Artificial Neural Networks

A Basic Course at IK 2008

Herbert Jaeger

Jacobs University Bremen

Course overview

1 Introduction

2 Feedforward Networks

3 Hopfield Networks and Boltzmann Machines

4 Recurrent Neural Networks for Time Series Processing

Session 2: Feedforward networks

1.1 For starters: the Perceptron

1.2 Multilayer perceptrons and the backpropagation algorithm

1.3 Convolutional Networks

2.1 For starters: the Perceptron

The forefather of artificial neural networks (ANNs)

Developed by the US American (neuro-)psychologist Frank Rosenblatt at the Cornell Aeronautical Laboratory

First techreport 1957, book publication 1962 (*Principles of Neurodynamics*)

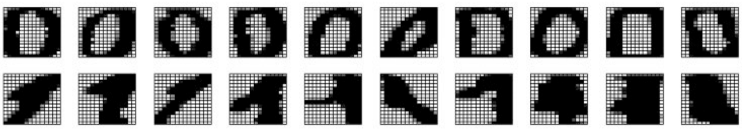
Contemporary with beginnings of symbolic Artificial Intelligence

Has all the standard features of the ANNs of today:

- A specific task (binary classification)
- A biologically oriented "architecture" of interconnected, abstracted neural processing elements
- A learning algorithm
- A mathematical analysis
- A state-of-the-art computer implementation
- A history of hype

The Perceptron's task: binary pattern classification

Typical data format: binary (black & white) pixel images from two classes, for instance handwritten zero and one images[1]:



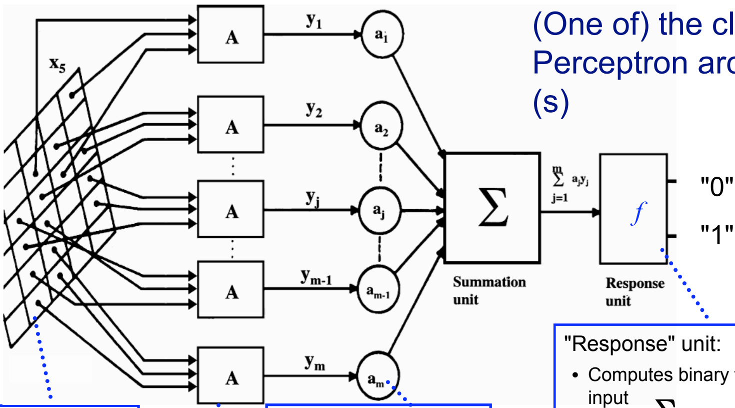
Task: binary classification, that is, upon input of such a pixel image, compute the following...

Output = "0", if the input image is an image of a zero

Output = "1", if the input image is an image of a one

1) Data from Robert Duin, http://ftp.ics.uci.edu/pub/ml-repos/machine-learning-databases/mfeat/mfeat-pix

(One of) the classical Perceptron architecture (s)



Input "retina"

Trainable connection weigths

"Associator" units:

- randomly connected to (few) retina cells
- can compute arbitrary Boolean functions ("predicates") of their inputs

"Response" unit:

- Computes binary function f of input $\sum a_j y_j$
- Typical choice: threshold unit

$$f(u) = \begin{cases} 0, \text{ if } u + b < 0 \\ 1, \text{ if } u + b \geq 0 \end{cases}$$

bias

Picture taken from Kanal (2001)

Early implementations

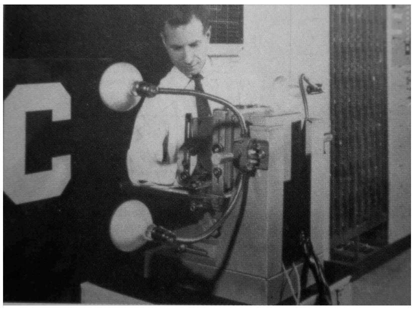
First implementations by Rosenblatt on an IBM 704 programmable digital computer at the Cornell Aeronautical Laboratory.

