21 shows a training sample $(x_i, y_i)_{i=1,\dots,11}$, where $N = 11$ x_i are chosen equidistantly.

We now want to solve the task of learning a good approximation for f from the training data (x_i, y_i) by applying polynomial curve fitting, an elementary technique

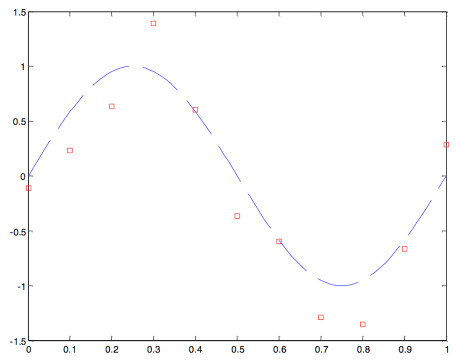


Figure 21: An example of training data (red squares) obtained from a noisy observation of an underlying "correct" function $sin(2\,\pi\,x)$ (dashed blue line).

you might be surprised to meet here as a case of machine learning. Consider an m-th order polynomial

$$p(x) = w_0 + w_1 x + \cdots + w_m x^m. \tag{27}$$

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

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

At this moment we don't bother how this task is solved computationally but 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 monomial coefficients w_j that minimize this MSE. Figure 22 shows the polynomials found in this way for $M = 1, 3, 10$.

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

- If we increase the order m, we get increasingly lower $\text{MSE}^{\text{train}}$.
- 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).

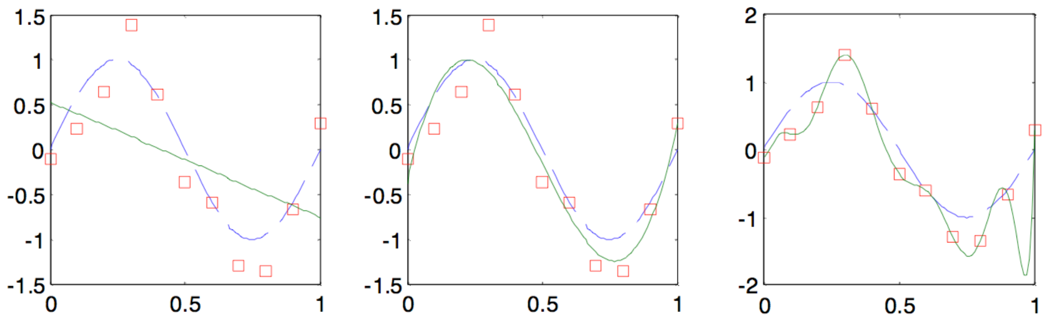


Figure 22: Fitting polynomials (green lines) for polynomial orders 1, 3, 10 (from left to right).

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.

7.2.2 Example 2: pdf estimation

Let us consider the task of estimating a 2-dimensional pdf over the unit square from 6 training data points $\{x_i\}_{i=1,\ldots,6}$, 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 23 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). 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).

7.2.3 Example 3: learning a decision boundary

Figure 24 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, 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.

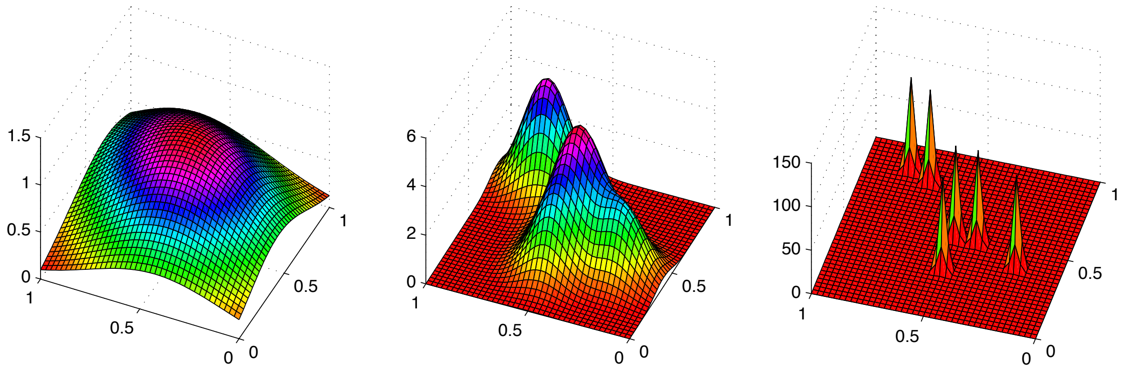


Figure 23: Estimating a pdf from 6 data points. Model flexibility grows from left to right. Note the different scalings of the z-axis: the integral of the pdf is 1.

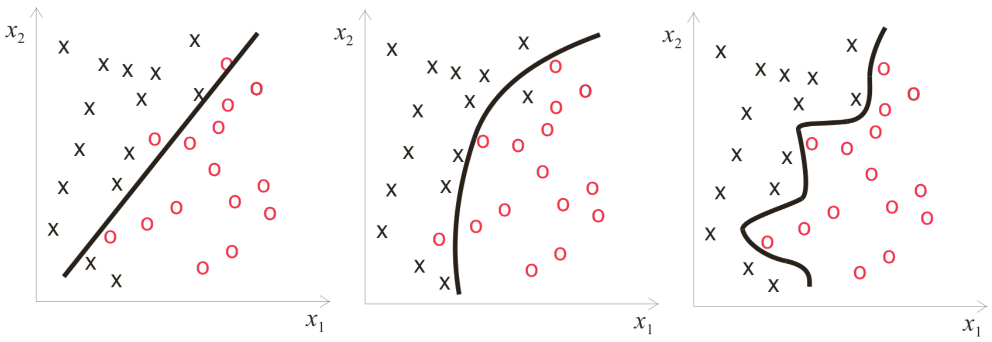


Figure 24: Learning a decision boundary for a 2-class classification task of 2-dimensional patterns (marked by black "x" and red "o").

7.2.4 Interim summary

The three examples stem from different learning tasks (function approximation, pdf learning, classification learning), and correspondingly the 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 adapt the model to each individual training point. This leads to small, even zero, training error; but it is 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 strikes the best possible balance.

Properly speaking, flexibility is not a characteristic of a single model, but of a 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 it in this lecture, 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 7.4 and 7.6 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-lu-te-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.

7.3 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 $(x_i, y_i)_{i=1,\ldots,N}$, which is drawn from a joint distribution $P_{X,Y}$ of two random variables X and Y. So far, we have focussed on pattern classification tasks, where the x_i were "patterns" and the y_i were class labels. But supervised learning tasks occur in many other variants, too. For instance, in *time series prediction* tasks, the x_i may be observations of a time series up to the current time, and the y_i would be the continuations of that time series into the future. Or, in general *function approximation* tasks, the x_i are arguments to an (unknown, random) function and the y_i are the results returned by that function. An example of a function approximation task which is both important and difficult is text translation: the x_i would be sentences in a source language, say English, and the y_i would be translations of these sentences in a target language, say French. In this subsection we will take an abstract view and just consider any kind of supervised learning based on a training sample $(x_i, y_i)_{i=1,\ldots,N}$. 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 $(\mathbf{f}_i, z_i)$ with data value spaces $\mathbb{R}^m, \mathbb{R}^k$ respectively. In English-French sentence translation tasks, the data value spaces of the argument variables X 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 $D : \mathcal{E}_X \to \mathcal{E}_Y$. We called this function a *decision function* earlier in these lecture notes, and indeed that is the term which is used in abstract statistical learning theory.

The decision function D obtained by a learning algorithm should be optimized toward some objective. This objective arises from the specific goals that one pursues. For instance, in pattern classification one may wish to minimize the number of misclassifications; or in function approximation one may wish to minimize the square error; or in credit risk prediction a bank may wish to minimize the financial loss resulting from failed loans. Abstracting away from such particulars, one introduces the concept of a *loss function.* A loss function is a function

$$L : \mathcal{E}_Y \times \mathcal{E}_Y \to \mathbb{R}^{\geq 0}. \tag{28}$$

The idea is that a loss function measures the "cost" of a mismatch between the target values y and the values $D(x)$ returned by a decision function. Higher cost means lower quality of D. 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

  $$L(D(x), y) = \begin{cases} 0, & \text{if } D(x) = y \\ 1, & \text{if } D(x) \neq y \end{cases} \tag{29}$$

  This loss was used in our derivation of optimial decision regions, compare Equation 7.