Rosenblatt went on to build an analog hardware implementation, the Mark 1 Perceptron. His rationale is very interesting:

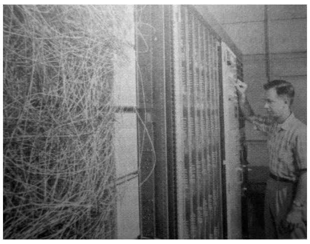
"As the number of connections in the network increases, ... the burden on a conventional digital computer soon becomes excessive, and it is anticipated that some of the models now under consideration may require actual construction before their capabilities can be fully explored. [...] The results of these [digital] programs, therefore, should be interpreted as indicating performances which might be expected from an "ideal" ... system, and not necessarily as representative of any particular engineering design. A Mark I perceptron, ... is expected to provide data on the performance of an actual physical system, which should be useful for comparative study." (Rosenblatt 1960)

A perfectly outdated attitude, right? *Well... actually, no.*

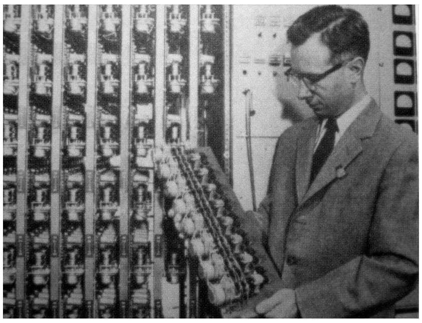
The Mark 1 perceptron



Pattern input: brightly lit B/W patterns taken by an array of 20 x 20 photocells



Input wiring



Adaptive weights realized by motor-driven potentiometers

(all images from Bishop 2006, Chapter 4.1)

Historical and modern perceptrons

Rosenblatt's classical perceptrons...

- Came in many variations
- incorporated "biologically plausible" features and terminology, e.g.
  - multilayer structure loosely modelled after visual system of mammals
  - random, local connectivity from retina to associator unit

Modern textbook perceptrons...

- Stripped off everything mathematically redundant
- Formal model becomes

$$y = f\left(\sum_{j=1,\ldots,m} w_j x_j + b\right)$$

where $y \in \{0,1\}$ is the output, f defined by $f(u) = 1$ iff $u \geq 0$ else $f(u) = 0$ is the unit step function, x_j are inputs (raw pixel values or preprocessed features – was y_j on previous slide), w_j are trainable weights – was a_j on previous slide, b is a constant bias.

The perceptron in today's terminology and notation

Basic formula: $y = f(\sum_{j=1,\dots,m} w_j x_j + b) = f(\mathbf{w}'\mathbf{x} + b)$

Where $y \in \{0,1\}$, where ' denotes transpose, $\mathbf{w} = (w_1,\dots,w_m)'$ is the weight vector, $\mathbf{x} = (x_1,\dots,x_m)'$ is the input pattern vector, f is the unit step function, b is a bias constant.

An equivalent, more convenient variant: $y = f(\tilde{\mathbf{w}}'\tilde{\mathbf{x}})$

Where $\tilde{\mathbf{w}} = (b, w_1,\dots,w_m)'$ is an extended weight vector, and $\tilde{\mathbf{x}} = (1, x_1,\dots,x_m)'$ is an input pattern with additional constant input 1.

This trick is common practice in the field of ANNs, and we will use it without further mention. That is, we will use the simple formula

$$\boxed{y = f(\mathbf{w}'\mathbf{x})}$$

and understand it in the sense of bias-included variant.

The classification task

Remember our perceptron formula: $y = f(\mathbf{w}'\mathbf{x})$

Task specification: given a family P of classified patterns

$$P = ((\mathbf{x}_i, y_i))_{i \in I}$$

where the $\mathbf{x}_i \in \mathbb{R}^m$ are m-dimensional pattern vectors (which includes a constant 1 input), and the $y_i \in \{0,1\}$ are class labels,

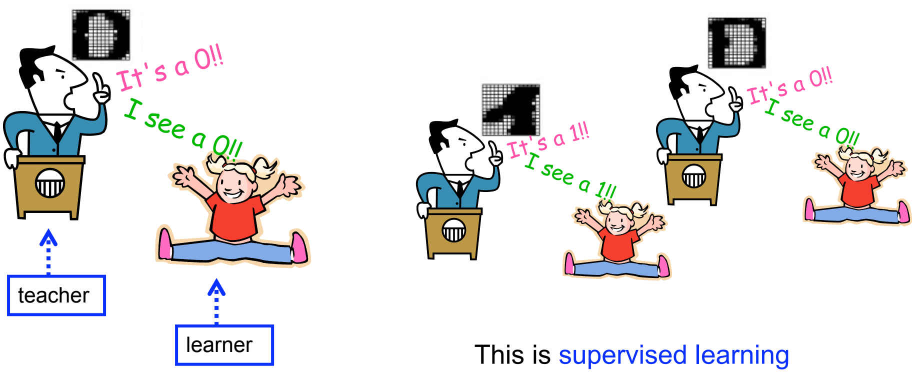
find weights $\mathbf{w}$ such that for all instances i,

$$y_i = f(\mathbf{w}'\mathbf{x}_i),$$

that is, the trained perceptron computes the correct class labels when given inputs from either of the two pattern classes.

How do humans learn 1's and 0's (and even more)?

One way how humans acquire tricky feats is by learning from a teacher, essentially trying to become able of imitating them.



This is supervised learning (or supervised training).

The learning task for perceptrons

Remember the basic formula: $y = f(\mathbf{w}'\mathbf{x})$

Remember the task specification: given classified patterns $P = ((\mathbf{x}_i, y_i))_{i \in I}$, find weights $\mathbf{w}$ such that for all i: $y_i = f(\mathbf{w}'\mathbf{x}_i)$.

This task is solved by supervised learning:

- The perceptron is repeatedly "shown" samples from the set P
- At each presentation, say the n-th one, where the example $(\mathbf{x}(n), y(n))$ is shown, the current weights $\mathbf{w}(n-1)$ are slightly adapted to become $\mathbf{w}(n)$ according to the perceptron learning rule:

Case 1: $y(n) = f(\mathbf{w}'(n-1)\ \mathbf{x}(n))$, that is, correct classification of n-th training sample with old weights $\mathbf{w}(n-1)$. Then, no change: $\mathbf{w}(n) = \mathbf{w}(n-1)$.

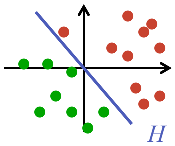
Case 2a: $y(n) = 1 \neq 0 = f(\mathbf{w}'(n-1)\ \mathbf{x}(n))$, false classification of class-1 sample as class-0. Then adapt $\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{x}(n)$.

Case 2b: $y(n) = 0 \neq 1 = f(\mathbf{w}'(n-1)\ \mathbf{x}(n))$, false classification of class-0 sample as class-1. Then adapt $\mathbf{w}(n) = \mathbf{w}(n-1) - \mathbf{x}(n)$.

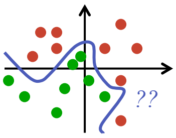
Perceptron learning rule convergence

Theorem (Rosenblatt 1962) (simpli- and sloppified formulation): Let the two classes C_0 and C_1 of the family $P = ((\mathbf{x}_i, y_i))_{i \in I}$ be linearly separable. Then the perceptron learning rule will converge after a finite number of steps, that is, after some step n_0, $\mathbf{w}(n_0) = \mathbf{w}(n_0 + 1) = \mathbf{w}(n_0 + 2)$ ...

Definition. Let $P_0, P_1 \subseteq \mathbb{R}^n$ be two subsets of $\mathbb{R}^n$. Then P_0 and P_1 are linearly separable if there is an n–1-dimensional hyperplane $H \subseteq \mathbb{R}^n$ such that P_0 and P_1 fall on the two different sides of this hyperplane.



Linearly separable sets in $\mathbb{R}^2$



Linearly inseparable sets

Note. P_0 and P_1 are linearly separable in $\mathbb{R}^n$ if and only if there is some $\mathbf{w} \in \mathbb{R}^n$, such that for all $p \in P_0$ it holds that $\mathbf{w}'p < 0$, while for all $p \in P_1$ it holds that $\mathbf{w}'p \geq 0$ – or vice versa.