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

  $$L(D(x), y) = \|D(x) - y\|^2. \tag{30}$$

  This loss is often just called "quadratic loss". We used it as a basis for deriving the linear regression based classifier in Section 6.

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

$$R(D) = E[L(D(X), Y)] \tag{31}$$

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 D would be given by $R(D) = \int_{\mathcal{E}_X \times \mathcal{E}_Y} L((D(x), y)\, f(x, y)\, d(x, y)$. However, the true distribution is unknown. The mission to find a decision function D which minimizes (31) 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 $(x_i, y_i)_{i=1,\ldots,N}$.

A natural escape from this impasse is to tune a learning algorithm such that instead of attempting to minimize the risk (31) it tries to minimize the *empirical risk*

$$R^{\text{emp}}(d) = 1/N \sum_{i=1}^{N} L(d(x_i), y_i), \tag{32}$$

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

$$D_{\text{opt}} = \underset{D \in \mathcal{D}}{\operatorname{argmin}} \; 1/N \sum_{i=1}^{N} L(D(x_i), y_i). \tag{33}$$

We already saw one example: linear regression is the learning algorithm which, for numerical X, Y, among the candidate set $\mathcal{D}$ of all affine linear decision functions finds the one that minimizes the quadratic empirical loss.

The empirical risk is often – especially in numerical function approximation tasks – also referred to as *training error*, and the risk as (expected) *testing 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 which combine a low empirical risk with a high risk. This is the the ugly face of overfitting which I highlighted in the previous Subsection 7.2. 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 theories, sometimes referred to as *statistical learning theory* (or better, theor*ies*), are mathematically demanding and 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 parts 2 and 3 of lecture notes of the "Principles of Statistical Modeling" course at `minds.jacobs-university.de/teaching/PSMFall2016`. You will find that the definitions of loss and risk given there are more involved than what I presented above, but the somewhat simplified version that I gave here is often used in introductory textbooks to machine learning.

7.4 Tuning learning flexibility through model class size

Back to business. From what we saw in Sections 7.1 and 7.2 it appears that in order to achieve a low testing error (that is, a low risk) it is crucial to determine an appropriate modelling flexibility. We introduced this theme only informally in that section. Let us re-consider two examples to get a clearer picture:

- In the digits classification example (Section 7.1, Figure 20), 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 $\mathcal{D}$ of candidate decision functions within which the empirical risk (33) is minimized. Specifically, using a fixed m meant that the optimial decision function D was selected from the set $\mathcal{D}_m$ which contains all decision functions which first extract m principal component features, then combine those feature values linearly into the hypothesis vectors. It is clear that $\mathcal{D}_{m-1}$ is contained in $\mathcal{D}_m$, 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 combination weight for the m-th feature is zero.

- In the polynomial curve fitting example from Section 7.2.1, the model parameters were the monomial coefficients $w_0, \ldots, w_m$ (compare Equation 27). After fixing the polynomial order m, optimal decision functions $p(x)$ were selected from the set $\mathcal{D}_m = \{p : \mathbb{R} \to \mathbb{R} \,|\, p(x) = \sum_{j=0}^{m} w_j\, x^j\}$. Again it is clear that $\mathcal{D}_{m-1}$ is contained in $\mathcal{D}_m$.

A note on terminology: I use the words "decision function" and "model" as synonyms, meaning the (classification) algorithm D which results from a learning procedure. The word "decision function" is standard in statistics, the word "model" is more popular in machine learning. I also remark that in statistics, the term "statistical model" means something entirely different which we will not discuss here – just be aware of that in case you read a textbook of statistics.

Generalizing from our 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 $\mathcal{D}_1 \subset \mathcal{D}_2 \subset \ldots \subset \mathcal{D}_L$ 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 $\mathcal{D}_{m+1}$ can achieve an empirical risk that is at least as low as the optimal model in class $\mathcal{D}_m$, but usually has a properly lower risk – and is closer to overfitting.

There are many ways how one can set up a sequence of learning algorithms which 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 for “higher” classes.

From now on we assume that a class inclusion sequence $\mathcal{D}_1 \subset \ldots \subset \mathcal{D}_L$ is given. We furthermore assume we have a loss function L and are in possession of a learning algorithm which for every class $\mathcal{D}_m$ can solve the optimization problem of minimizing the empirical risk

$$D_{\text{opt}\,m} = \underset{D \in \mathcal{D}_m}{\text{argmin}} \;\; 1/N \sum_{i=1}^{N} L(D(x_i), y_i). \tag{34}$$

So... how can we find the best model class m_{opt} which gives us the best *risk* – note: *not* the best *empirical* risk? Or stated in more basic terms, which model class will give us the smallest expected test error? Expressed formally, how can we find

$$m_{\text{opt}} = \underset{m}{\text{argmin}} \;\; R(D_{\text{opt}\,m})? \tag{35}$$

7.5 Finding the right modeling flexibility by cross-validation

Statistical learning theory has come up with a few analytical methods to approximately solve (35). But these methods are based on additional assumptions which are neither easy to verify nor often granted. By far the most widely used method to determine an (almost) optimal model class 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 curves in Figure 20.

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

The idea of cross-validation is to artificially split the training data set $S = (x_i, y_i)_{i=1,\ldots,N}$ 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. Data in T is used to train optimal models $D_{\text{opt}\,m}$ from $\mathcal{D}_1, \ldots, \mathcal{D}_L$, in turn. The test generalization performance on “new” data is then tested on the validation set, for each $m = 1, \ldots, L$. It is determined which model $D_{\text{opt}\,m}$ performs best on the validation data. Its class index m is then taken to be the sought solution m_{opt} to (35). After this screening of model classes for the best test performance, a model within the found optimal class is then finally trained on the original complete training data set S.

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

A clever way to answer this question is to split S into many subsets S_j of equal size ($j = 1, ..., K$). Then carry out K complete screening runs via cross validation, where in the j-th run the subset S_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 m_{opt}. This is called *K-fold cross-validation.* Here is the procedure in detail:

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

Setup. Procure a model class inclusion sequence $\mathcal{D}_1 \subset \ldots \subset \mathcal{D}_L$ ranging from highly constrained models $\mathcal{D}_1$ which will likely underfit to very flexible, large models $\mathcal{D}_L$ which will likely overfit.

Result. A model $D_{m_{\text{opt}}}$ that was trained within the presumably optimal class $\mathcal{D}_{m_{\text{opt}}}$ – neither underfitting nor overfitting.

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

Step 2. Repeat for $j = 1, \ldots, K$:

Step 2.1 Designate S_j as validation set V_j and the union of the other $S_{j'}$ as training set T_j.

Step 2.2 Repeat for $m = 1, \ldots, L$:

Step 2.2.1 Within the class $\mathcal{D}_m$, compute the optimal model

$$D_{\text{opt}\, m\, j} = \underset{D \in \mathcal{D}_m}{\text{argmin}}\; 1/|T_j| \sum_{(x_i, y_i) \in T_j} L(D(x_i), y_i).$$

Step 2.2.2 Test $D_{\text{opt}\, m\, j}$ on the current validation set V_j by computing the validation risk

$$R^{\text{val}}_{m\, j} = 1/N' \sum_{(x_i, y_i) \in V_j} L(D_{\text{opt}\, m\, j}(x_i), y_i).$$

Step 3. For $m = 1, \ldots, L$ average the K validation risks obtained from the "folds" carried out for this class, obtaining

$$R^{\text{val}}_m = 1/K \sum_{j=1,\ldots,K} R^{\text{val}}_{m\, j}.$$

Step 4. Find the optimal class by looking for that m which minimizes the averaged validation risk:

$$m_{\text{opt}} = \underset{m}{\text{argmin}}\; R^{\text{val}}_m.$$

Step 5. Compute $D_{m_{\text{opt}}}$ using the complete original training data set:

$$D_{m_{\text{opt}}} = \underset{D \in \mathcal{D}_{m_{\text{opt}}}}{\text{argmin}}\; 1/N \sum_{i=1,\ldots,N} L(D(x_i), y_i).$$

This procedure contains two nested loops and looks expensive. For economy, one starts with the low-end class and expands it stepwise, assessing the generalization quality through cross-validation for each class m, until the validation risk starts to rise. The class 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 S_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 class size 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.

7.6 Using regularization for tuning modeling flexibility

There are several methods to endow a learning algorithm with variable degrees of flexibility. One such method was described in Section 7.4: in order to increase the flexibility of a learning algorithm, increase the number of adjustable parameters — said more simply, bigger models are more flexible. In this section I present a quite different approach, which however is equally simple, practical, and in widespread use. It is called *model regularization.*

When one uses regularization to vary the modeling flexibility, one does not vary the model class size at all. Instead, one varies the minimization task (33).