Thus, P_0 and P_1 are linearly separable if and only if there exists some perceptron that can classify them.

The hype

A sniplet from the New York Times (*"New Navy Device Learns by Doing",* NYT July 8, 1958), after a press conference held by Rosenblatt at the US Office of Naval Research on July 7, 1958 (cited after Olazaran 1996):

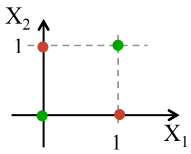
The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language, it was predicted.[17]

The shattering

1969: Marvin Minsky and Seymour Papert, pioneers of (symbolic) AI, publish *Perceptrons.*

In this book they strip down the "biologistic" original perceptron account to the bare-boned textbook understanding that we also considered here and point out, among other things, that the XOR function is not linearly separable and thus cannot be learnt by perceptrons.

X_1	X_2	XOR(X_1, X_2)
0	0	0
0	1	1
1	0	1
1	1	0



As a consequence of this simple insight ("perceptrons can't even learn the simplest Boolean functions"), ANNs become disreputable as an approach to realize machine intelligence for a decade, and research in the field takes a dive...

2.2 Multilayer perceptrons and the backpropagation algorithm

The flows of Science...

Rosenblatt's (and his contemporaries') perspective was a (one might say, romantic) mixture of ideas concerning

- Human cognitive processing
- Brain physiology
- Computing machinery (digital and analog)
- Applications
- Mathematics

Over the years, these views differentiated and separated (cf. the "habitats" of neural networks)

In Machine Learning, the mathematics (and applications) angle is dominating

This has led to a completely unromantic view on ANNs (and other ML methods)

The completely unromantic view on ANNs in ML

ANNs (as any other formalism in machine learning) are seen as trainable representations of some general types of mathematical objects:

- Functions (we'll stick to this for a while)
- Probability distributions (see session 3)
- Dynamical systems (see session 4)

The most basic view on ANNs in ML is that they are function approximators.

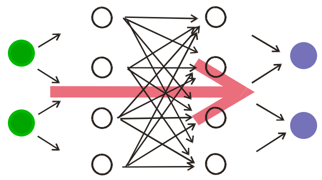
Almost every "intelligent" information processing task can be cast as a function, for example:

- Classification and pattern recognition (function from pattern set to class label set)
- Time series prediction (function from previous history to future observation)
- Evaluation (function from input pattern to degrees of belief / quality / reward)

Function approximator = feedforward ANN

A function $f: U \longrightarrow Y$ maps to each input argument $\mathbf{u} \in U$ (exactly) one output value $\mathbf{y} \in Y$.

In feedforward ANNs, the connections between neurons are all directed "from left to right" (figuratively speaking):

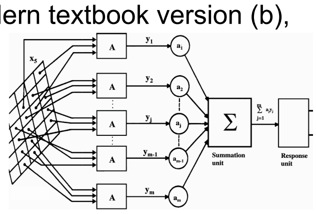


Upon presentation of an input pattern $\mathbf{u}$, a "wave" of activation sweeps through the network, transforming the input, until at the ouput layer an output pattern $\mathbf{y}$ is obtained.

The perceptron as a function approximator

The perceptron, whether in the original "romantic" version (a) or the modern textbook version (b),

(a)



(b) $\boxed{y = f(\mathbf{w}'\mathbf{x})}$

is a feedforward ANN und thus "just" implements a function f.

The shattering of the perceptron hype by Minsky and Papert amounts to the observation that perceptrons can implement only rather limited functions:

- Functions with values in $\{0, 1\}$ that can be described by a linear "decision hyperplane".

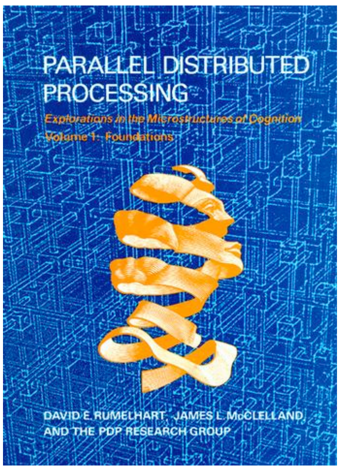
1969: Minsky and Papert point out the obvious.

1986, -- 17 years later...

Neural networks reappear with a flourish:

Rumelhart, D.E. and McClelland, J.L. (eds.): *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol 1, MIT Press (still in print!)

The PDP bible



An edited collection of (long, foundational) articles

Two landmark chapters:

- G.E. Hinton and T.J. Sejnowski: *Learning and Relearning in Boltzmann Machines* (session 3)
- D.E. Rumelhart, G.E. Hinton, and R.J. Williams: *Learning Internal Representations by Error Propagation* (this session)

The Rumelhart/Hinton/Williams chapter (re-)introduces the error backpropagation, or simply "backprop" algorithm

This algorithms allows, in principle, to train feedforward ANNs to represent essentially *any* nonlinear function f.

Backprop is typically applied to multilayer feedforward ANNs, aka "multilayer perceptrons".

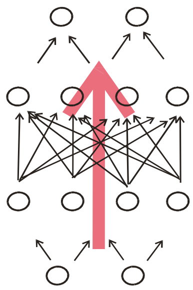
Multilayer perceptrons: architecture

Output layer

Hidden layer 2

Hidden layer 1

Input layer



$$x_i^m = g\Big(\sum_{\text{units } j \text{ in layer } m-1} w_{ij}^m \; x_j^{m-1}\Big)$$

x_i^m i-th unit in layer m

w_{ij}^m weight of connection from unit x_j^{m-1} to x_i^m

g output activation function

- MLP is made of many individual processing elements (neurons, units)
- Units are ordered in layers: input, hidden, output
- Number of input units is dimension of input arguments, number of output units dim of output values
- Units are connected by directed, weighted links (synapses, connections)
- MLP from figure is a "three-layer perceptron" (layers of *connections* are counted)
- Each unit represents a real number x (activation)
- Input is written into input layer by setting activations to input arguments
- Activation is propagated forward through network along connections
- Each unit (except input) computes its activation according to formula on the left: a weighted sum of activations of ingoing units, passed through an activation function
- Activations of output units are read out as value vector

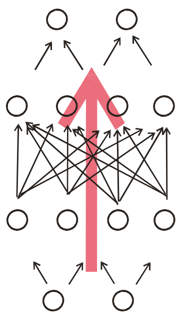
Multilayer perceptrons: single neuron update

Output layer

Hidden layer 2

Hidden layer 1

Input layer

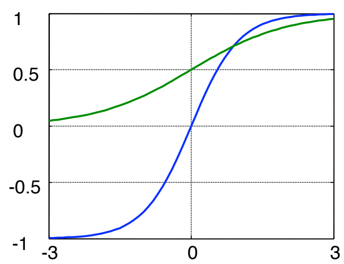


$$x_i^m = g\Big(\sum_{\text{units } j \text{ in layer } m-1} w_{ij}^m \; x_j^{m-1}\Big)$$

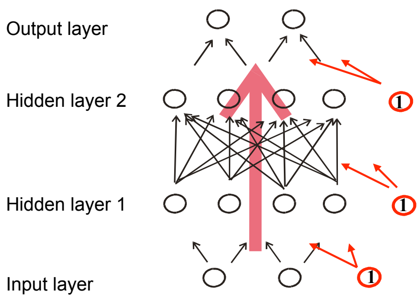
Activation function g is a sigmoid ("s-shaped") function

There are two standard options for g:

- the logistic function $\boxed{g(a) = \dfrac{1}{1+e^{-a}}}$
- the hyperbolic tangent $\boxed{g(a) = \tanh(a) = \dfrac{e^{a} - e^{-a}}{e^{a} + e^{-a}}}$

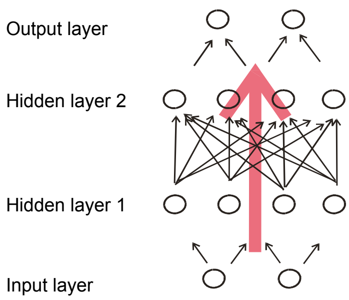


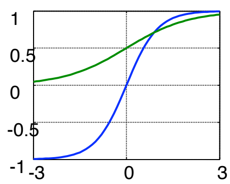
Multilayer perceptrons: bias units



- A frequent modification to the basic architecture
- Each neuron layer (except outputs) gets an additional bias unit whose activation is frozen at 1.