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 22, 23, 24).

When one uses model regularization, one fixes a single model structure and size with a fixed number of trainable parameters. Structure and size of the considered model should be rich and large enough to be able to overfit the available training data. 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.

This baseline model type is characterized by a set of trainable parameters. In our example of digit classification through linear regression from raw images these traineable 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 θ. This is a vector that has as many components as there are trainable parameters in the chosen baseline model. We assume that we have M tuneable parameters, that is $\theta \in \mathbb{R}^M$.

Such a high-flexibility baseline model type would inevitably lead to overfitting when an "optimal" model would be learnt using the basic learning equation (33) which I repeat here for convenience:

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

In order to downregulate 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 θ that code models with a high degree of geometrical "wiggliness".

The design of a useful penality term is up to the data engineer's ingenuity. A good penalty term should, of course, assign high penalty values to parameter vectors θ 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 7.2.1 one might consider as a baseline all 10th order polynomials but penalize the "oscillations" seen in the right panel of Figure 22, 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,

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

   Investing a little calculus exercise, it can be seen that this integral can be resolved into a quadratic form $R(\theta) = \theta'\, C\, \theta$ where C is an 11×11 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 also called the *L_2-norm regularizer* because it measures the (squared) L_2-norm of the parameter vector θ.

Once one has decided on a regularizer, one simply adds it to the empirical risk function. This means to replace (33) by

$$D_{\text{opt}} = \underset{D \in \mathcal{D}}{\operatorname{argmin}} \left(1/N \sum_{i=1}^{N} L(D(x_i), y_i) + \alpha^2\, R(\theta_D) \right). \tag{36}$$

where θ_D is the collection of parameter values in the candidate model D.

Computing a solution to this minimization task means to find a set of parameters which simultaneously minimizes the original risk and the penalty term. The factor α^2 in (36) controls how strongly one wishes the regularization to impact on the solution. Increasing α means decreasing the model capacity. For $\alpha^2 = 0$ one returns to the original un-regularized 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 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.

Returning once again to our eternal overfitting diagram in Figure 20, the increase in model flexibility (going from left to right on the x-axis) is realized by a decrease in the tradeoff parameter α^2. On the left side of the diagram one finds the training / testing errors obtained from regularized learning runs with large α^2, on the right side the ones with small α^2.

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 baseline model type with a very large (unregularized) flexibility, which typically means to select a big model with many parameters (maybe hundreds of thousands). Then one starts a sequence of training trials where in the first trials one uses a strong regularization (large α^2), which is subsequently decreased in each training trial. In each trial one uses (K-fold or simple) cross-validation to assess the generalization qualities on a validation dataset. The validation risk will initially shrink from trial to trial, then start to grow again. At that turning point one has found the best α^2.

7.7 Ridge regression

Let us briefly take a fresh look at linear regression, now in the light of general supervised learning and regularization. Recall from Section 6 that the learning task solved by linear regression is to find

$$w_{\text{opt}} = \operatorname*{argmin}_{w \in \mathbb{R}^n} \sum_{i=1}^{N} (w'\, x_i - y_i)^2, \tag{37}$$

where $(x_i, y_i)_{i=1,\ldots,N}$ is a set of training data with $x_i \in \mathbb{R}^n, y_i \in \mathbb{R}$. Like any other supervised learning algorithm, linear regression may lead to overfitting solutions w_{opt}. It is *always* advisable to control the flexibility of linear regression with an L_2 norm regularizer, that is, instead of solving (37) go for

$$w_{\text{opt}} = \operatorname*{argmin}_{w \in \mathbb{R}^n} \left(\left(\sum_{i=1}^{N} (w'\, x_i - y_i)^2 \right) + \alpha^2 \, \|w\|^2 \right) \tag{38}$$

and find the best regularization coefficient α^2 by cross-validation. The optimization problem (38) admits a closed-form solution, namely the *ridge regression* formula that we have already met in Equation 26. Rewriting it a little to make it match with the current general scenario, here it is again:

$$w'_{\text{opt}} = (X\,X' + \alpha^2\, I_{n \times n})^{-1}\, X\, Y, \tag{39}$$

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

In Section 6 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 (37) 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.

7.8 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 $(x_i, y_i)_{i=1,\ldots,N}$ 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 $D : \mathbb{R}^n \to \mathbb{R}$ which has a low quadratic risk

$$R(D) = E_{X,Y}[(D(X) - Y)^2], \tag{40}$$

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 $(x_i, y_i)_{i=1,\ldots,N}$, estimates a model $\hat{D}$. 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 $(x_i, y_i)_{i=1,\ldots,N}$ as input and returns a model $\hat{D}$. Importantly, if we would run $\mathcal{A}$ repeatedly, but using freshly sampled training data $(x_i, y_i)_{i=1,\ldots,N}$ in each run, then the returned models $\hat{D}$ would be varying from trial to trial – because the input samples $(x_i, y_i)_{i=1,\ldots,N}$ are different in different trials. Applying these varying $\hat{D}$ to some fixed $x \in \mathbb{R}^n$, the resulting values $\hat{D}(x)$ would show a random behavior too. The distribution of these variable values $\hat{D}(x)$ 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), because the random variation of the models $\hat{D}$ is determined by the variation of the training samples, that is, by $P_{X,Y}$. A good statistician could give us a formal derivation of the distribution of $\hat{D}(x)$ 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 the distribution of $\hat{D}(x)$ has an expectation which is ultimately determined by $P_{X,Y}$, so we are justified to write $E[\hat{D}(x)]$.

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

If you feel that you have made friends with this $E[\hat{D}(x)]$ 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 $\mathcal{D}$), the quadratic risk (40) is minimized by the function

$$\Delta(x) = E_{Y|X=x}[Y|X = x], \tag{41}$$

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, Δ remains unknown because the underlying true distribution $P_{X,Y}$ cannot be exactly known.

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

$$E[(\hat{D}(x) - \Delta(x))^2].$$

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

$$\begin{aligned}(\hat{D}(x) - \Delta(x))^2 &= (\hat{D}(x) - E[\hat{D}(x)] + E[\hat{D}(x)] - \Delta(x))^2 \\ &= (\hat{D}(x) - E[\hat{D}(x)])^2 + (E[\hat{D}(x)] - \Delta(x))^2 \\ &\quad +2\,(\hat{D}(x) - E[\hat{D}(x)])\,(E[\hat{D}(x)] - \Delta(x)). \end{aligned} \tag{42}$$

Now observe that

$$E\left[(\hat{D}(x) - E[\hat{D}(x)])\,(E_{X,Y}[\hat{D}(x)] - \Delta(x))\right] = 0, \tag{43}$$

because the second factor $(E[\hat{D}(x)] - \Delta(x))$ is a constant, hence

$$\begin{aligned}&E\left[(\hat{D}(x) - E[\hat{D}(x)])\,(E_{X,Y}[\hat{D}(x)] - \Delta(x))\right] = \\ &= E\left[\hat{D}(x) - E_{X,Y}[\hat{D}(x)]\right]\,(E_{X,Y}[\hat{D}(x)] - \Delta(x)),\end{aligned}$$

and

$$\begin{aligned}E\left[\hat{D}(x) - E[\hat{D}(x)]\right] &= E[\hat{D}(x)] - E[E[\hat{D}(x)]] \\ &= E[\hat{D}(x)] - E[\hat{D}(x)] \\ &= 0.\end{aligned}$$

Inserting (43) into (42) and taking the expectation on both sides (42) of finally gives us

$$E[(\hat{D}(x) - \Delta(x))^2] = \\ = \underbrace{\left(E[\hat{D}(x)] - \Delta(x)\right)^2}_{\text{(squared) bias}} + \underbrace{E\left[\left(\hat{D}(x) - E[\hat{D}(x)]\right)^2\right]}_{\text{variance}}. \tag{44}$$

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

When the model flexibility is too low (underfitting), the bias term dominates the expected modeling error; when the flexibility is too high (overfitting), the variance term is the main source of mismatch. This is why the underfitting versus overfitting challenge is also called the bias-variance tradeoff (or dilemma).