Multilayer perceptrons: special treatment of output units





Problem: using a sigmoid activation function g in the output units makes it impossible to learn output values greater than 1.

Solutions:

(a) Use teacher values $g(y)$ instead of y; in applying the trained network, transform network output by g^{-1} to recover original output range.

(b) Use linear output units (drop g in output layer)

MLP training

Objective: for a given MLP structure, minimize the training squared error

$$SE_{\text{train}} = \sum_{n=1,\dots,N} \left\| \mathbf{y}_n - \tilde{\mathbf{y}}_n \right\|^2 = \sum_{n=1,\dots,N} E(n)$$

where $\mathbf{y}_n$ is the correct teacher output value, $\tilde{\mathbf{y}}_n$ the network output (depends on the weights w_{ij}!).

- SE_{train} is an "error landscape" over weight space.
- Weights are the model parameters q here.

Approach: gradient descent on SE_{train}, finding a local minimum of the error over weight space, thus a set of network weights that gives a (locally) minimal training error.

- requires, for every weight w_{ij}^m, an analytical expression for

$$\frac{\partial SE_{train}}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,\dots N} E(n)}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,\dots N} \left\| \mathbf{y}_n - \tilde{\mathbf{y}}_n \right\|^2}{\partial w_{ij}^m}$$

The backpropagation algorithm 1

Objective: find analytical expression for

$$\frac{\partial SE_{train}}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,\dots N} E(n)}{\partial w_{ij}^m} = \frac{\partial \sum_{n=1,\dots N} \left\| \mathbf{y}_n - \tilde{\mathbf{y}}_n \right\|^2}{\partial w_{ij}^m}$$

Difficulty: each $\partial SE_{train} / \partial w_{ij}^m$ depends on all training points and all (other) weights!

Solution: the famous backpropagation algorithm (Rumelhart, Hinton and Williams 1986; precursors / co-inventors: Werbos 1974, Le Cun 1986; history: Frasconi et al 1993).

- A scheme to iteratively compute the gradients $\partial SE_{train} / \partial w_{ij}^m$, starting from the output units for whose weights the gradient is easily obtained, working backwards through the network.
- For details see accompanying script or any NN textbook.
- Rather easy to implement and understand.
- Was **the** breakthrough for making neural networks useful for, and accepted by the engineering community.

The backpropagation algorithm 2: how to use it

- **Given:** training data $(\mathbf{x}_n, \mathbf{y}_n)_{n=1,\ldots,N}$, MLP with fixed structure but unknown weights.
- **Wanted:** set of weights that minimize training error.
- **Procedure:**
  1. Initialize network with random weights (usually some random small values)
  2. Repeat the following routine (an "epoch"):
     i. For each training input $\mathbf{x}_n$, compute all activations of the network's units.
     ii. Use these to compute the gradient $\partial SE_{train} / \partial w_{ij}^m$ for all weights (i.e., parameter set q), using the backpropagation algorithm.
     iii. Update the weights with our familiar gradient-descent formula

$$\theta_{new} = \theta_{old} - \eta \nabla SE_{train}(\theta_{old})$$

  3. Terminate when the error SE_{train} saturates, or when a fixed maximum number of iterations is reached.

The backpropagation algorithm 3: comments

- The "backpropagation algorithm" is actually not a complete learning algorithm, but "just" a method for the substep of computing the gradient within a gradient-descent learning algorithm.
- The generic difficulties of gradient descent algorithms strike: careful tuning of learning rate h, danger of slow convergence in the presence of different curvatures of error landscape, local minima only are found.
- A single gradient-descent-step (epoch) requires evaluation of the network on all training samples.
- Typical number of epochs: 50 to 1000.
- On large datasets becomes computationally expensive.
- A lot of experience and hand-tuning and trial-and-error is needed for very good results.
- Not biologically plausible
- The MLP structure (number of units, number of layers) has to be optimized heuristically.
- Typical network sizes in applications: about 10 to 30 hidden neurons.
- Many ready-made, free and commercial implementations

Multilayer perceptrons: universal approximation property

- Roughly speaking, every nonlinear function $f: \mathbb{R}^d \to \mathbb{R}^m$ can be approximated arbitrarily well by MLPs.
- A bit more precisely, for every error tolerance ε, and a compact subspace I of $\mathbb{R}^d$ (e.g., an d-dim interval), there exists some MLP that for inputs $\mathbf{x} \in I$ yields outputs that differ from $f(\mathbf{x})$ by less than ε.
- Naturally, the smaller the tolerance, the larger must the MLPs become.
- Theorems about universal approximation property are of little practical value because they are very generous about network size (large MLPs are prone to overfitting!), and because they ignore that we have only finite training data.
- It is more interesting to connect network size to the average square error R (risk) on new data. Not so many theorems. A classic for two-layer MLPs is due to Barron (1994):

$$R = O\left(\frac{C_f^2}{L^1}\right) + O\left(\frac{L^1 d}{N}\log(N)\right),$$

where C_f is a certain complexity measure for f, L^1 is number of hidden neurons, d is input dimension, and N is size of training sample.

Why should one use neural networks?

- There are many other methods for function approximation, e.g. Fourier decompositions, Volterra series,...
- What is the advantage of using ANNs? Why have they created such excitement?
- One reason is claimed biological plausibility. ANNs look smart...
- A harder reason lies in the following theorem, likewise due to Barron. When the number of hidden units is optimized for minimal generalization error, depending on the training sample size N, the average square error R on new data is

$$R = O(C_f((d/N)\log N)^{1/2})$$

- Thus the rate of risk convergence with optimally selected network sizes, as a function of sample size is of order $(d/N)^{1/2}$, where the exponent $1/2$ is independent on the input dimension d.
- In contrast, in Taylor, Fourier, Volterra expansion or other methods, where a fixed set of smooth basis functions is linearly combined to approximate the target function, one obtains a rate of convergence of $(1/N)^{2s/(2s+d)}$ (for some s). Here d occurs in the exponent by essentially $1/d$.
- Take-home conclusion: compared to other methods, whose performance degrades quickly with increasing dimension d, MLP function approximation is not exponentially sensitive to input dimension d. For high-dim input data, MLPs are thus superior.

MLPs: the workhorse and showcase ANN

MLPs (with a single hidden layer), trained by backprop, today ...

- are an established basic modelling technology,
- are used routinely when nonlinear functions have to be approximated from teaching data,
- are supported by a large number of software tools,
  - for Matlab users I recommend the "netlab" toolbox, www.ncrg.aston.ac.uk/netlab/ which is accompanied by a fully-fledged textbook;
- make up for (fully subjective personal estimate) 90% at least of all deployed ANNs,
- fail when the target functions become really complex.

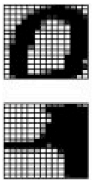
2.3 Convolutional networks: towards really complex function approximation

What does "really complex" mean?

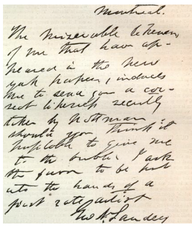
Mathematically (selection):
- highly varying; multiscale; heterogeneous; high-dimensional inputs

Technologically:
- Needs multi-module, multidisciplinary software architectures to be computed

Non-complex example (by today's standards): classify isolated digits 0 - 9 from preprocessed, standardized pixel images



Complex example: transcribe handwritten text (close to impossible today, even with large-scale integrated software architectures):



www.sonofthesouth.net

Veeery complex example: follow CNN video stream and generate an abstract library, semantically annotated

blog.eventful.com

Towards ANNs for "really complex" functions

A freshly emerging topic (Bengio and LeCun 2007)