8 A first view of neural networks: the Multilayer Perceptron

Artificial neural networks (ANNs) are employed in two major scientific domains:

- In computational neuroscience, ANNs are investigated 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 training 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 common 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 the decades since the two branches diverged further and further from each other, despite repeated and persistent attempts to re-unite them. Today most ANN research in machine learning has lost all connections to 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 (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 an edge-labelled graph whose nodes are the neurons and whose vertices are the links, labelled with their weights. The structure of an ANN with M units can thus be represented by its $M \times M$ sized *connection weight matrix* W, where the entry $W(i,j) = w_{ij}$ 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 W therefore determine the network's connection topology.
- At any given moment while an ANN is performing a computation, the units each have a real-valued *activation*. In an ANN with M units, all of these activations together are combined into an *state vector* $x \in \mathbb{R}^M$.
- The state vector is computed or updated according to a state update function f, which (re-)computes the network state $x(t)$ at computation timestep t based on external input $u(t)$ and/or its previous state $x(t-1)$. The state update depends on the connection weights, so we may write $x(t+1) = f(x(t), u(t), W)$.
- The state update function is almost always *local*: the activation $x_i(t+1)$ of unit i depends only on the activations $x_j(t)$ of units j that "feed into" i, that is where $w_{ij} \neq 0$.
- The global algorithmical functionality of an ANN results from the combined local interactions between 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.

- The global functionality of an ANN is determined by the connection weights W. Absolutely drastic changes in functionality can be achieved by changing the nonzero weights in W. For instance, an ANN with a given topology (that is, fixed zeros in W) can serve as a recognizer of handwritten digits with some W_α, and can serve as a recognizer of facial expressions with another W_β!
- The hallmark of ANNs is that the global functionality is *learnt* from training data. In typical learning procedures, the weight matrix W is iteratively changed, starting from a random initial W_0. In a sequence of learning steps, W_0 is incrementally adjusted with small changes in each step, leading to a sequence of weights $W_1, W_2, \ldots$, up to some learning time point T when W_T is taken to be good enough.

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*, or often also – for historical reasons – *multilayer perceptrons* (MLPs).

MLPs are used for the supervised learning of vectorial input-output tasks. In such tasks the training sample is of the kind $(u_i, y_i)_{i=1,\ldots,N}$, where $u \in \mathbb{R}^n, y \in \mathbb{R}^k$, drawn from a joint distribution $P_{U,Y}$. (Note that here the notation departs from the one used in earlier sections: I now use u instead of x to denote input patterns, in order to avoid confusion with the network states x.) The MLP is trained to produce outputs $y \in \mathbb{R}^k$ upon inputs $u \in \mathbb{R}^n$ in a way that this input-output mapping is similar to the relationships $u_i \mapsto y_i$ found in the training data. Similarity is measured by a suitable loss function.

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 MLPs and their variations are the most widely used workhorse tool 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 u can be sent to the input units, 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 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 W, one could also write $\mathcal{N}_W : \mathbb{R}^n \to \mathbb{R}^k$ 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 common choice for L is the quadratic loss $L(\mathcal{N}(u), y) = \|\mathcal{N}(u) - y\|^2$, but other loss functions can also be used. Given the loss function, the goal of training an MLP is to find a weight matrix W_{opt} which minimizes the empirical loss, that is

$$W_{\text{opt}} = \underset{W}{\operatorname{argmin}} \sum_{i=1}^{N} L(\mathcal{N}_W(u_i), y_i). \tag{45}$$

"Function approximation" sounds dire 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, stock market time series. Outputs y are indicator 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.

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.

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 system 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 settings for the desired plant behavior, plus optionally observation data from the current plant performance. The outputs are the control inputs which are sent to the plant.

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 personal investment for any engineer or data analyst.

8.1 MLP structure

Figure 25 gives a schematic of the architecture of an 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 25). 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* $x_i^m \in \mathbb{R}$.

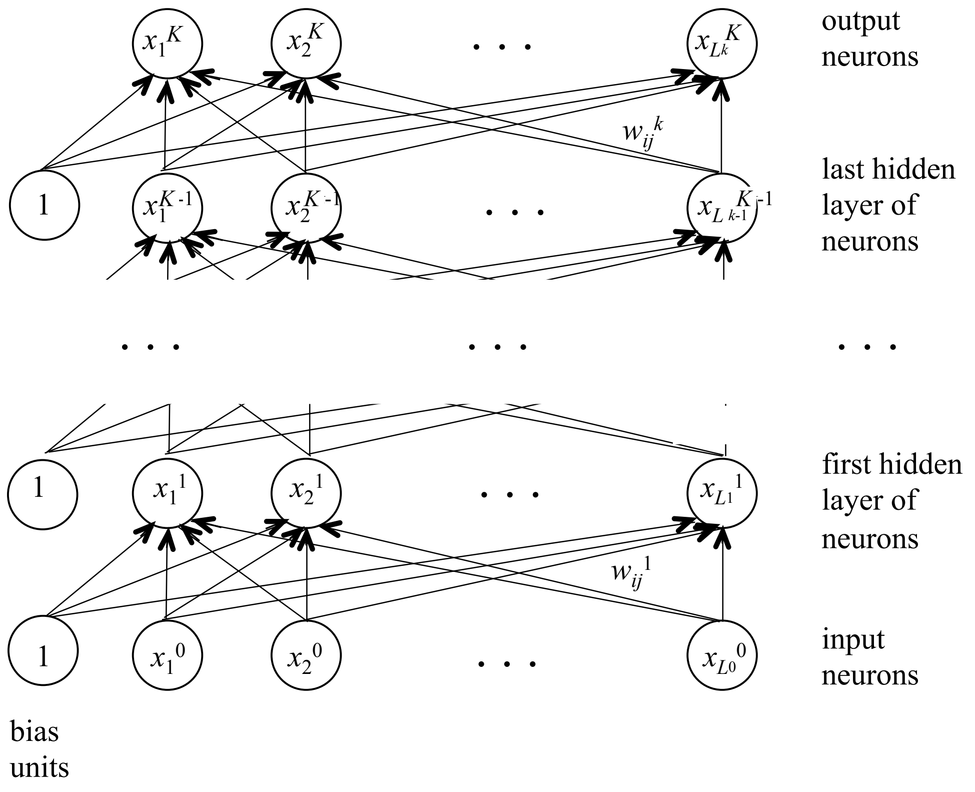


Figure 25: Schema of an MLP with $K-1$ hidden layers of neurons.

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 $\mathcal{N}(u)$ of an argument $u \in \mathbb{R}^{L_0}$ 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 $m < K$, 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

$$x_i^m = \sigma\left(\sum_{j=1}^{L^{m-1}} w_{ij}^m \, x_j^{m-1} + w_{i0}^m\right). \tag{46}$$

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* σ. 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 25).

Equation 46 can be more conveniently written in matrix form. Let $x^m = (x_1^m, \ldots, x_{L^m}^m)'$ be the activation vector in layer m, let $w^m = (w_{10}^m, \ldots, w_{L^m 0}^m)'$ be the vector of bias weights, and let $W^m = (w_{ij}^m)_{i=1,\ldots,L^m;\, j=1,\ldots,L^{m-1}}$ be the connection weight matrix for links between layers $m-1$ and m. Then (46) becomes

$$x^m = \sigma\left(W^m\, x^{m-1} + w^m\right), \tag{47}$$

where the activation function σ is applied component-wise to the activation vector.

3. The L^K-dimensional activation vector $\mathbf{y}$ of the output layer $m = K$ are computed from the activations of the pre-output layer $m = K-1$ by

$$y = x^K = W^K\, x^{K-1} + w^K, \tag{48}$$

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 y is the result $y = \mathcal{N}(u)$.

The activation function σ is traditionally either the hyperbolic tangent (tanh) function or the *logistic sigmoid* given by $\sigma(a) = 1/(1+\exp(-a))$. Figure 26 gives plots of these two S-shaped functions. Functions of such shape are often called *sigmoids*. There are two grand reasons for applying 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 f_{MLP}. Without these sigmoids, f_{MLP} 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 often used, the *rectifier* function defined by $r(a) = 0$ for $a < 0$ and $r(a) = a$ for $a \geq 0$. The rectifier has a little less pleasing mathematical properties compared to the classical sigmoids but can be computed much more cheaply. 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.

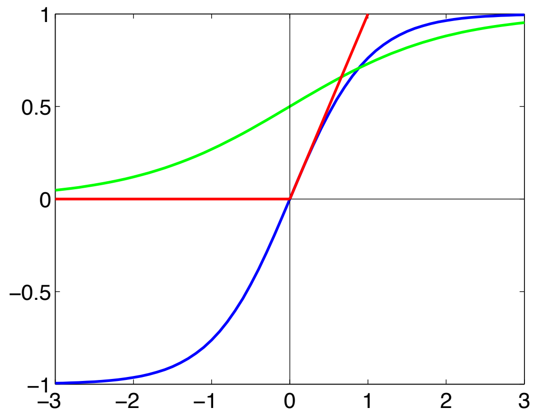


Figure 26: The tanh (blue), the logistic sigmoid (green), and the rectifier function (red).

In intuitive terms, the operation of an MLP can be summarized as follows. After an input vector u is written into the input units, a "wave of activation" sweeps forward through the layers of the network. The activation vector x^m in each layer m is directly triggered by the activations x^{m-1} according to (47). The data transformation from x^{m-1} to x^m is a relatively "mild" one: just an affine linear map $W^m\, x^{m-1} + w^m$ followed by a wrapping with the sigmoid σ. But when several such "mild" transformations are applied in sequence, very complex "foldings" of the input vector u can be effected. 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.

8.2 Universal approximation and "deep" networks

One reason for the popularity of MLPs is that they can approximate arbitrary functions $f : \mathbb{R}^n \to \mathbb{R}^k$ very well. 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 $\|f - \mathcal{N}\| < \varepsilon$.*

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 ε 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 practical implementation. You can find more details concerning such approximation theorems and related results in my ML lecture notes `http://minds.jacobs-university.de/sites/default/files/uploads/teaching/lectureNotes/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 "folds"), it can be in principle 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 2.1 used 17 hidden 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 well-behaved and 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 (GPU-based) yet affordable computing hardware have 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 catch-term *deep learning*.

8.3 Training an MLP: general set-up

Starting from a training sample $S = (u_i, y_i)_{i=1,...,N}$ (where $u \in \mathbb{R}^n, y \in \mathbb{R}^k$), a basic training procedure for an MLP $\mathcal{N}$ goes like follows.

1. **Fix an MLP structure.** Decide how many hidden layers the MLP shall have, how many units each layer shall have, and what sigmoid is used. This initial fixing of a particular *architecture* is nontrivial and will be largely guided by the engineer's experience, or just trial and error search (based on cross-validation experimentation). The structure should be rich enough that data overfitting becomes possible. After the architecture is set, training the

MLP boils down to determine all the weights w_{ij}^m in the chosen architecture. Lumping all these weights together in a parameter vector θ, a given choice of weights gives a particular MLP with the chosen structure which in turn instantiates a function $\mathcal{N}_\theta$.

2. **Fix an empirical risk function with a suitable regularizer.** The task of training an MLP from a sample S is framed as an optimization task, where the goal is to find a function $\mathcal{N}_\theta$ which minimizes an empirical risk function

$$R^{\text{emp}}(\mathcal{N}_\theta) = \sum_{i=1,\dots,N} L(\mathcal{N}_\theta(u_i), y_i) \tag{49}$$

based on the chosed loss function L. A convenient choice is the quadratic loss, equipped with the L_2 norm regularizer

$$R_\alpha^{\text{emp}}(\mathcal{N}_\theta) = \frac{1}{N} \sum_{i=1,\dots,N} \|y_i - \mathcal{N}_\theta(u_i)\|^2 + \alpha^2 \, \|\theta\|^2. \tag{50}$$

Here I used subscript α in $R_\alpha^{\text{emp}}(\mathcal{N}_\theta)$ to indicate that this risk is modulated by a regularizer weighted with α^2. This risk is popular mainly because it leads to simple computations and it often leads to good or at least very useful models $\mathcal{N}$. Experienced ML engineers often employ other functions, but we will not further investigate this issue here.

3. **Find weights θ which minimize the chosen risk function E.** Abstractly speaking, an MLP training algorithm attempts to solve the minimization problem

$$\hat{\theta}_c = \underset{\theta}{\text{argmin}} \; R_c^{\text{emp}}(\mathcal{N}_\theta) \tag{51}$$

at least approximately. A concrete method to solve (51) is outlined in the next Subsections 8.4 and 8.5 — the famous *backpropagation* algorithm. This algorithm is almost always used for MLP training. It is an iterative procedure which leads to a sequence of models $\theta_c^{(0)}, \theta_c^{(1)}, \theta_c^{(2)}, \dots$ with decreasing risks $R^{\text{emp}}(\mathcal{N}_{\theta_c^{(0)}}) > R^{\text{emp}}(\mathcal{N}_{\theta_c^{(1)}}) > R^{\text{emp}}(\mathcal{N}_{\theta_c^{(2)}}) > \dots$. One stops these iterations when further decreases fall below some predetermined threshold, or when one runs out of allotted processing time, or when a validation error starts to increase (see next step 4). The last computed $\theta_c^{(t)}$ is returned as the final model estimate $\hat{\theta}_c$.

4. **Use cross-validation to ensure a good bias-variance tradeoff.** Be aware that efforts for minimizing the *training* error $R^{\text{emp}}(\mathcal{N}_\theta)$ open the doors for overfitting. Steps 1. — 3. above should be embedded in a cross-validation scheme and iteratively be repeated until a lowest validation error is found. Cross-validation schemes imply that models of increasing flexibility are trained and tested. In earlier sections we met to ways how model flexibility can be tuned:

1. Use increasingly large MLPs (more layers or more neurons in the layers). Fix the regularization parameter α in (50) once and for all (possibly to $\alpha = 0$, that is, no regularization).
2. Use a unchanging structure for your MLP with a size large enough to admit overfitting, and make use of the regularization parameter α in your risk function (50). Start with a large α, solve (51), test the model $\hat{\theta}_\alpha$ by computing the average loss on validation data. Increase model flexibility by decreasing α and repeat. The validation loss should first decrease and then increase again (compare, – again! – Figure 20). Stop when the validation loss starts to increase again.

Another method to prevent overfitting is *early stopping*. In this method, the MLP structure is again chosen to be so large that overfitting becomes easily possible. The regularization coefficient α is frozen at some fixed value (possibly zero). The backpropagation iterations at each iteration t are combined with tests of the models $\theta_\alpha^{(t)}$ on some validation data. When the validation error starts to grow, the backpropagation optimization iterations are stopped and the current model is taken as the final model. Early stopping is the method of choice in deep learning scenarios, because here a single run of step 3 is typically very expensive and one cannot afford repetitions of it.

8.4 Model optimization by gradient descent: general principle

We are faced with a training sample S, parametrized models $\mathcal{N}_\theta$, and an empirical risk function $R^{\text{emp}}(\mathcal{N}_\theta)$ that we wish to minimize. There is a standard strategy to tackle the minimization problem (51), namely, iterative optimization by *gradient descent*. This strategy can always be tried when the risk function is differentiable with respect to parameters θ. All error functions that one uses for MLPs have this property. (Except if one uses the rectifier sigmoid; it may lead to non-differentiable situations when the argument of this sigmoid is exactly zero – but this situation occurs with probability close to zero, and if it occurs nonetheless, one can use an arbitrary value for the undefined derivative. Since these situations occur so extremely rarely, the impact of that arbitrary value on the overall optimization search is negligible.)

The general scheme of a gradient-descent optimization goes like this:

1. Fix an initial model $\theta^{(0)}$. This needs an educated guess. A widely used strategy is to set all 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 initial model (Hinton and Salakuthdinov, 2006).

For convenience of notation we group all parameters $w_i^{(0)} \in \theta^{(0)}$ in a vector of size $L = |\theta|$ and identify $\theta^{(0)}$ with this vector, i.e. we represent the parameters as $\theta^{(0)} = (w_1^{(0)}, \ldots, w_L^{(0)})'$.

2. For each $w_i \in \theta$, compute the partial derivative of the risk function with respect to w_i at the position $\theta^{(0)}$:

$$\frac{\partial R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}})}{\partial w_i} \tag{52}$$

   and assemble all of these partials into a row vector

$$\nabla R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}}) = \left(\frac{\partial R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}})}{\partial w_1}, \ldots, \frac{\partial R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}})}{\partial w_L} \right).$$

   This vector is called the *gradient* of the risk $R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}})$ at the point $\theta^{(0)}$. The gradient can be understood as a direction vector in parameter space Θ. It marks the direction of steepest *increase* of $R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}})$ at the point $\theta^{(0)} \in \Theta$.

3. Update $\theta^{(0)}$ by moving a little bit in the direction of steepest *decrease* of the error:

$$\theta^{(1)} = \theta^{(0)} - \lambda^{(1)} \, \nabla R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(0)}}). \tag{53}$$

   The control parameter $\lambda^{(1)} > 0$ is called *stepsize* or *learning rate* or *adaptation rate.*

4. Iterate steps 2. and 3. This gives a sequence of model estimates $\theta^{(n)} = \theta^{(n-1)} - \lambda^{(n)} \, \nabla R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(n-1)}})$. Each $\theta^{(n)}$ should give a slightly smaller risk than $\theta^{(n-1)}$ because the gradient at each step was designed to show just exactly the parameter change direction that reduces the risk most strongly.

5. Stop the iterations when some predetermined termination criterion is met. For instance, one may stop when the gradient vector has shrunken close to the zero vector, that is when $\|\nabla R^{\mathrm{emp}}(\mathcal{N}_{\theta^{(n-1)}})\|$ falls below some small, predetermined threshold. Or one may invoke an early stopping method — then in each iteration n, the current model $\theta^{(n)}$ is tested on some validation dataset and one stops when the validation error starts to grow.

Fixing appropriate stepsizes $\lambda^{(n)}$ is a delicate affair. If set too small, there is little progress in each iteration and the overall optimization process is slowed down. If too big, the adaptation process becomes instable (the risk function then may jump upwards instead of decreasing steadily). Novel, clever methods for online adjustment of stepsize have been one of the enabling factors for deep learning. For shallow MLPs typical stepsizes that can be fixed without much thinking are in the order from 0.001 to 0.01.

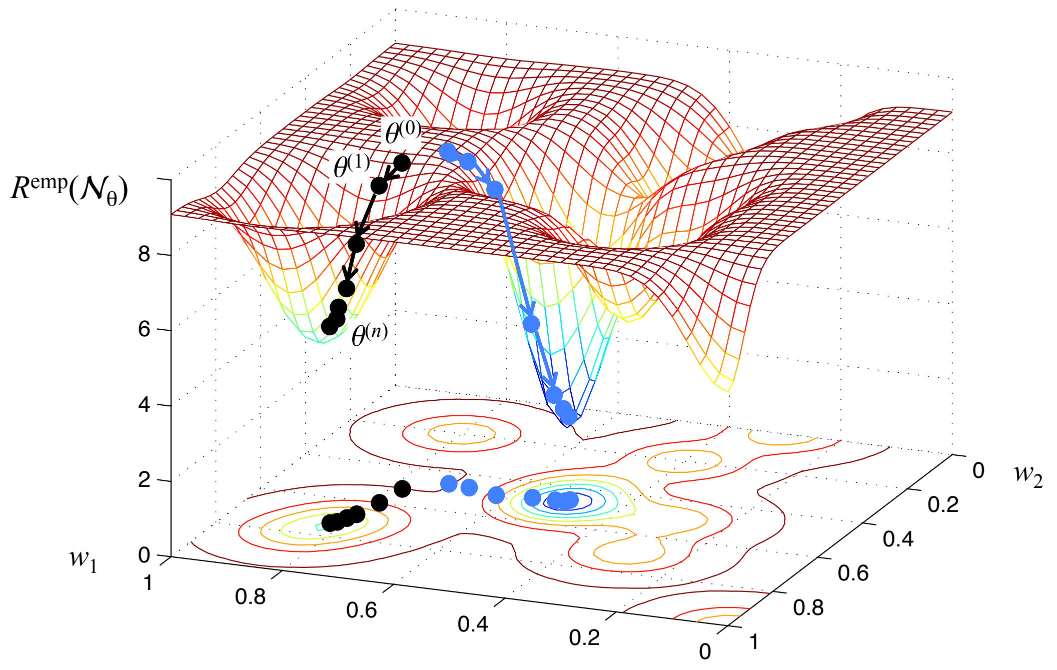


Figure 27: Gradient descent visualized as "ball rolling down the slope" in an error landscape. For explanation see text.

The risk function $R^{\mathrm{emp}}(\mathcal{N}_\theta)$ can be seen as a *performance landscape* (also called *error landscape* because often the risk is a mean square error value) over the parameter space Θ when $R^{\mathrm{emp}}(\mathcal{N}_\theta)$ is thought of as an "elevation" over points θ. Figure 27 visualizes a schematic error landscape over a 2-parameter space $\Theta = \{w_1, w_2\}$.

In a physical metaphor, gradient descent can be understood as a "ball rolling down the slope of the error landscape until it is trapped in a trough". The negative gradient is the direction of steepest descent. The train of black dots in Figure 27 marks a sequence of models $\theta^{(0)}, \theta^{(1)}, \ldots$ that would be computed by the gradient descent procedure. It is obvious from this visualization that the final model $\theta^{(n)}$ will be located at the bottom of the trough that is adjacent to the guessed starting position $\theta^{(0)}$. If the first guess would be different (blue dots in the figure), the iterations might move to another, and possibly better (that is, lower) trough. Gradient descent methods can only find *local minima* of the risk function!

It is clear that the quality of the final estimate $\theta^{(n)}$ crucially depends on the initial guess $\theta^{(0)}$. In principle, one could attempt to find the global optimum by exhaustive grid search: compute the gradient descent solution from all initial $\theta^{(0)}$ that grid-cover all of Θ. This is infeasible for any realistic-sized Θ. In today's deep learning, the strategy to take the sting out of the local optima problem is to use very large networks (which allow overfitting) together with clever regularization methods (which prevent overfitting). Then from almost every halfway reasonable initialization $\theta^{(0)}$ the gradient descent could be continued into an overfitting mode if not early-stopped. In deep learning the challenge is not to get stuck in a local optimum with a too high value of the empirical risk, but to ensure good model generalization properties at the empirical risk that one decides to (early) stop at.

8.5 The backpropagation algorithm

Let us take a closer look at the empirical risk (49). Its gradient can be written as a sum

$$\nabla R^{\text{emp}}(\mathcal{N}_\theta) = \nabla \left(\frac{1}{N} \sum_{i=1,\ldots,N} L(\mathcal{N}_\theta(u_i), y_i) \right) = \frac{1}{N} \sum_{i=1,\ldots,N} \nabla L(\mathcal{N}_\theta(u_i), y_i),$$

and this is also how it is actually computed: for each training data example (u_i, y_i), the gradient $\nabla L(\mathcal{N}_\theta(u_i), y_i)$ is evaluated 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 (u_i, 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 — or "backprop" for short — or even just "BP" for even shorter — is a subroutine in the gradient descent game. It is a particular algorithmic scheme for calculating the gradient $\nabla L(\mathcal{N}_\theta(u_i), y_i)$ for a single data point (u_i, y_i). You know from calculus courses that computing derivatives of a function is a mechanical thing where certain derivation rules have to be applied in a natural sequence. However, when one computes a gradient $\nabla L(\mathcal{N}_\theta(u_i), y_i)$ for an MLP in this mechanically straightforward fashion, one earns a cost of $O(L^2)$. When the number L of network weights is large (a few hundreds for small MLPs used for simple tasks, and easily half a million for deep networks applied to serious real-life modeling problems), this cost $O(L^2)$ is too high for practical exploits. The backprop algorithm is a clever scheme for computing and storing certain auxiliary quantities which cuts down the cost from $O(L^2)$ to $O(L)$.

Here is how backprop works.

1. *Given:* an MLP $\mathcal{N}_\theta$ with parameters (that is, weights) θ which have either been randomly fixed (at the beginning of training, before the first epoch) or which are the result from the previous epoch.

2. *Wanted:* the gradient $\nabla L(\mathcal{N}_\theta(u), y)$, where (u, y) is one of the training data pairs. Notice that when a training example (u, y) is fixed, $L(\mathcal{N}_\theta(u), y)$ can be regarded as a function from parameter space $\mathbb{R}^L$ to the nonnegative reals (as in Figure 27). That is, for every network weight w^m_{ij} (including the bias weights w^m_{i0}, see Equation 46) we wish to calculate

$$\frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial w^m_{ij}}. \tag{54}$$

   This is what the BP algorithm does.

3. BP works in two stages. In the first stage, called the *forward pass*, the network $\mathcal{N}_\theta$ is presented with the input u and the output $\hat{y} = \mathcal{N}_\theta(u)$ is computed using the "forward" formulas (47) and (48). During this forward pass, for each unit x^m_i which is not a bias unit and not an input unit (that is, $m \geq 1$ and $i \neq 0$) the quantity

$$a^m_i = \sum_{j=0,\ldots,L^{m-1}} w^m_{ij}\, x^{m-1}_j \tag{55}$$

   is computed – this is sometimes referred to as the *potential* of unit x^m_i, that is its internal state before it is passed through the sigmoid.

4. *A little math in between.* Applying the chain rule of calculus we have

$$\frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial w^m_{ij}} = \frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial a^m_i}\, \frac{\partial a^m_i}{\partial w^m_{ij}}. \tag{56}$$

   Define

$$\delta^m_i = \frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial a^m_i}. \tag{57}$$

   Using (55) we find

$$\frac{\partial a^m_i}{\partial w^m_{ij}} = x^{m-1}_j. \tag{58}$$

   Combining (57) with (58) we get

$$\frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial w^m_{ij}} = \delta^m_i\, x^{m-1}_j. \tag{59}$$

   Thus, in order to calculate the desired derivatives (54), we only need to compute the values of δ^m_i for each hidden and output unit.

5. *Computing the δ's for output units.* For output units x^K_i we did not use a sigmoid, see (48). The potentials a^K_i are thus identical to the output values $\hat{y}_i$ and we obtain

$$\delta^K_i = \frac{L(\mathcal{N}_\theta(u), y)}{\partial y_i}. \tag{60}$$

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 $L(\mathcal{N}_\theta(u), y) = \|\mathcal{N}_\theta(u) - y\|^2$, for instance, we get

$$\delta_i^K = \frac{\partial \|\mathcal{N}_\theta(u) - y\|^2}{\partial y_i} = \frac{\partial \|\hat{y} - y\|^2}{\partial y_i} = \frac{\partial (\hat{y}_i - y_i)^2}{\partial y_i} = 2\,(y_i - \hat{y}_i). \qquad (61)$$

6. *Computing the δ's for hidden units.* In order to compute δ_i^m for $1 \le m < K$ we again make use of the chain rule. We find

$$\delta_i^m = \frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial a_i^m} = \sum_{l=1,\ldots,L^{m+1}} \frac{\partial L(\mathcal{N}_\theta(u), y)}{\partial a_l^{m+1}} \, \frac{\partial a_l^{m+1}}{\partial a_i^m}, \qquad (62)$$

which is justified by the fact that the only path by which a_i^m can affect $L(\mathcal{N}_\theta(u), y)$ is through the potentials a_l^{m+1} of the next higher layer. If we substitute (57) into (62) and observe (55) we get

$$\begin{aligned} \delta_i^m &= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \, \frac{\partial a_l^{m+1}}{\partial a_i^m} \\ &= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \, \frac{\partial \sum_{j=0,\ldots,L^m} w_{lj}^{m+1} \, \sigma(a_j^m)}{\partial a_i^m} \\ &= \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \, \frac{\partial \, w_{li}^{m+1} \, \sigma(a_i^m)}{\partial a_i^m} \\ &= \sigma'(a_i^m) \sum_{l=1,\ldots,L^{m+1}} \delta_l^{m+1} \, w_{li}^{m+1}. \qquad (63) \end{aligned}$$

This formula describes how the δ_i^m in a hidden layer can be computed by "back-propagating" the δ_l^{m+1} from the next higher layer. The formula can be used to compute all δ's, starting from the output layer (where (60) is used or, in the case of a quadratic loss, Equation 61), 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 $\sigma'(a) = \sigma(a)\,(1 - \sigma(a))$, which leads to

$$\sigma'(a_i^m) = x_i^m \,(1 - x_i^m).$$

Although simple in principle, and readily implemented, using the backprop algorithm appropriately is something of an art. Here is only place to hint at some difficulties:

- The stepsize λ in (53) 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. It is however not possible to provide an analytical treatment of how to set the stepsize optimally. Generally one uses larger stepsizes in early epochs.

- Gradient descent on nonlinear performance landscapes may sometimes be very slow in areas where the gradient is small in some directions.

- 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.

- Finally, finding a network structure (number of units, number of layers) that is appropriate for a given task is not trivial. A decent amount of experimentation and cross-validation exploration may be needed.

The backpropagation "trick" to speed up gradient calculations in layered systems has been independently discovered several times and in a diversity of contexts, not only in neural network or machine learning 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 et al., 1986). From that point onwards, backpropagation in MLPs developed into what today is likely the most widely used model 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. A number of refined "second order" gradient descent methods have been developed which need fewer epochs to converge, but where each epoch takes longer to compute. 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 `http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/NNalgs.zip`).

Appendix

A 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 very many other items). The marketing optimizers of that shop are not interested in the exact age but only in age brackets, say $a_1 = \mathsf{at\ most\ 10\ years\ old}$, $a_2 = 11 - 20$ years, $a_3 = 21 - 30$ years, $a_4 = \mathsf{older\ than\ 30}$. Gender is roughly categorized into the possibilities $g_1 = \mathsf{f}, g_2 = \mathsf{m}, g_3 = \mathsf{o}$. 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
g_2	0.005	0.15	0.15	0.04
g_3	0.0	0.05	0.05	0.01

(64)

The cell (i, j) in this 3×4 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 (64), instead of $P(X = g_i, Y = a_j)$ we could have written $P(\mathsf{Gender} = g_i, \mathsf{Age} = a_j)$. 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 (65).

	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**	

(65)

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:

$$P(X = g_i \,|\, Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(Y = a_j)}. \tag{66}$$

There are two equivalent versions of this formula:

$$P(X = g_i, Y = a_j) = P(X = g_i \,|\, Y = a_j)P(Y = a_j) \tag{67}$$

and

$$P(Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(X = g_i \,|\, Y = a_j)}, \tag{68}$$

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.

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 $s_1 =$ less than 5 Euros to $s_{20} =$ 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:

$$P(X = g_i, Z = s_j) = \sum_{k=1,2,3,4} P(X = g_i, Y = a_k, Z = s_j) \tag{69}$$

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 28). 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.

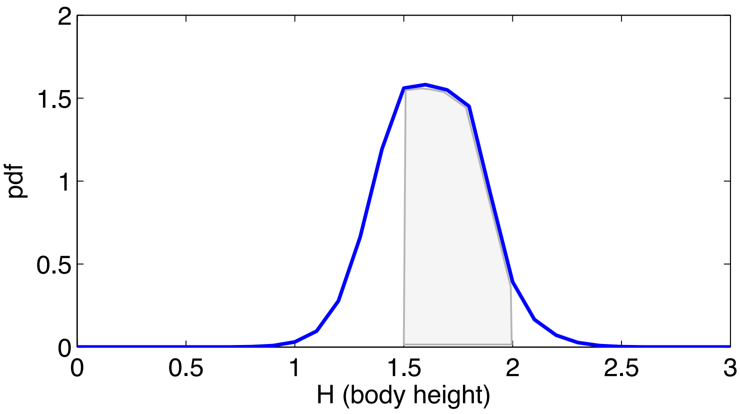


Figure 28: A hypothetical distribution of human body sizes in some reference population, represented by a pdf.

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 body height comes out between 1.5 and 2.0 meters is obtained by

$$P(H \in [1.5, 2.0]) = \int_{1.5}^{2.0} f_H(x)dx, \tag{70}$$

see the shaded area in Figure 28. 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 70. 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 28), 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 29 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.

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.

$$\int_{a_1}^{b_1} \ldots \int_{a_k}^{b_k} f(x_1, \ldots, x_k)\, dx_k \ldots dx_1,$$

where also the cases $a_i = -\infty$ and $b_i = \infty$ are possible. A more compact notation for the same integral is

$$\int_D f(\mathbf{x})\, d\mathbf{x},$$

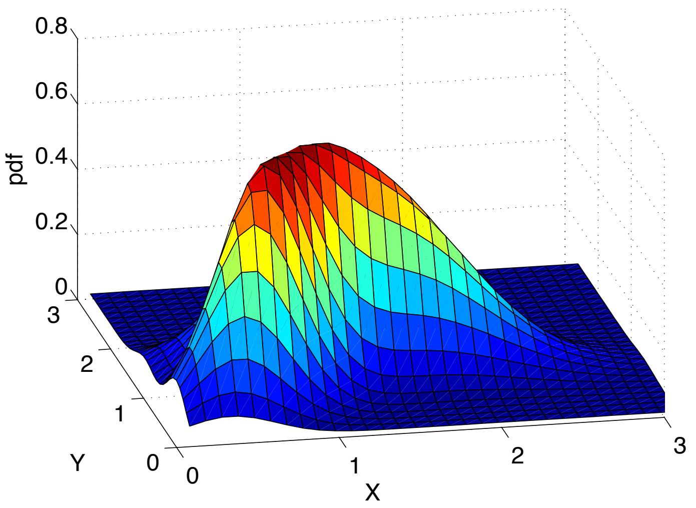


Figure 29: An exemplary joint distribution of two continuous-valued RVs X, Y, represented by its pdf.

where D denotes the k-dimensional "box" $[a_1, b_1] \times \ldots \times [a_k, b_k]$ and $\mathbf{x}$ 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 28, probabilities are obtained from a k-dimensional pdf $f_{X_1,\ldots,X_k}$ by integrating over sub-intervals. For such a k-dimensional 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

$$P(X_1 \in [r_1, s_1], \ldots, X_k \in [r_k, s_k]) = \int_{r_1}^{s_1} \ldots \int_{r_k}^{s_k} f(x_1, \ldots, x_k)\, dx_k \ldots dx_1. \quad (71)$$

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 (69), for instance, one would get

$$f_{X_1,X_3}(x_1, x_3) = \int_{a_2}^{b_2} f_{X_1,X_2,X_3}(x_1, x_2, x_3)\, dx_2. \quad (72)$$

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 (66) 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 $P(X \in [a, b] \mid Y = c)$, using pdfs.

We can obtain this probability from the joint pdf $f_{X,Y}$ and the marginal pdf f_Y by

$$P(X \in [a,b] \,|\, Y = c) = \frac{\int_a^b f_{X,Y}(x,c)\, dx}{f_Y(c)}. \tag{73}$$

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

$$f_{X\,|\,Y=c}(x) = \frac{\int_a^b f_{X,Y}(x,c)\, dx}{f_Y(c)}. \tag{74}$$

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.

More generally speaking, conditional distributions inform a scientist about cause-effect relationships. 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 this appendix (and in the lecture) I present 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 (among other things) an introduction to probability theory in the graduate lecture "Principles of Statistical Modeling" – you can register for this course as a 3rd year specialization course. 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 very difficult concept since it is actually a family of distributions) of X given Y.

B The argmax operator

Let $\varphi : D \to \mathbb{R}$ be some function from some domain D to the reals. Then

$$\operatorname*{argmax}_a \varphi(a)$$

is that $d \in D$ for which $\varphi(d)$ is maximal among all values of φ on D. If there are several arguments a for which φ gives the same maximal value, – that is, φ does not have a unique maximum –, or if φ has no maximum at all, then the argmax is undefined.

For example, in the expression $\operatorname*{argmax}_j P(Y = c_j)\, f_j(x)$ (see Equation (6)) the domain D is the set of classes $D = \{1, \ldots, k\}$ and the function φ takes its arguments i to the real numbers $P(Y = c_j)\, f_j(x)$.

C Derivation of Equation 14

$$\begin{aligned}
&1/N \sum_i \|x_i - \mathbf{d} \circ \mathbf{f}(x_i)\|^2 = \\
&= 1/N \sum_i \|\bar{x}_i - \sum_{k=1}^{m} (\bar{x}_i'\, u_k)\, u_k\|^2 \\
&= 1/N \sum_i \| \sum_{k=1}^{n} (\bar{x}_i'\, u_k)\, u_k - \sum_{k=1}^{m} (\bar{x}_i'\, u_k)\, u_k\|^2 \\
&= 1/N \sum_i \| \sum_{k=m+1}^{n} (\bar{x}_i'\, u_k)\, u_k\|^2 \\
&= 1/N \sum_i \sum_{k=m+1}^{n} (\bar{x}_i'\, u_k)^2 = \sum_{k=m+1}^{n} 1/N \sum_i (\bar{x}_i'\, u_k)^2 \\
&= \sum_{k=m+1}^{n} \sigma_k^2.
\end{aligned}$$

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 data value space. This is the set of values that might be delivered by the random variable. For instance, the data value space of the gender RV G could be cast as $\{\mathrm{m}, \mathrm{f}, \mathrm{o}\}$ – a symbolic (and finite) set. A reasonable data value 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 data value space for the spending RV S could be just the real numbers $\mathbb{R}$.

Note that in the A and S examples, the data value 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 data value space. The important thing about a data value 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 data value space. We just saw symbolic, integer, and real data value spaces. Real data value 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 data value 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 data value 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 data value space $\mathbb{R}^n$ and pdf f_X is defined as

$$E[X] = \int_{\mathbb{R}^n} x\, f_X(x)\, dx, \tag{75}$$

where the integral is written in a common shorthand for

$$\int_{x_1=-\infty}^{\infty} \cdots \int_{x_n=-\infty}^{\infty} (x_1, \ldots, x_n)' \, f_X((x_1, \ldots, x_n)) \, dx_n \ldots dx_1.$$

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 $(x_i, y_i)_{i=1,\ldots,N}$ drawn from the joint distribution $P_{X,Y}$ of two RVs X, Y, we may compute the average value of the x_i by

$$\text{mean}(\{x_1, \ldots, x_N\}) = 1/N \sum_{i=1}^{N} x_i,$$

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 data value space $\mathbb{R}$ is the expected squared deviation from the expectation

$$\sigma^2(X) = E[\bar{X}^2], \qquad (76)$$

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 $(x_i)_{i=1,\ldots,N}$ 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

$$\mathrm{Cov}(X,Y) = E[\bar{X}\,\bar{Y}], \tag{77}$$

which in terms of a pdf $f_{\bar{X}\,\bar{Y}}$ for the joint distribution for the centered RVs spells out to

$$\mathrm{Cov}(X,Y) = \int_{\mathbb{R}\times\mathbb{R}} x\,y\,f_{\bar{X}\,\bar{Y}}((x,y)')\,dx\,dy.$$

An unbiased estimate of the covariance, based on a sample $(x_i, y_i)_{i=1,\ldots,N}$ is given by

$$\widehat{\mathrm{Cov}}((x_i,y_i)_{i=1,\ldots,N}) = 1/(N-1) \left(x_i - 1/N \sum_i x_i \right) \left(y_i - 1/N \sum_i y_i \right).$$

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

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sigma(X)\,\sigma(Y)}. \tag{78}$$

It is easy to show that $-1 \leq \mathrm{Corr}(X,Y) \leq 1$. 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 $\mathrm{Corr}(X,Y) = 0$, X and Y are said to be *uncorrelated*.

The quantity $\mathrm{Corr}(X, Y)$ is also referred to as *(population) Pearson's correlation coefficient*, and is often denoted by the greek letter $\varrho(X, Y) = \mathrm{Corr}(X, Y)$.

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

$$E[X\,Y],$$

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 α.

2. Expectation is idempotent:

   $$E[E[X]] = E[X].$$

3. $$\mathrm{Cov}(X, Y) = E[X\,Y] - E[X]\,E[Y].$$

References

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`.

Y. Bengio and Y. LeCun. Scaling learning algorithms towards AI. In Bottou L., Chapelle O., DeCoste D., and Weston J., editors, *Large-Scale Kernel Machines*. MIT Press, 2007.

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (second edition)*. Wiley Interscience, 2001.

S. Edelman. The minority report: some common assumptions to reconsider in the modelling of the brain and behaviour. *J of Experimental and Theoretical Artificial Intelligence*, 2015. URL `http://www.tandfonline.com/action/showCitFormats?doi=10.1080/0952813X.2015.1042534`.

B. Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. Wiley, 1998.

et al. Graves, A. Hybrid computing using a neural network with dynamic external memory. *Nature*, 7626:471–476, 2016.

G. E. Hinton and R. R. Salakuthdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(July 28):504–507, 2006.

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.

J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998. http://ict.ewi.tudelft.nl/∼duin/papers/pami_98_ccomb.pdf.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. 2013. URL `http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionali pdf`.

T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proc. of BMVC*, 2915. URL `http://www.robots.ox.ac.uk:5000/~vgg/publications/2015/Parkhi15/parkhi15.pdf`.

S. E. Peters, C. Zhang, M. Livny, and C. Ré. A machine reading system for assembling synthetic paleontological databases. *PLOS-ONE*, 9(12): e113523, 2014. URL `http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0113523`.

F. Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11(2):305–345, 1999.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing Vol. 1*, pages 318–362. MIT Press, 1986. Also as Technical Report, La Jolla Inst. for Cognitive Science, 1985.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Preprint: arXiv:1404.7828.

D. Silver et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

P. Young, A. Lai, M. Hodosh, and J. Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *Transactions of the Association for Computational Linguistics*, 2: 67–78, 2014.