No thorough theoretical insights yet

It seems clear that *hierarchical* or *multilayer* or *multimodular* model architectures are needed
- "Higher" processing levels deal with more abstract / slower / coarser aspects of the data
- Similar to what is known about processing levels in mammalian cortex
- Similar to multi-module "engineering solutions" for speech recognition systems

Multilayer perceptrons with larger number of layers (> 2!) don't train well with backprop, for apparently two reasons:
- The back-propagated error gradient has a tendency to vanish exponentially; that is, the layers close to the input train exceedingly slowly
- The problem of ending in poor local optima becomes more severe as the number of layers increases

Solution approach: don't use backprop (sessions 3 & 4)

Solution approach: use backprop, but exploit task specifics to avoid vanishing gradients (rest of this session)

LeCun's LeNets

LeNet-1, ..., LeNet-7: a family of really multi-layered feedforward networks specialized on visual pattern recognition

Check out http://yann.lecun.com/, http://www.cs.nyu.edu/~yann/research/norb/index.html, http://yann.lecun.com/exdb/lenet/index.html for a spectacular impression of state-of-the art machine learning (and of professional self-advertising...)

Starting paper: Le Cun et al 1998, comparison with other recognition methods on a very difficult dataset: Le Cun et al 2004

Design goals: classify patterns in a way that tolerates

- Noise
- Scaling, shifting, rotation
- Graphical variants

LeCun claims that about 10% of all cheques in the U.S. are read with software built on LeNets.

Very similar, earlier, classical approach: the Neocognitron (Fukushima 2007).

Basic rationale of LeNets

"Double pyramid of features" (Le Cun et al 1998):

- Close to input retina, have small number of types of low-level features, but large number of locations where these features are measured
- High-up in the processing cascade, have large number of high-level feature types but only few or a single instance of each.

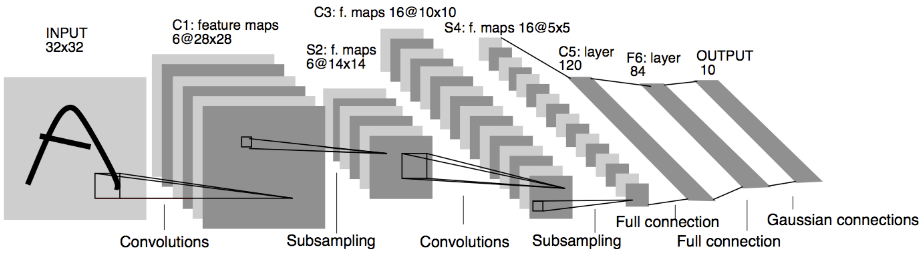
Low-level features: local graphical primitives, like "vertical line", "corner", ...

Shown: some occurences of feature "diagonal rising line"

Highest-level features: object concepts, like "elephant", "car", "person", ... many thousands more

Shown: singular occurrences of two top-level features "elephant" and "car"

LeNet architecture



7 layers (+ input layer); per layer as many "feature maps" as there are features

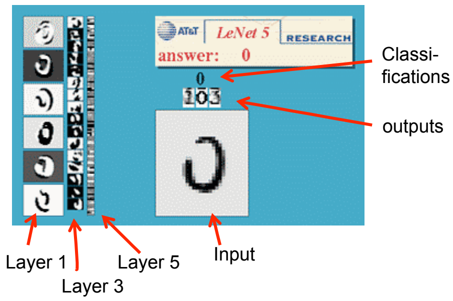
Altogether 360,000 connections and 60,000 trainable parameters

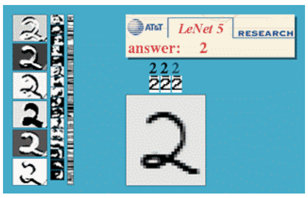
Trained by (refined) backprop

Used as primary processing layer in commercial cheque reading systems

See Le Cun et al (1998) for details

Performance examples 1: LeNet-5 for digit recognition





Performance examples 2: LeNet-7 for invariant object recognition in cluttered scenes

Training samples (about 200,000 of these): cluttered and jittered photographs of 50 plastic models of animals, humans, cars, planes and trucks (5 classes)

Test error on similar images: 16.7%.

Trained LeNet-7 generalizes to real-life images: