PV021: Neural networks

Tomáš Brázdil

Course organization

Course materials:

- **Main:** The lecture
- Neural Networks and Deep Learning by Michael Nielsen

  `http://neuralnetworksanddeeplearning.com/`

  (Extremely well written modern online textbook.)
- Deep learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville

  `http://www.deeplearningbook.org/`

  (A very good overview of the state-of-the-art in neural networks.)

Course organization

Evaluation:

- Project
  - teams of two students
  - implementation of a selected model + analysis of given data
  - implementation either in C, C++, or in Java **without use of any specialized libraries for data analysis and machine learning**
  - need to get over a given accuracy threshold (a gentle one, just to eliminate non-functional implementations)
- Oral exam
  - I may ask about anything from the lecture **including some proofs that occur only on the whiteboard!**
- Application of any deep learning toolset on given (difficult) data. We prefer TensorFlow but you may use another library (CNTK, Caffe, DeepLearning4j, ...) The goal is to get the best results on increasingly more difficult datasets.

  The team with the best result on the hardest dataset will automatically get $> F$ at the exam.

FAQ

Q: Why English?

A: Couple of reasons. First, all resources about modern neural nets are in English, it is rather cumbersome to translate everything to Czech (combination of Czech and English is ugly). Second, to attract non-Czech speaking students to the course.

Q: Why we cannot use specialized libraries in projects?

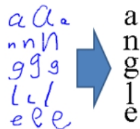
A: In order to "touch" the low level implementation details of the algorithms. You should not even use libraries for linear algebra and numerical methods, so that you will be confronted with rounding errors and numerical instabilities.

Machine learning in general

- Machine learning = construction of systems that may learn their functionality from data

  (... and thus do not need to be programmed.)

  - spam filter
    - learns to recognize spam from a database of "labelled" emails
    - consequently is able to distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labelled by their correct meaning
    - consequently is able to recognize text



  - . . .
  - and lots of much much more sophisticated applications ...

- Basic attributes of learning algorithms:
  - **representation**: ability to capture the inner structure of training data
  - **generalization**: ability to work properly on new data

Machine learning in general

Machine learning algorithms typically construct mathematical models of given data. The models may be subsequently applied to fresh data.

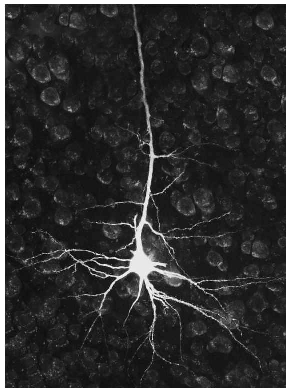
There are many types of models:

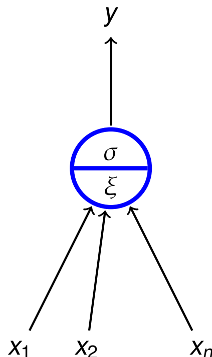
- decision trees
- support vector machines
- hidden Markov models
- Bayes networks and other graphical models
- **neural networks**
- ...

Neural networks, based on models of a (human) brain, form a natural basis for learning algorithms!

Artificial neural networks

- **Artificial neuron** is a *rough mathematical approximation* of a biological neuron.
- **(Aritificial) neural network (NN)** consists of a number of interconnected artificial neurons. "Behavior" of the network is encoded in connections between neurons.





Zdroj obrázku: `http://tulane.edu/sse/cmb/people/schrader/`

Why artificial neural networks?

Modelling of biological neural networks (computational neuroscience).

- simplified mathematical models help to identify important mechanisms
  - How a brain receives information?
  - How the information is stored?
  - How a brain develops?
  - ...
- neuroscience is strongly multidisciplinary; precise mathematical descriptions help in communication among experts and in design of new experiments.

I will not spend much time on this area!

Why artificial neural networks?

Neural networks in machine learning.

- Typically primitive models, far from their biological counterparts (but often inspired by biology).
- Strongly oriented towards concrete application domains:
  - decision making and control - autonomous vehicles, manufacturing processes, control of natural resources
  - games - backgammon, poker, GO
  - finance - stock prices, risk analysis
  - medicine - diagnosis, signal processing (EKG, EEG, ...), image processing (MRI, roentgen, ...)
  - text and speech processing - automatic translation, text generation, speech recognition
  - other signal processing - filtering, radar tracking, noise reduction
  - ...

I will concentrate on this area!

Important features of neural networks

- Massive parallelism
  - many slow (and "dumb") computational elements work in parallel on several levels of abstraction
- Learning
  - a kid learns to recognize a rabbit after seeing several rabbits
- Generalization
  - a kid is able to recognize a new rabbit after seeing several (old) rabbits
- Robustness
  - a blurred photo of a rabbit may still be classified as a picture of a rabbit
- Graceful degradation
  - Experiments have shown that damaged neural network is still able to work quite well
  - Damaged network may re-adapt, remaining neurons may take on functionality of the damaged ones

The aim of the course

- We will concentrate on
  - basic techniques and principles of neural networks,
  - fundamental models of neural networks and their applications.
- You should learn
  - basic models
    (multilayer perceptron, convolutional networks, recurent network (LSTM), Hopfield and Boltzmann machines and their use in pre-training of deep nets)
  - Standard applications of these models
    (image processing, speech and text processing)
  - Basic learning algorithms
    (gradient descent & backpropagation, Hebb's rule)
  - Basic practical training techniques
    (data preparation, setting various parameters, control of learning)
  - Basic information about current implementations
    (TensorFlow, CNTK)

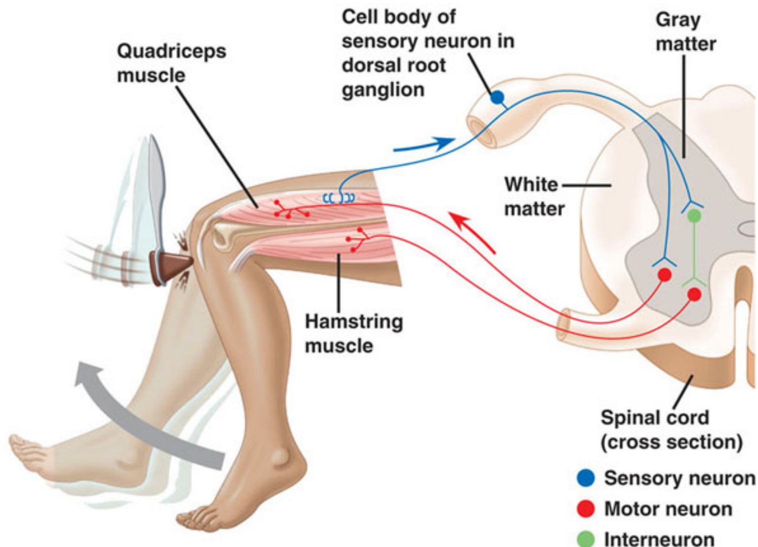
Biological neural network

- Human neural network consists of approximately 10^{11} (100 billion on the short scale) neurons; a single cubic centimeter of a human brain contains almost 50 million neurons.
- Each neuron is connected with approx. 10^4 neurons.
- Neurons themselves are very complex systems.

Rough description of nervous system:

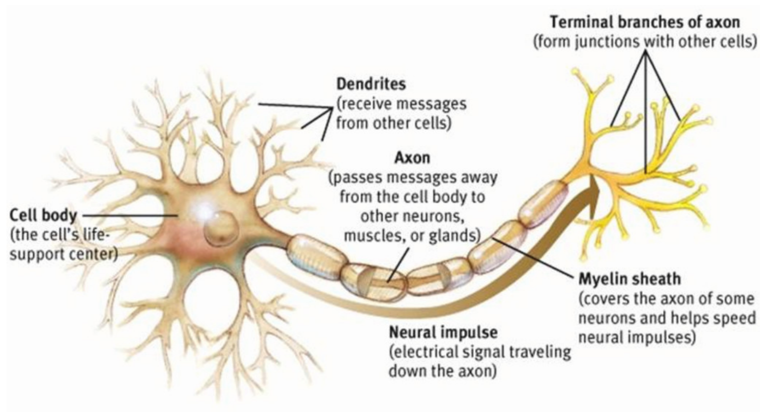
- External stimulus is received by *sensory receptors* (e.g. eye cells).
- Information is futher transfered via peripheral nervous system (PNS) to the central nervous systems (CNS) where it is processed (integrated), and subseqently, an output signal is produced.
- Afterwards, the output signal is transfered via PNS to *effectors* (e.g. muscle cells).

Biological neural network



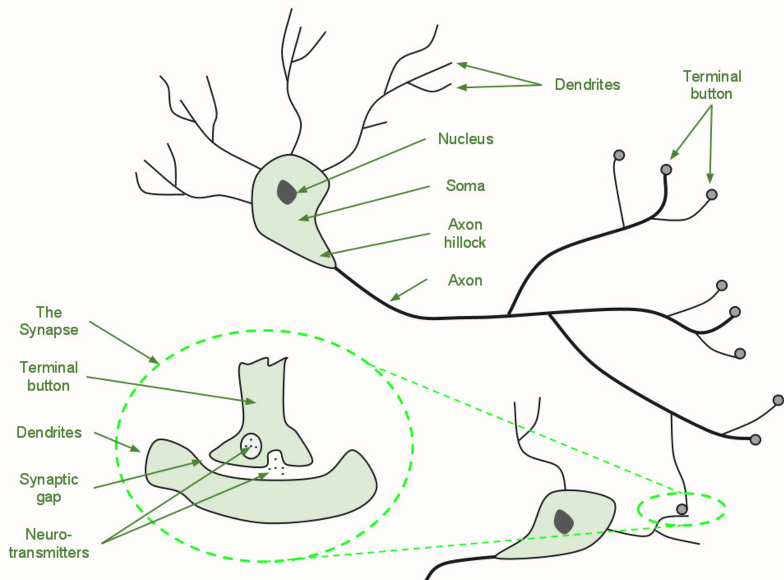
Zdroj: N. Campbell and J. Reece; Biology, 7th Edition; ISBN: 080537146X

Biological neuron

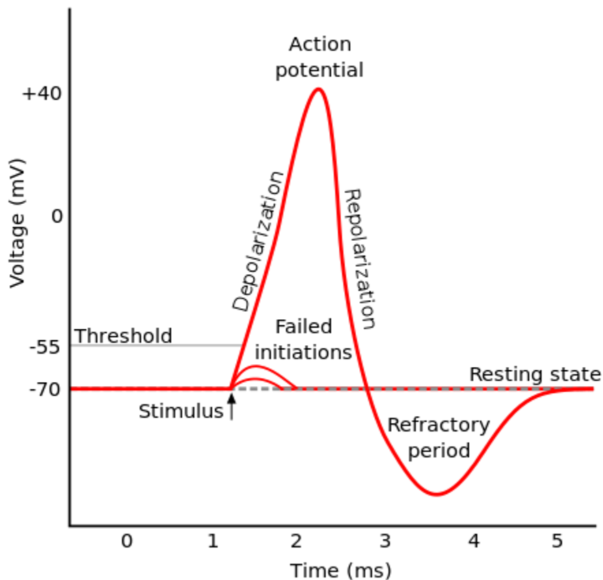


Zdroj: `http://www.web-books.com/eLibrary/Medicine/Physiology/Nervous/Nervous.htm`

Synaptic connections



Action potential



Summation

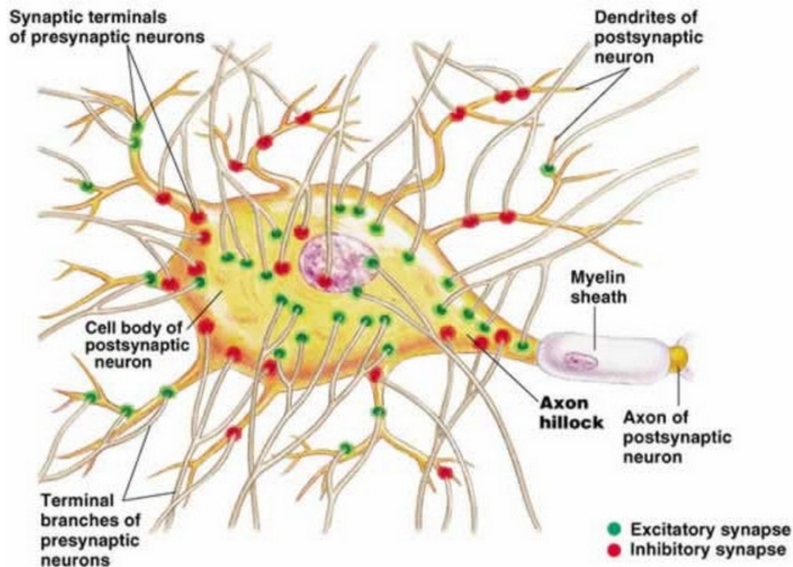
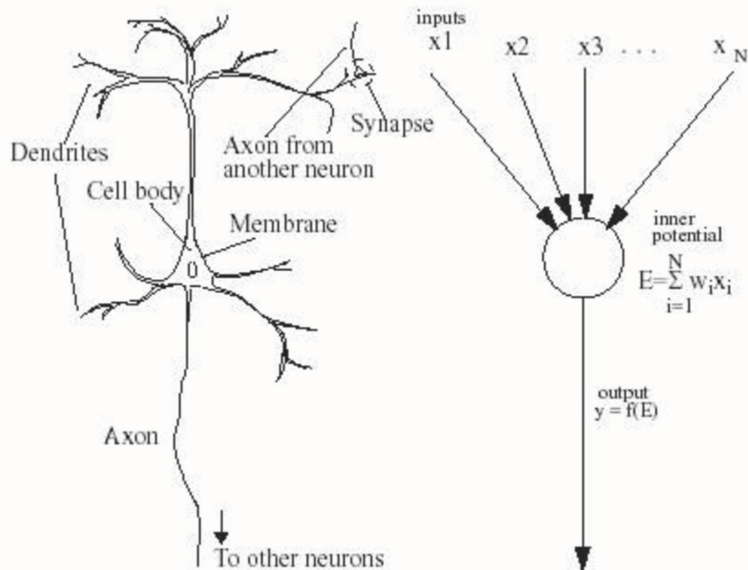
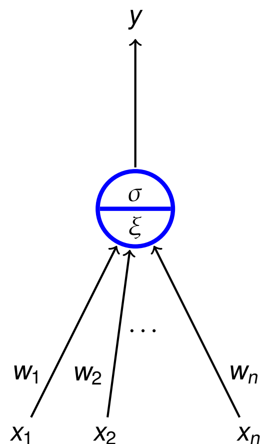


Figure 48.11(a), page 972, Campbell's *Biology, 5th Edition*

Biological and Mathematical neurons



Formal neuron (without bias)

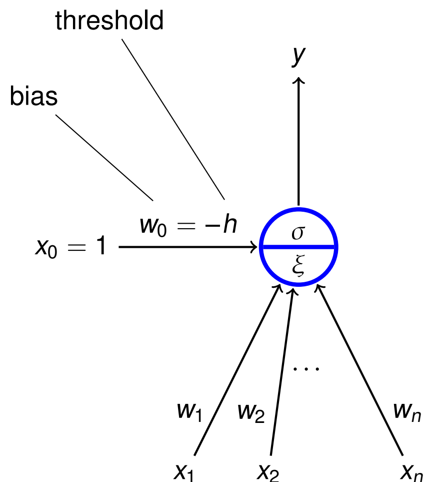


- $x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$
  where σ is an **activation function**;
  e.g. a *unit step function*

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq h\,; \\ 0 & \xi < h. \end{cases}$$

where $h \in \mathbb{R}$ is a *threshold*.

Formal neuron (with bias)



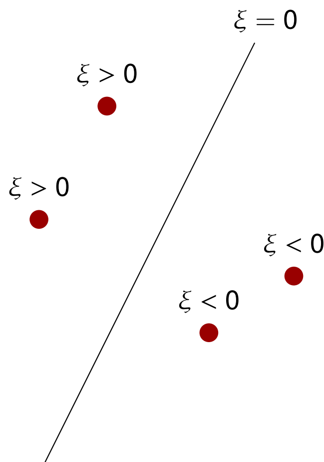
- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$ where σ is an **activation function**;

  e.g. a *unit step function*

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

(The threshold h has been substituted with the new input $x_0 = 1$ and the weight $w_0 = -h$.)

Neuron and linear separation

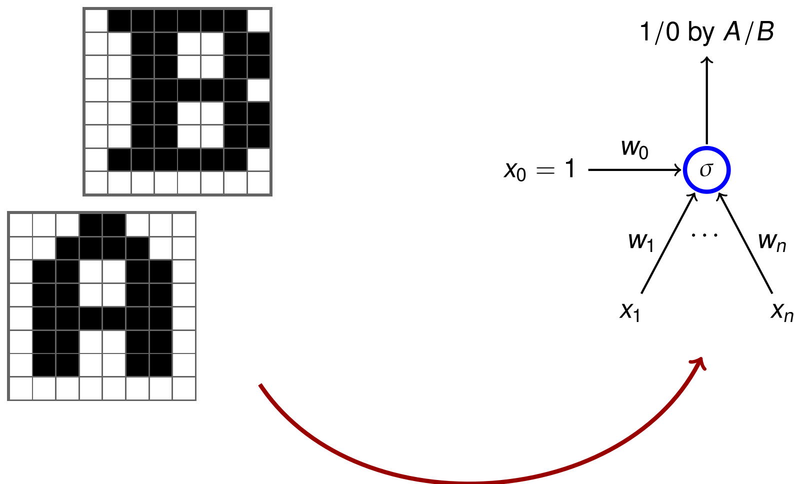


- inner potential

$$\xi = w_0 + \sum_{i=1}^{n} w_i x_i$$

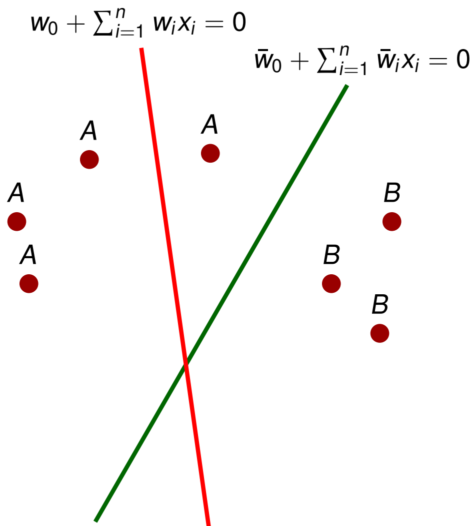
  determines a separation hyperplane in the n-dimensional **input space**
  - in 2d line
  - in 3d plane
  - $\cdots$

Neuron and linear separation



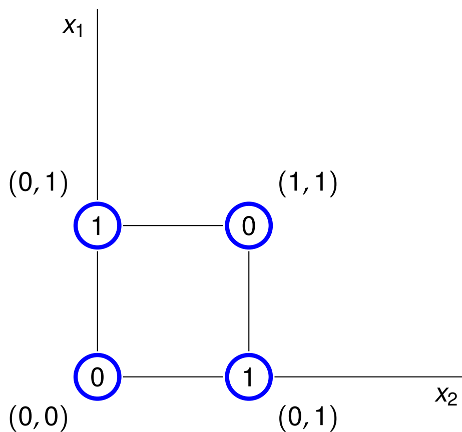
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

Neuron and linear separation



- Red line classifies incorrectly
- Green line classifies correctly (may be a result of a correction by a learning algorithm)

Neuron and linear separation (XOR)



- No line separates ones from zeros.

Neural networks

Neural network consists of formal neurons interconnected in such a way that the output of one neuron is an input of several other neurons.

In order to describe a particular type of neural networks we need to specify:

- Architecture

  How the neurons are connected.
- Activity

  How the network transforms inputs to outputs.
- Learning

  How the weights are changed during training.

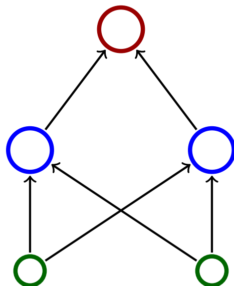
Architecture

Network architecture is given as a digraph whose nodes are neurons and edges are connections.

We distinguish several categories of neurons:

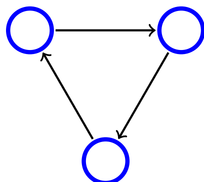
- Output neurons
- Hidden neurons
- Input neurons

(In general, a neuron may be both input and output; a neuron is hidden if it is neither input, nor output.)

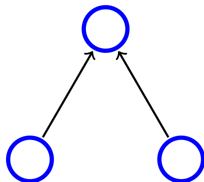


Architecture – Cycles

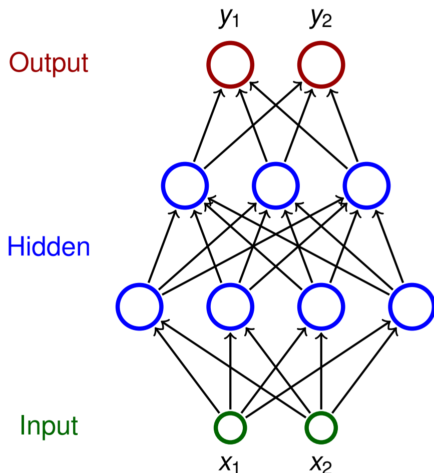
- A network is **cyclic** (recurrent) if its architecture contains a directed cycle.



- Otherwise it is **acyclic** (feed-forward)



Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i+1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

Activity

Consider a network with n neurons, k input and ℓ output.

- **State** of a network is a vector of output values of all neurons.

  (States of a network with n neurons are vectors of $\mathbb{R}^n$)

- **State-space** of a network is a set of all states.

- **Network input** is a vector of k real numbers, i.e. an element of $\mathbb{R}^k$.
- **Network input space** is a set of all network inputs.

  (sometimes we restrict ourselves to a proper subset of $\mathbb{R}^k$)

- **Initial state**

  Input neurons set to values from the network input
  (each component of the network input corresponds to an input neuron)

  Values of the remaining neurons set to 0.

Activity – computation of a network

- **Computation** (typically) proceeds in discrete steps.
  In every step the following happens:
  1. A set of neurons is selected according to some rule.
  2. The selected neurons change their states according to their inputs (they are simply evaluated).
     (If a neuron does not have any inputs, its value remains constant.)

  A computation is **finite** on a network input $\vec{x}$ if the state changes only finitely many times (i.e. there is a moment in time after which the state of the network never changes).
  We also say that the network **stops on** $\vec{x}$.

- **Network output** is a vector of values of all output neurons in the network (i.e. an element of $\mathbb{R}^{\ell}$).
  Note that the network output keeps changing throughout the computation!

MLP uses the following selection rule:

In the i-th step evaluate all neurons in the i-th layer.

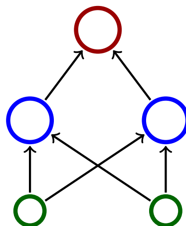
Activity – semantics of a network

Definition

Consider a network with n neurons, k input, ℓ output. Let $A \subseteq \mathbb{R}^k$ and $B \subseteq \mathbb{R}^\ell$. Suppose that the network stops on every input of A.
Then we say that the network computes a function $F : A \to B$ *if for every network input $\vec{x}$ the vector $F(\vec{x}) \in B$ is the output of the network after the computation on $\vec{x}$ stops.*

Example 1

This network computes a function from $\mathbb{R}^2$ to $\mathbb{R}$.



Activity – inner potential and activation functions

In order to specify activity of the network, we need to specify how the inner potentials ξ are computed and what are the activation functions σ.

We assume (unless otherwise specified) that

$$\xi = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

here $\vec{x} = (x_1, \ldots, x_n)$ are inputs of the neuron and $\vec{w} = (w_1, \ldots, w_n)$ are weights.

There are special types of neural network where the inner potential is computed differently, e.g. as a "distance" of an input from the weight vector:

$$\xi = \left\| \vec{x} - \vec{w} \right\|$$

here $\|\cdot\|$ is a vector norm, typically Euclidean.

Activity – inner potential and activation functions

There are many activation functions, typical examples:

- Unit step function

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

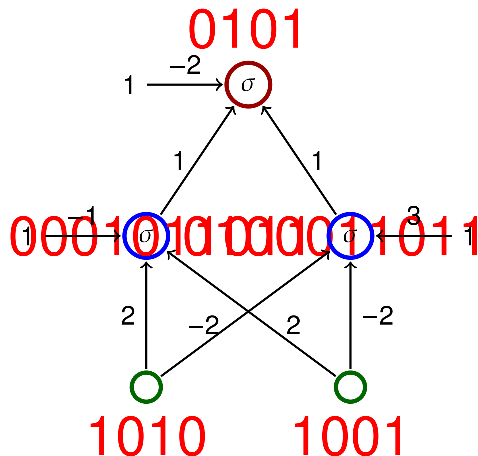
- (Logistic) sigmoid

$$\sigma(\xi) = \frac{1}{1 + e^{-\lambda \cdot \xi}}$$ here $\lambda \in \mathbb{R}$ is a *steepness* parameter.

- Hyperbolic tangens

$$\sigma(\xi) = \frac{1 - e^{-\xi}}{1 + e^{-\xi}}$$

Activity – XOR



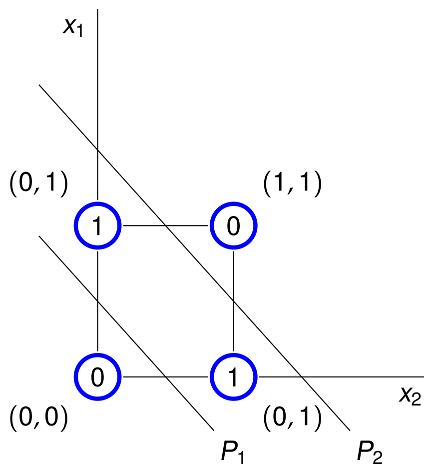
- Activation function is a unit step function

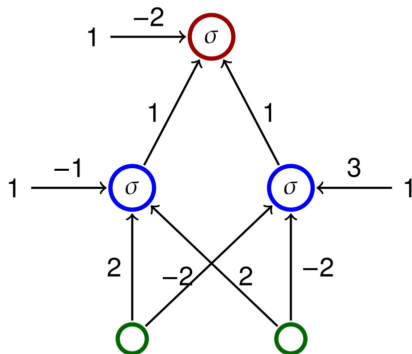
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

- The network computes $XOR(x_1, x_2)$

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

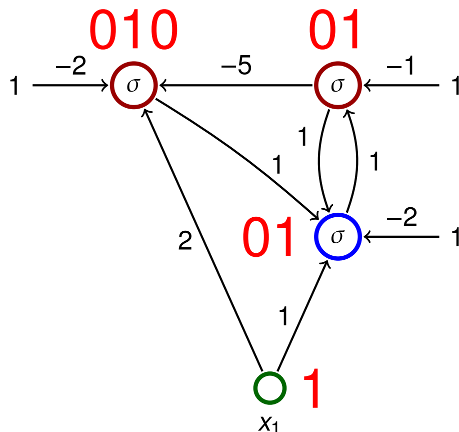
Activity – MLP and linear separation





- The line P_1 is given by $-1 + 2x_1 + 2x_2 = 0$
- The line P_2 is given by $3 - 2x_1 - 2x_2 = 0$

Activity – example



The activation function is the unit step function

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

The input is equal to 1

Learning

Consider a network with n neurons, k input and ℓ output.

- **Configuration** of a network is a vector of all values of weights.

  (Configurations of a network with m connections are elements of $\mathbb{R}^m$)
- **Weight-space** of a network is a set of all configurations.
- **initial configuration**

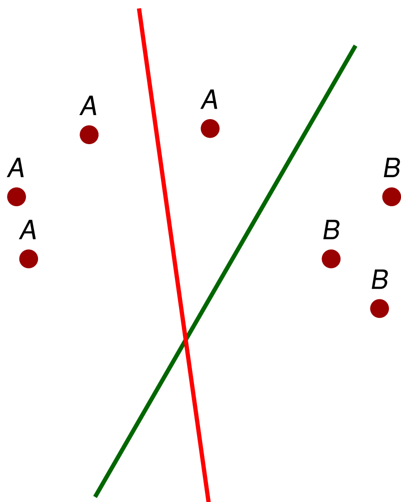
  weights can be initialized randomly or using some sophisticated algorithm

Learning algorithms

Learning rule for weight adaptation.
(the goal is to find a configuration in which the network computes a desired function)

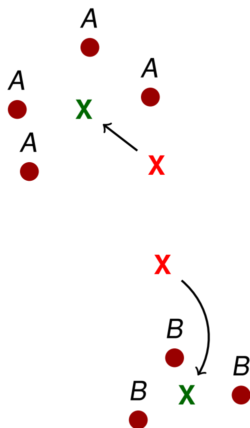
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.
- Unsupervised learning
  - The training set contains only inputs.
  - The goal is to determine distribution of the inputs (clustering, deep belief networks, etc.)

Supervised learning – illustration



- classification in the plane using a single neuron
- training examples are of the form (point, value) where the value is either 1, or 0 depending on whether the point is either A, or B
- the algorithm considers examples one after another
- whenever an incorrectly classified point is considered, the learning algorithm turns the line in the direction of the point

Unsupervised learning – illustration

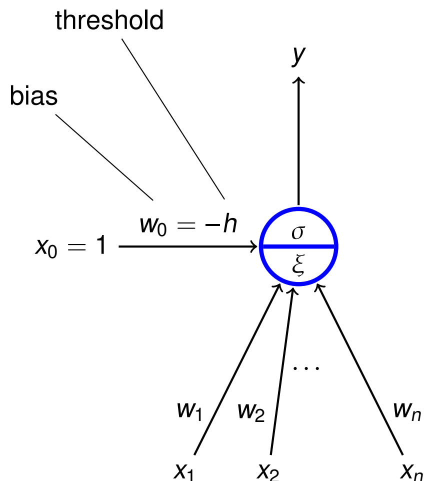


- we search for two *centres* of clusters
- red crosses correspond to potential centres before application of the learning algorithm, green ones after the application

Summary – Advantages of neural networks

- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks
- generalization and robustness
  - information is encoded in a distributed manned in weights
  - "close" inputs typicaly get similar values
- Graceful degradation
  - damage typically causes only a decrease in precision of results

Formal neuron (with bias)



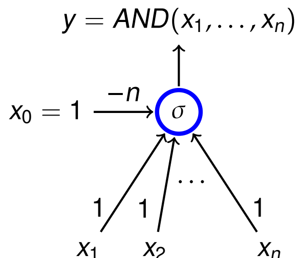
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$ where σ is an **activation function**;

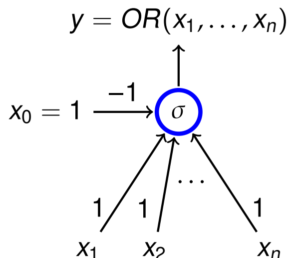
  e.g. a *unit step function*

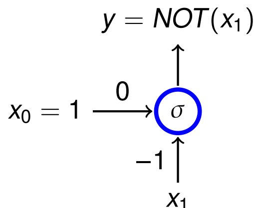
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Boolean functions

Activation function: *unit step function* $\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$







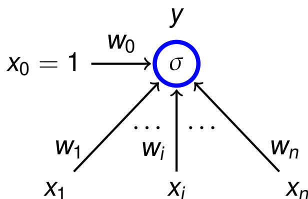
Boolean functions

Theorem

Let σ be the unit step function. Two layer MLPs, where each neuron has σ as the activation function, are able to compute all functions of the form $F : \{0, 1\}^n \rightarrow \{0, 1\}$.

Proof.

- Given a vector $\vec{v} = (v_1, \ldots, v_n) \in \{0, 1\}^n$, consider a neuron $N_{\vec{v}}$ whose output is 1 iff the input is $\vec{v}$:

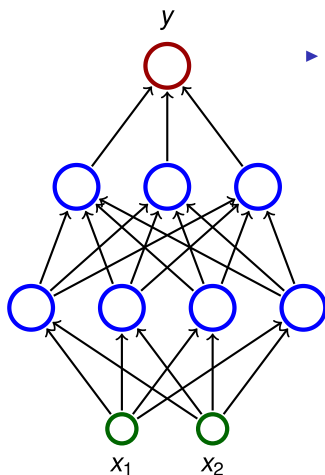


$$w_0 = -\sum_{i=1}^{n} v_i$$

$$w_i = \begin{cases} 1 & v_i = 1 \\ -1 & v_i = 0 \end{cases}$$

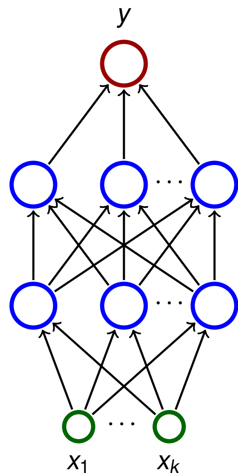
- Now let us connect all outputs of all neurons $N_{\vec{v}}$ satisfying $F(\vec{v}) = 1$ using a neuron implementing OR. □

Non-linear separation



- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the half-spaces ⇒ convex sets.
  - The third layer may e.g. make unions of some convex sets.

Non-linear separation – illustration



- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)
  - Each hypercube K can be separated using a two layer network N_K
    (i.e. a function computed by N_K gives 1 for points in K and 0 for the rest).
  - Finally, connect outputs of the nets N_K satisfying $K \cap A \neq \emptyset$ using a neuron implementing OR.

Non-linear separation - sigmoid

Theorem (Cybenko 1989 - informal version)

Let σ be a continuous function which is sigmoidal, i.e. satisfies

$$\sigma(x) = \begin{cases} 1 & \text{pro } x \to +\infty \\ 0 & \text{pro } x \to -\infty \end{cases}$$

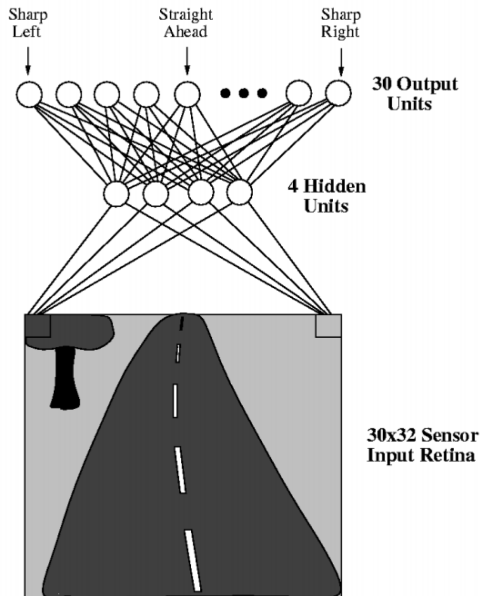
For every reasonable set $A \subseteq [0,1]^n$, there is a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following:*

For most vectors $\vec{v} \in [0,1]^n$ we have that $\vec{v} \in A$ iff the network output is > 0 for the input $\vec{v}$.

For mathematically oriented:

- "reasonable" means Lebesgue measurable
- "most" means that the set of incorrectly classified vectors has the Lebesgue measure smaller than a given $\varepsilon > 0$

Non-linear separation - practical illustration



- ALVINN drives a car
- The net has $30 \times 32 = 960$ inputs (the input space is thus $\mathbb{R}^{960}$)
- Input values correspond to shades of gray of pixels.
- Output neurons "classify" images of the road based on their "curvature".

Zdroj obrázku: `http://jmvidal.cse.sc.edu/talks/ann/alvin.html`

Function approximation - three layers

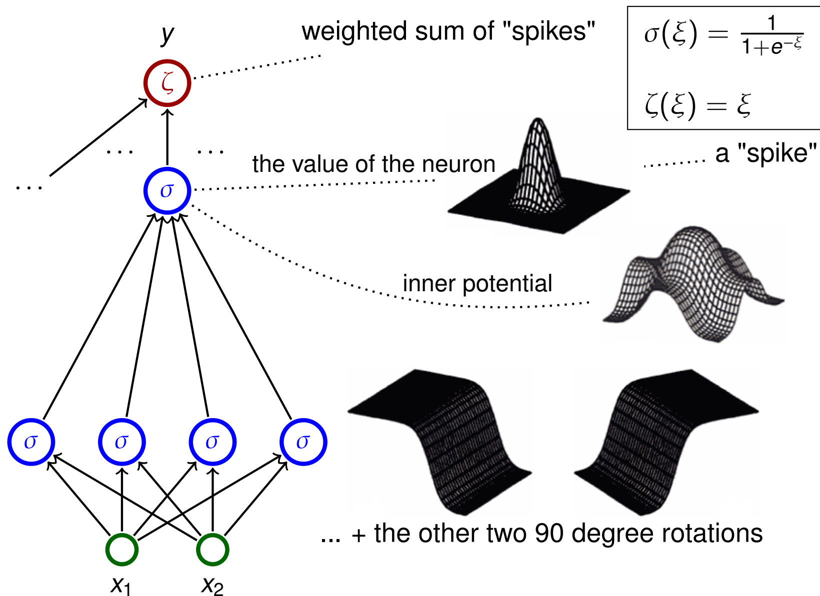
Let σ be a logistic sigmoid, i.e.

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}$$

For every continuous function $f : [0,1]^n \rightarrow [0,1]$ and $\varepsilon > 0$ there is a three-layer network computing a function $F : [0,1]^n \rightarrow [0,1]$ such that

- there is a linear activation in the output layer, i.e. the value of the output neuron is its inner potential ξ,
- the remaining neurons have the logistic sigmoid σ as their activation,
- for every $\vec{v} \in [0,1]^n$ we have that $|F(\vec{v}) - f(\vec{v})| < \varepsilon$.

Function approximation – three layer networks



Function approximation - two-layer networks

Theorem (Cybenko 1989)

Let σ be a continuous function which is sigmoidal, i.e. is increasing and satisfies

$$\sigma(x) = \begin{cases} 1 & \textit{pro } x \to +\infty \\ 0 & \textit{pro } x \to -\infty \end{cases}$$

For every continuous function $f : [0,1]^n \to [0,1]$ and every $\varepsilon > 0$ there is a function $F : [0,1]^n \to [0,1]$ computed by a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following*

$$|f(\vec{v}) - F(\vec{v})| < \varepsilon \qquad \textit{pro každé } \vec{v} \in [0,1]^n.$$

Neural networks and computability

- Consider recurrent networks (i.e. containing cycles)
  - with real weights (in general);
  - one input neuron and one output neuron (the network computes a function $F : A \to \mathbb{R}$ where $A \subseteq \mathbb{R}$ contains all inputs on which the network stops);
  - parallel activity rule (output values of all neurons are recomputed in every step);
  - activation function

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ \xi & 0 \leq \xi \leq 1\,; \\ 0 & \xi < 0. \end{cases}$$

- We encode words $\omega \in \{0,1\}^+$ into numbers as follows:

$$\delta(\omega) = \sum_{i=1}^{|\omega|} \frac{\omega(i)}{2^i} + \frac{1}{2^{|\omega|+1}}$$

E.g. $\omega = 11001$ gives $\delta(\omega) = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^5} + \frac{1}{2^6}$ ($= 0.110011$ in binary form).

Neural networks and computability

A network **recognizes** a language $L \subseteq \{0,1\}^+$ if it computes a function $F : A \to \mathbb{R}$ ($A \subseteq \mathbb{R}$) such that

$$\omega \in L \text{ iff } \delta(\omega) \in A \text{ and } F(\delta(\omega)) > 0.$$

- Recurrent networks with rational weights are equivalent to Turing machines
  - For every recursively enumerable language $L \subseteq \{0,1\}^+$ there is a recurrent network with rational weights and less than 1000 neurons, which recognizes L.
  - The halting problem is undecidable for networks with at least 25 neurons and rational weights.
  - There is "universal" network (equivalent of the universal Turing machine)
- Recurrent networks are super-Turing powerful
  - For **every** language $L \subseteq \{0,1\}^+$ there is a recurrent network with less than 1000 nerons which recognizes L.

Summary of theoretical results

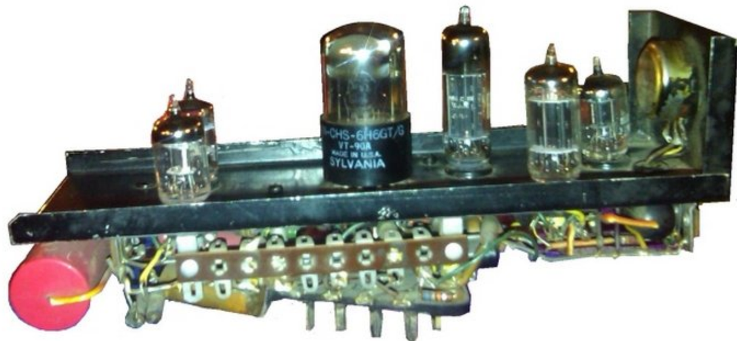
- Neural networks are very strong from the point of view of theory:
  - All Boolean functions can be expressed using two-layer networks.
  - Two-layer networks may approximate any continuous function.
  - Recurrent networks are at least as strong as Turing machines.
- These results are purely theoretical!
  - "Theoretical" networks are extremely huge.
  - It is very difficult to handcraft them even for simplest problems.
- From practical point of view, the most important advantage of neural networks are: learning, generalization, robustness.

Neural networks vs classical computers

	Neural networks	Classical computers
Data	implicitly in weights	explicitly
Computation	naturally parallel	sequential, localized
Robustness	robust w.r.t. input corruption & damage	changing one bit may completely crash the computation
Precision	imprecise, network recalls a training example "similar" to the input	(typically) precise
Programming	learning	manual

History of neurocomputers

- 1951: SNARC (Minski et al)
  - the first implementation of neural network
  - a rat strives to exit a maze
  - 40 artificial neurons (300 vacuum tubes, engines, etc.)



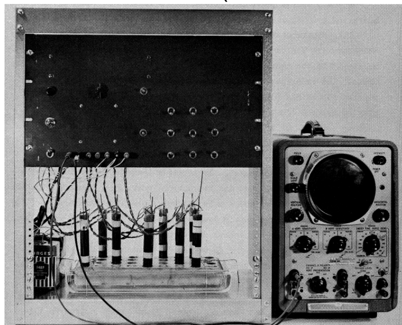
History of neurocomputers

- 1957: Mark I Perceptron (Rosenblatt et al) - the first successful network for image recognition

  - single layer network
  - image represented by 20×20 photocells
  - intensity of pixels was treated as the input to a perceptron (basically the formal neuron), which recognized figures
  - weights were implemented using potentiometers, each set by its own engine
  - it was possible to arbitrarily reconnect inputs to neurons to demonstrate adaptability

History of neurocomputers

- 1960: ADALINE (Widrow & Hof)



  - single layer neural network
  - weights stored in a newly invented electronic component **memistor**, which remembers history of electric current in the form of resistance.
  - Widrow founded a company Memistor Corporation, which sold implementations of neural networks.
- 1960-66: several companies concerned with neural networks were founded.

History of neurocomputers

- 1967-82: dead still after publication of a book by Minski & Papert (published 1969, title *Perceptrons*)
- 1983-end of 90s: revival of neural networks
  - many attempts at hardware implementations
    - application specific chips (ASIC)
    - programmable hardware (FPGA)
  - hw implementations typically not better than "software" implementations on universal computers (problems with weight storage, size, speed, cost of production etc.)
- end of 90s-cca 2005: NN suppressed by other machine learning methods (support vector machines (SVM))
- 2006-now: The boom of neural networks!
  - deep networks – often better than any other method
  - GPU implementations
  - ... some specialized hw implementations (Google's TPU)

History in waves ...

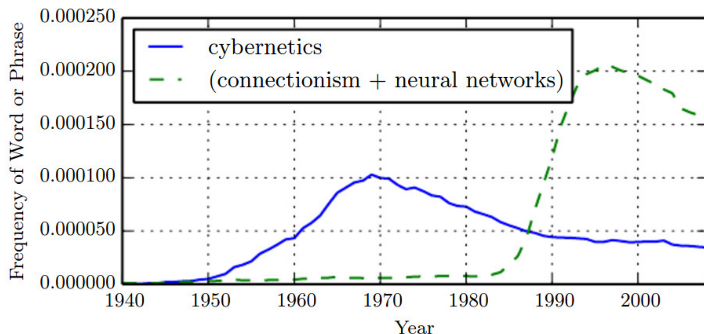
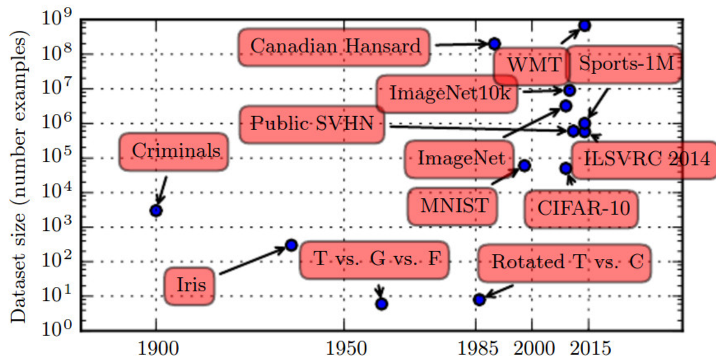


Figure: The figure shows two of the three historical waves of articial neural nets research, as measured by the frequency of the phrases “cybernetics” and “connectionism” or “neural networks” according to Google Books (the third wave is too recent to appear).

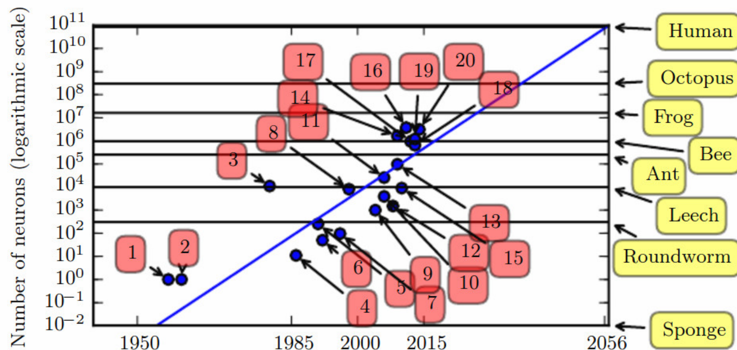
Current hardware – What do we face?

Increasing dataset size ...



Current hardware – What do we face?

... and thus increasing size of neural networks ...



2. ADALINE
4. Early back-propagation network (Rumelhart et al., 1986b)
8. Image recognition: LeNet-5 (LeCun et al., 1998b)
10. Dimensionality reduction: Deep belief network (Hinton et al., 2006)
    ... here the third "wave" of neural networks started
15. Digit recognition: GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
18. Image recognition (AlexNet): Multi-GPU convolutional network (Krizhevsky et al., 2012)
20. Image recognition: GoogLeNet (Szegedy et al., 2014a)

Current hardware – What do we face?

... as a reward we get this ...

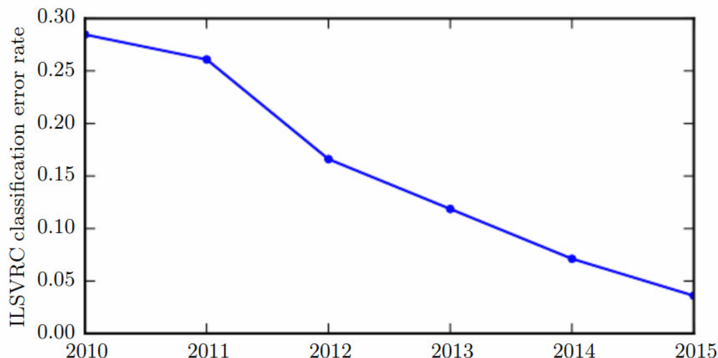


Figure: Since deep networks reached the scale necessary to compete in the ImageNetLarge Scale Visual Recognition Challenge, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky et al. (2014b) and He et al. (2015).

Current hardware

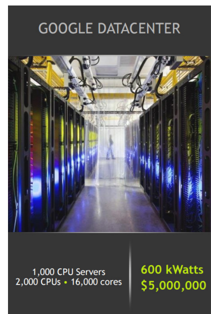
In 2012, Google trained a large network of 1.7 billion weights and 9 layers

The task was image recognition (10 million youtube video frames)

The hw comprised a 1000 computer network (16 000 cores), computation took three days.

In 2014, similar task performed on Commodity Off-The-Shelf High Performance Computing (COTS HPC) technology: a cluster of GPU servers with Infiniband interconnects and MPI.

Able to train 1 billion parameter networks on just 3 machines in a couple of days.
Able to scale to 11 billion weights (approx. 6.5 times larger than the Google model) on 16 GPUs.



Current hardware – NVIDIA DGX Station

- 4x GPU (Tesla V100)
- TFLOPS = 480
- GPU memory 64GB total
- NVIDIA Tensor Cores: 2,560
- NVIDIA CUDA Cores: 20,480
- System memory: 256 GB
- Network: Dual 10 Gb LAN
- NVIDIA Deep Learning SDK

Current software

- **TensorFlow** (Google)
  - open source software library for numerical computation using data flow graphs
  - allows implementation of most current neural networks
  - allows computation on multiple devices (CPUs, GPUs, ...)
  - Python API
  - **Keras**: a library on top of TensorFlow that allows easy description of most modern neural networks
- **CNTK** (Microsoft)
  - functionality similar to TensorFlow
  - special input language called BrainScript
- **Theano**:
  - The "academic" grand-daddy of deep-learning frameworks, written in Python. Strongly inspired TensorFlow (some people developing Theano moved on to develop TensorFlow).
- There are others: Caffe, Torch (Facebook), Deeplearning4j, ...

Current software – Keras

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape
# here, 20-dimensional vectors.
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
          batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

Other software implementations

Most "mathematical" software packages contain some support of neural networks:

- MATLAB
- R
- STATISTICA
- Weka
- ...

The implementations are typically not on par with the previously mentioned dedicated deep-learning libraries.

SyNAPSE (USA)

- Big research project, partially funded by DARPA
- Among the main subjects IBM a HRL, collaboraton with top US universities, e.g. Boston, Stanford
- The project started in 2008
- Invested tens of millions USD.

The goal

- Develop a neural network comparable with a real brain of a mammal
- The resulting hw chip should simulate 10 billion neurons, 100 trillion synaptic connections, consume 1 kilowatt ($\sim$ a small heater), size 2 dm^3
- Oriented towards development of a new parallel computer architecture rather than neuroscience.

SyNAPSE (USA) – some results

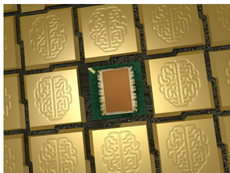
A cat brain simulation (2009)

- A simulation of a network with 10^9 neurons, 10^{13} synapses
- Simulated on a supercomputer Dawn (Blue Gene/P), 147,450 CPU, 144 tB of memory
- 643 times slower than the real brain
- The network modelled according to the real-brain structure (hierarchical model of a visual cortex, 4 layers)
- The authors claim that they observed some behaviour similar to the behaviour of the real brain (signal propagation, α, γ waves)

... simulation was heavily criticised (see latter)

... in 2012 the number of neurons increased to 530 bn neurons a 100 tn synapses

SyNAPSE (USA) – TrueNorth



- A chip with 5.4 billion elements
- 4096 neurosynaptic cores connected by a network, implementing 1 million programmable "spiking" neurons, 256 million programmable synaptic connections
- global frequency 1-kHz
- low energy consumption, approx. 63mW

- Offline learning, implemented some known algorithms (convolutional networks, RBM etc.)
- Applied to simple image recognition tasks.

Human Brain Project, HBP (Europe)

- Funded by EU, budget 10^9 EUR for 10 years
- Successor of Blue Brain Project at EPFL Lausanne
- Blue Brain started in 2005, ended in 2012, Human Brain Project started in 2013

The original goal: Deeper understanding of human brain

- **networking in neuroscience**
- diagnosis of brain diseases
- thinking machine

The approach:

- study of brain tissue using current technology
- modelling of biological neurons
- simulation of the models (program NEURON)

HBP (Europe

Blue brain project (2008)

- Model of a part of the brain cortex of a rat (approx. 10,000 neurons), much more complex model of neurons than in SyNAPSE
- Simulated on a supercomputer of the type Blue Gene/P (provided by IBM on discount), 16,384 CPU, 56 teraflops, 16 terabytů paměti, 1 PB disk space
- Simulation 300x slower than the real brain

Human brain project (2015):

- Simplified model of the nervous system of a rat (approx. 200 000 neurons)

SyNAPSE vs HBP

2011: IBM Simulates 4.5 percent of the Human Brain, and All of the Cat Brain (Scientific American)

"... performed the first near real-time cortical simulation of the brain that exceeds the scale of a cat cortex" (IBM)

This announcement has been heavily criticised by Dr. Markram (head of HBP)

"This is a mega public relations stunt – a clear case of scientific deception of the public"

"Their so called "neurons" are the tiniest of points you can imagine, a microscopic dot"

"Neurons contain 10's of thousands of proteins that form a network with 10's of millions of interactions. These interactions are incredibly complex and will require solving millions of differential equations. They have none of that."

SyNAPSE vs HBP

"Eugene Izhikevik himself already in 2005 ran a simulation with 100 billion such points interacting just for the fun of it: (over 60 times larger than Modha's simulation)"

Why did they get the Gordon Bell Prize?
"They seem to have been very successful in influencing the committee with their claim, which technically is not peer-reviewed by the respective community and is neuroscientifically outrageous."

But is there any innovation here?
"The only innovation here is that IBM has built a large supercomputer."

But did Mohda not collaborate with neuroscientists?
"I would be very surprised if any neuroscientists that he may have had in his DARPA consortium realized he was going to make such an outrageous claim. I can't imagine that the San Fransisco neuroscientists knew he was going to make such a stupid claim. Modha himself is a software engineer with no knowledge of the brain."

... and in the meantime in Europe

In 2014, the European Commission received an open letter signed by more than 130 heads of laboratories demanding a substantial change in the management of the whole project.

Peter Dayan, director of the computational neuroscience unit at UCL:

“The main apparent goal of building the capacity to construct a larger-scale simulation of the human brain is radically premature.”

“We are left with a project that can’t but fail from a scientific perspective. It is a waste of money, it will suck out funds from valuable neuroscience research, and would leave the public, who fund this work, justifiably upset.”

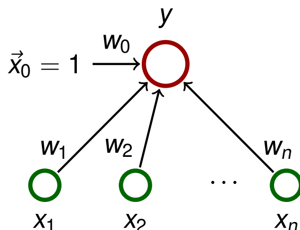
... and in 2016

The European Commission and the Human Brain Project Coordinator, the École Polytechnique Fédérale de Lausanne (EPFL), have signed the first Specific Grant Agreement (SGA1), releasing EUR 89 million in funding retroactively from 1st April 2016 until the end of March 2018. The signature of SGA1 means that the HBP and the European Commision have agreed on the HBP Work Plan for this two year period.

The SGA1 work plan will move the Project closer to achieving its aim of establishing a cutting-edge, ICT-based scientific Research Infrastructure for brain research, cognitive neuroscience and brain-inspired computing.

ADALINE

Architecture:



$\vec{w} = (w_0, w_1, \dots, w_n)$ and $\vec{x} = (x_0, x_1, \dots, x_n)$ where $x_0 = 1$.

Activity:

- inner potential: $\xi = w_0 + \sum_{i=1}^{n} w_i x_i = \sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
- activation function: $\sigma(\xi) = \xi$
- network function: $y[\vec{w}](\vec{x}) = \sigma(\xi) = \vec{w} \cdot \vec{x}$

ADALINE

Learning:

- Given a **training set**

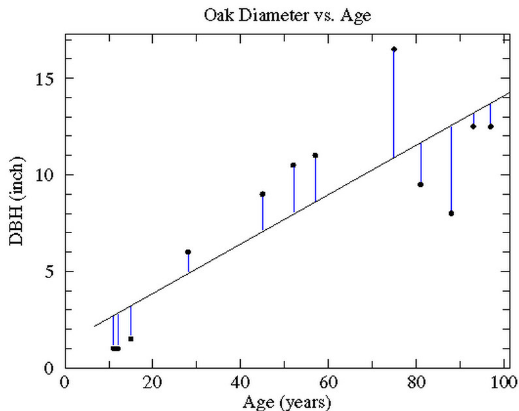
$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

Here $\vec{x}_k = (x_{k0}, x_{k1} \ldots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \mathbb{R}$ is the expected output.

Intuition: The network is supposed to compute an affine approximation of the function (some of) whose values are given in the training set.

Oaks in Wisconsin

Age (years)	*DBH* (inch)
97	12.5
93	12.5
88	8.0
81	9.5
75	16.5
57	11.0
52	10.5
45	9.0
28	6.0
15	1.5
12	1.0
11	1.0



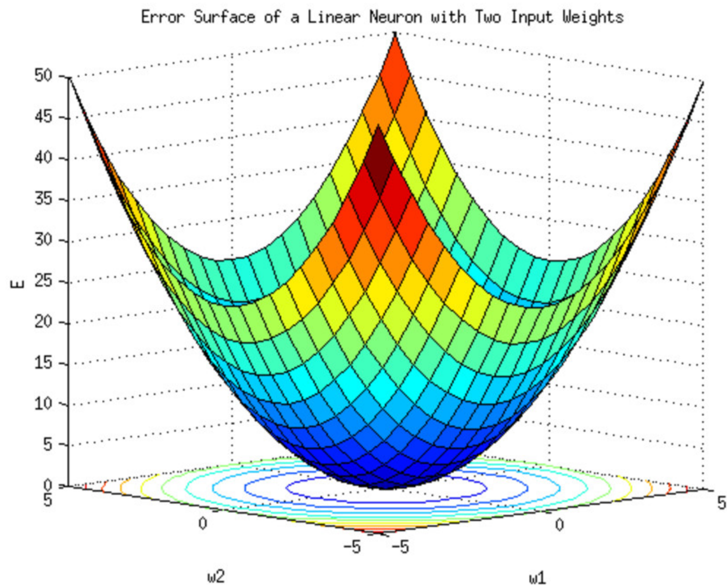
ADALINE

- **Error function:**

$$E(\vec{w}) = \frac{1}{2}\sum_{k=1}^{p}\left(\vec{w}\cdot\vec{x}_k - d_k\right)^2 = \frac{1}{2}\sum_{k=1}^{p}\left(\sum_{i=0}^{n} w_i x_{ki} - d_k\right)^2$$

- The goal is to find $\vec{w}$ which minimizes $E(\vec{w})$.

Error function



Gradient of the error function

Consider **gradient** of the error function:

$$\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_0}(\vec{w}), \ldots, \frac{\partial E}{\partial w_n}(\vec{w}) \right)$$

Intuition: $\nabla E(\vec{w})$ is a vector in the **weight space** which points in the direction of the *steepest ascent* of the error function.

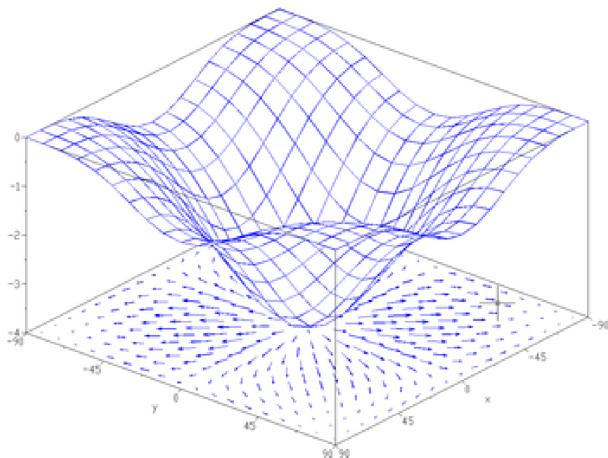
Note that the vectors $\vec{x}_k$ are just parameters of the function E, and are thus fixed!

Fact

If $\nabla E(\vec{w}) = \vec{0} = (0, \ldots, 0)$*, then* $\vec{w}$ *is a global minimum of* E*.*

For ADALINE, the error function $E(\vec{w})$ is a convex paraboloid and thus has the unique global minimum.

Gradient - illustration



Caution! This picture just illustrates the notion of gradient ... it is not the convex paraboloid $E(\vec{w})$!

Gradient of the error function (ADALINE)

$$
\begin{aligned}
\frac{\partial E}{\partial w_\ell}(\vec{w}) &= \frac{1}{2}\sum_{k=1}^{p} \frac{\delta}{\delta w_\ell}\left(\sum_{i=0}^{n} w_i x_{ki} - d_k\right)^2 \\
&= \frac{1}{2}\sum_{k=1}^{p} 2\left(\sum_{i=0}^{n} w_i x_{ki} - d_k\right)\frac{\delta}{\delta w_\ell}\left(\sum_{i=0}^{n} w_i x_{ki} - d_k\right) \\
&= \frac{1}{2}\sum_{k=1}^{p} 2\left(\sum_{i=0}^{n} w_i x_{ki} - d_k\right)\left(\sum_{i=0}^{n}\left(\frac{\delta}{\delta w_\ell} w_i x_{ki}\right) - \frac{\delta E}{\delta w_\ell} d_k\right) \\
&= \sum_{k=1}^{p}\left(\vec{w}\cdot\vec{x}_k - d_k\right) x_{k\ell}
\end{aligned}
$$

Thus

$$
\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_0}(\vec{w}), \ldots, \frac{\partial E}{\partial w_n}(\vec{w})\right) = \sum_{k=1}^{p}\left(\vec{w}\cdot\vec{x}_k - d_k\right)\vec{x}_k
$$

ADALINE - learning

Batch algorithm (gradient descent):

Idea: In every step "move" the weights in the direction *opposite* to the gradient.

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$, weights $\vec{w}^{(t+1)}$ are computed as follows:

$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \vec{x}_k - d_k\right) \cdot \vec{x}_k\end{aligned}$$

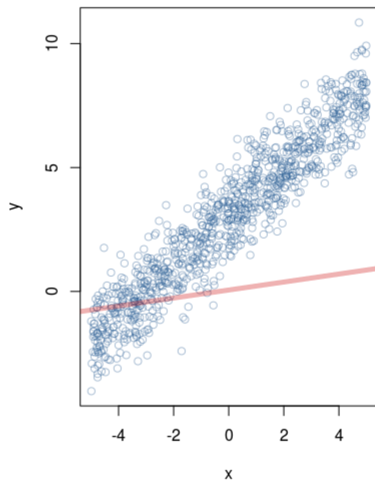
Here $k = (t \mod p) + 1$ and $0 < \varepsilon \leq 1$ is a *learning rate*.

Proposition

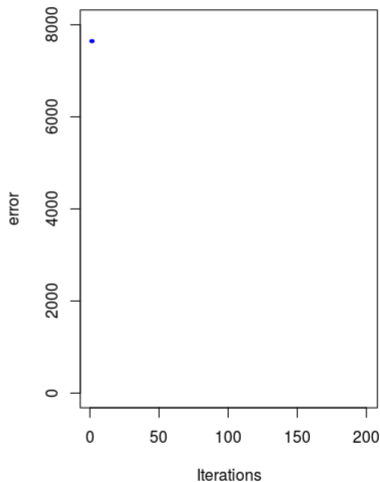
For sufficiently small $\varepsilon > 0$ the sequence $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$ converges (componentwise) to the global minimum of E (i.e. to the vector $\vec{w}$ satisfying $\nabla E(\vec{w}) = \vec{0}$).

ADALINE – Animation

Linear regression by gradient descent



Error function



Linear regression by gradient descent

Error function

ADALINE - learning

Online algorithm (Delta-rule, Widrow-Hoff rule):

- weights in $\vec{w}^{(0)}$ initialized randomly close to 0
- in the step $t+1$, weights $\vec{w}^{(t+1)}$ are computed as follows:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \left(\vec{w}^{(t)} \cdot \vec{x}_k - d_k\right) \cdot \vec{x}_k$$

Here $k = t \mod p + 1$ and $0 < \varepsilon(t) \leq 1$ is a learning rate in the step $t+1$.

Note that the algorithm does not work with the complete gradient but only with its part determined by the currently considered training example.

Theorem (Widrow & Hoff)

If $\varepsilon(t) = \frac{1}{t}$, then $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$ converges to the global minimum of E.

ADALINE - classification

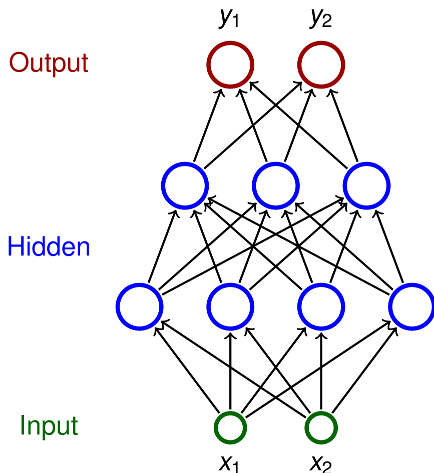
How to use the ADALINE for classification?

- The training set is

$$\mathcal{T} = \left\{ \left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right) \right\}$$

  kde $\vec{x}_k = (x_{k0}, x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^{n+1}$ a $d_k \in \{1, -1\}$.
  Here d_k determines a class.
- Train the network using the ADALINE algorithm.
- We may expect the following:
  - if $d_k = 1$, then $\vec{w} \cdot \vec{x}_k \geq 0$
  - if $d_k = -1$, then $\vec{w} \cdot \vec{x}_k < 0$
- This does not have to be always true but if the training set is reasonably linearly separable, then the algorithm typically gives satisfactory results.

Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i + 1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)

MLP – activity

Activity:

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function σ_j for neuron j (arbitrary differentiable)
[e.g. logistic sigmoid $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$]
- State of non-input neuron $j \in Z \setminus X$ after the computation stops:

$$y_j = \sigma_j(\xi_j)$$

(y_j depends on the configuration $\vec{w}$ and the input $\vec{x}$, so we sometimes write $y_j(\vec{w}, \vec{x})$)

- The network computes a function $\mathbb{R}^{|X|}$ do $\mathbb{R}^{|Y|}$. Layer-wise computation: First, all input neurons are assigned values of the input. In the ℓ-th step, all neurons of the ℓ-th layer are evaluated.

MLP – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{\left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p\right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

MLP – learning algorithm

Batch algorithm (gradient descent):

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t + 1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

  $$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

  where

  $$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

  is a *weight update* of w_{ji} in step $t + 1$ and $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t + 1$.

Note that $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$ is a component of the gradient ∇E, i.e. the weight update can be written as $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

MLP – error function gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \smallsetminus X$ we get

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \smallsetminus (Y \cup X)$$

(Here all y_j are in fact $y_j(\vec{w}, \vec{x}_k)$).

MLP – error function gradient

- If $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$ for all $j \in Z$, then

$$\sigma'_j(\xi_j) = \lambda_j y_j (1 - y_j)$$

and thus for all $j \in Z \smallsetminus X$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \lambda_r y_r (1 - y_r) \cdot w_{rj} \qquad \text{for } j \in Z \smallsetminus (Y \cup X)$$

- If $\sigma_j(\xi) = a \cdot \tanh(b \cdot \xi_j)$ for all $j \in Z$, then

$$\sigma'_j(\xi_j) = \frac{b}{a}(a - y_j)(a + y_j)$$

MLP – computing the gradient

Compute $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$ as follows:

Initialize $\mathcal{E}_{ji} := 0$
(By the end of the computation: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

For every $k = 1, \ldots, p$ do:

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$
2. **backward pass:** compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ using *backpropagation* (see the next slide!)
3. compute $\frac{\partial E_k}{\partial w_{ji}}$ for all w_{ji} using
$$\frac{\partial E_k}{\partial w_{ji}} := \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$
4. $\mathcal{E}_{ji} := \mathcal{E}_{ji} + \frac{\partial E_k}{\partial w_{ji}}$

The resulting $\mathcal{E}_{ji}$ equals $\frac{\partial E}{\partial w_{ji}}$.

MLP – backpropagation

Compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ as follows:

- if $j \in Y$, then $\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$
- if $j \in Z \setminus Y \cup X$, then assuming that j is in the ℓ-th layer and assuming that $\frac{\partial E_k}{\partial y_r}$ has already been computed for all neurons in the $\ell + 1$-st layer, compute

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$

(This works because all neurons of $r \in j^{\rightarrow}$ belong to the $\ell + 1$-st layer.)

Complexity of the batch algorithm

Computation of $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)})$ stops in time linear in the size of the network plus the size of the training set.

(assuming unit cost of operations including computation of $\sigma_r'(\xi_r)$ for given ξ_r)

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$
3. computes $\frac{\partial E_k}{\partial w_{ji}}$ and adds it to $\mathcal{E}_{ji}$ (a constant time operation in the unit cost framework)

The steps 1. - 3. take linear time.

Note that the speed of convergence of the gradient descent cannot be estimated ...

MLP – learning algorithm

Online algorithm:

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

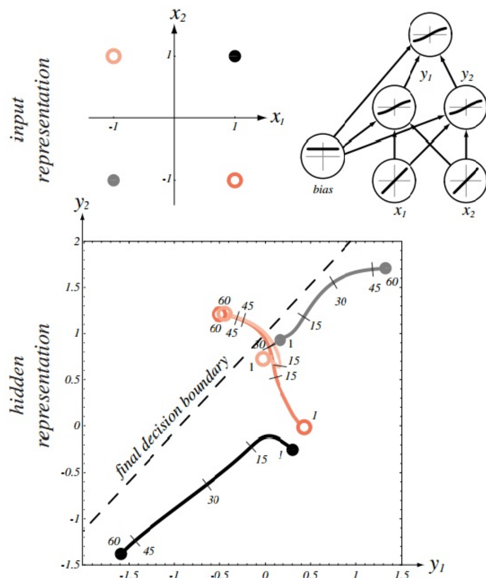
  where

$$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E_k}{\partial w_{ji}}(w_{ji}^{(t)})$$

  is the *weight update* of w_{ji} in the step $t+1$ and $0 < \varepsilon(t) \leq 1$ is the *learning rate* in the step $t+1$.

There are other variants determined by selection of the training examples used for the error computation (more on this later).

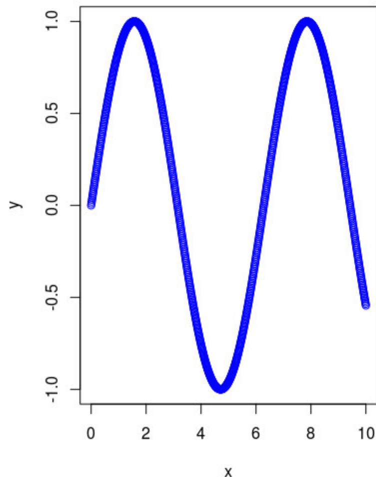
Illustration of the gradient descent – XOR

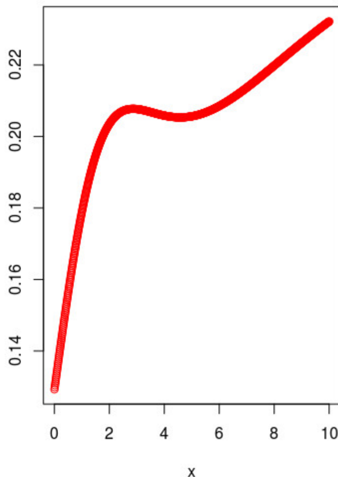


Source: Pattern Classification (2nd Edition); Richard O. Duda, Peter E. Hart, David G. Stork

Animation (sin(x)), network 1-5-1)

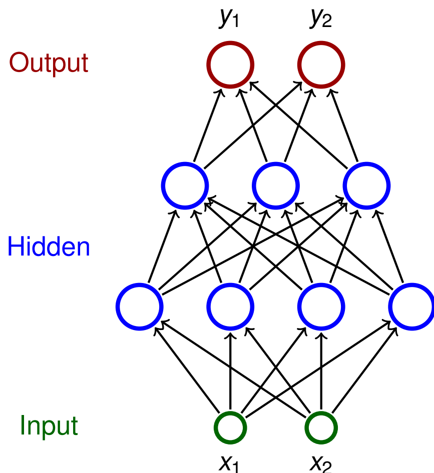
One iteration:





10 iterations:

Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i + 1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)

MLP – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

  Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

  where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

MLP – batch learning

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t + 1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta \vec{w}^{(t)}$$

Here

$$\Delta \vec{w}^{(t)} = -\varepsilon(t) \cdot \nabla E(\vec{w}^{(t)}) = -\varepsilon(t) \cdot \sum_{k=1}^{p} \nabla E_k(\vec{w}^{(t)})$$

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t + 1$
- $\nabla E(\vec{w}^{(t)})$ is the gradient of the error function
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error function *for the training example k*

MLP – error functions

- **square error:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where $E_k(\vec{w}) = \frac{1}{2}\sum_{j\in Y}\Big(y_j(\vec{w},\vec{x}_k) - d_{kj}\Big)^2$

- **mean square error (mse):**

$$E(\vec{w}) = \frac{1}{p}\sum_{k=1}^{p} E_k(\vec{w})$$

I will use mse throughout the rest of this lecture.

MLP – mse gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \frac{1}{p} \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \setminus X$ we get

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

(Here all y_j are in fact $y_j(\vec{w}, \vec{x}_k)$).

Practical issues of gradient descent

- Training efficiency:
  - What size of a minibatch?
  - How to choose the learning rate $\varepsilon(t)$ and control SGD ?
  - How to pre-process the inputs?
  - How to initialize weights?
  - How to choose desired output values of the network?
- Quality of the resulting model:
  - When to stop training?
  - Regularization techniques.
  - How large network?

For simplicity, I will illustrate the reasoning on MLP + mse.
Later we will see other topologies and error functions with different but always somewhat related issues.

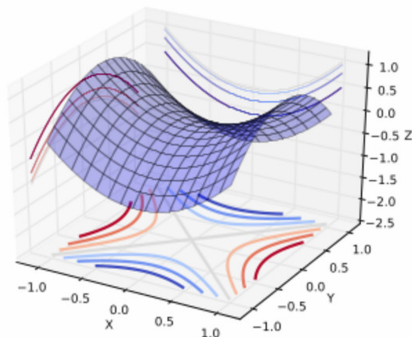
Issues in gradient descent

Lots of local minima where the descent gets stuck:

- The model identifiability problem: Swapping incoming weights of neurons *i* and *j* leaves the same network topology – **weight space symmetry**
- Recent studies show that for sufficiently large networks all local minima have low values of the error function.

Saddle points

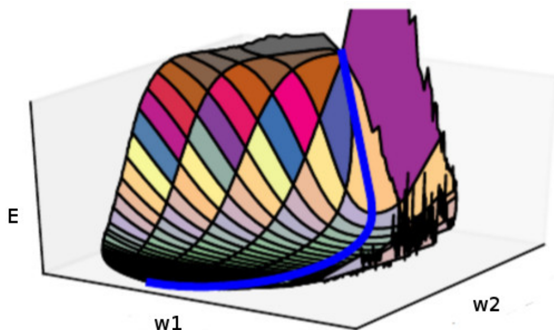
One can show (by a combinatorial argument) that larger networks have exponentially more saddle points than local minima.



Issues in gradient descent – too slow descent

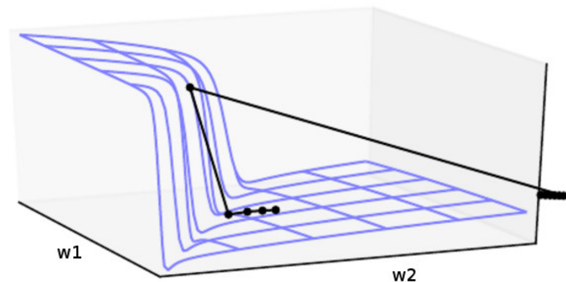
- flat regions

  E.g. if the inner potentials are too large (in abs. value), then their derivative is extremely small.

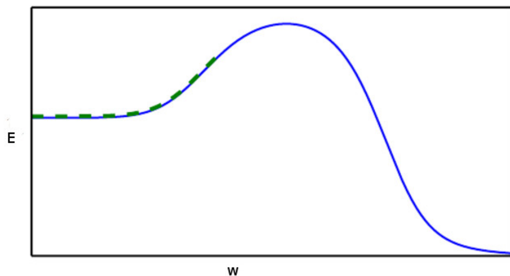


Issues in gradient descent – too fast descent

- steep cliffs: the gradient is extremely large, descent skips important weight vectors



Issues in gradient descent – local vs global structure



What if we initialize on the left?

Issues in computing the gradient

- vanishing and exploding gradients

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

- inexact gradient computation:
  - Minibatch gradient is only an estimate of the true gradient.
  - Note that the variance of the estimate is (roughly) $\sigma/\sqrt{m}$ where m is the size of the minibatch and σ is the variance of the gradient estimate for a single training example.
    (E.g. minibatch size 10 000 means 100 times more computation than the size 100 but gives only 10 times less variance.)

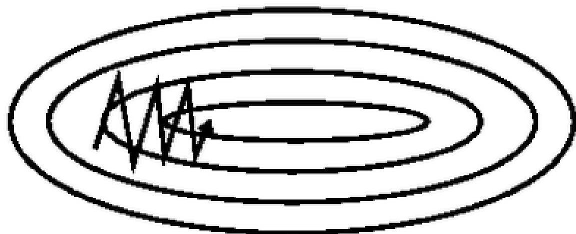
Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches.
- If all examples in the batch are to be processed in parallel (as is the typical case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process.

Moment

Issue in the gradient descent:

- $\nabla E(\vec{w}^{(t)})$ constantly changes direction (but the error steadily decreases).

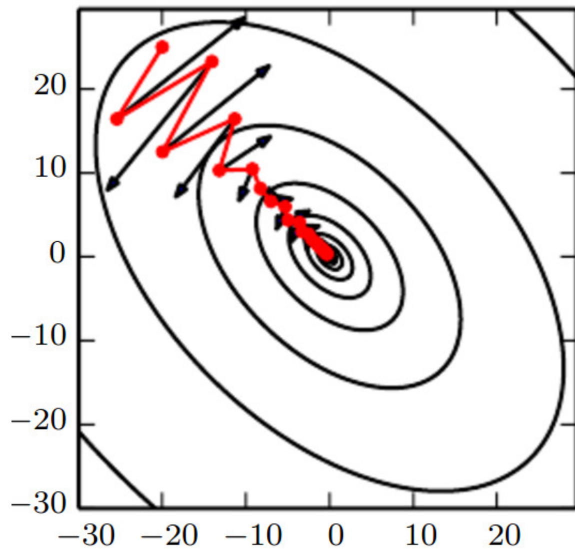


Solution: In every step add the change made in the previous step (weighted by a factor α):

$$\Delta \vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)}) + \alpha \cdot \Delta w_{ji}^{(t-1)}$$

where $0 < \alpha < 1$.

Momentum – illustration



SGD with momentum

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

where

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)}) + \alpha \Delta\vec{w}^{(t-1)}$$

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $0 < \alpha < 1$ measures the "influence" of the moment
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially.

Learning rate

Generic rules for adaptation of $\varepsilon(t)$

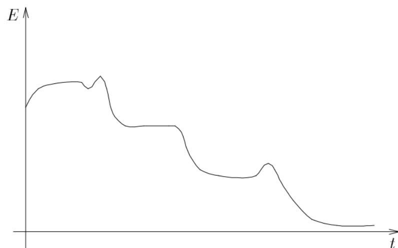
- Start with a larger learning rate (e.g. $\varepsilon = 0.1$).

  Later decrease as the descent is supposed to settle in a minimum of E.

  Some tools allow to set a list of learning rates, each rate for one epoch of the descent.

In case you may observe the error evolving:

- If the error decreases, increase slightly the rate.
- If the error increases, decrease the rate.
- Note that the error may increase for the short period without any harm to convergence of the learning process.



AdaGrad

So far we have considered a uniform learning rate.

It is better to have

- larger rates for weights with smaller updates,
- smaller rates for weights with larger updates.

AdaGrad uses individually adapting learning rate for each weight.

SGD with AdaGrad

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2\ldots$), compute $\vec{w}^{(t+1)}$:
  - Choose (randomly) a minibatch $T \subseteq \{1, \ldots, p\}$
  - Compute

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\frac{\eta}{\sqrt{r_j^{(t)} + \delta}} \cdot \sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)})$$

and

$$r_j^{(t)} = r_j^{(t-1)} + \left(\sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)}) \right)^2$$

- η is a constant expressing the influence of the learning rate, typically 0.01.
- $\delta > 0$ is a smoothing term (typically 1e-8) avoiding division by 0.

RMSProp

The main disadvantage of AdaGrad is the accumulation of the gradient throughout the whole learning process.

In case the learning needs to get over several "hills" before settling in a deep "valley", the weight updates get far too small before getting to it.

RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

SGD with RMSProp

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), compute $\vec{w}^{(t+1)}$:
  - Choose (randomly) a minibatch $T \subseteq \{1, \ldots, p\}$
  - Compute

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\frac{\eta}{\sqrt{r_j^{(t)} + \delta}} \cdot \sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)})$$

and

$$r_j^{(t)} = \rho r_j^{(t-1)} + (1 - \rho) \left(\sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)}) \right)^2$$

- η is a constant expressing the influence of the learning rate (Hinton suggests $\rho = 0.9$ and $\eta = 0.001$).
- $\delta > 0$ is a smoothing term (typically 1e-8) avoiding division by 0.

Other optimization methods

There are more methods such as AdaDelta, Adam (roughly RMSProp combined with momentum), etc.

A natural question: Which algorithm should one choose?

Unfortunately, there is currently no consensus on this point.

According to a recent study, the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam.

The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm.

Choice of (hidden) activations

Generic requirements imposed on activation functions:

1. differentiability

   (to do gradient descent)

2. non-linearity

   (linear multi-layer networks are equivalent to single-layer)

3. monotonicity

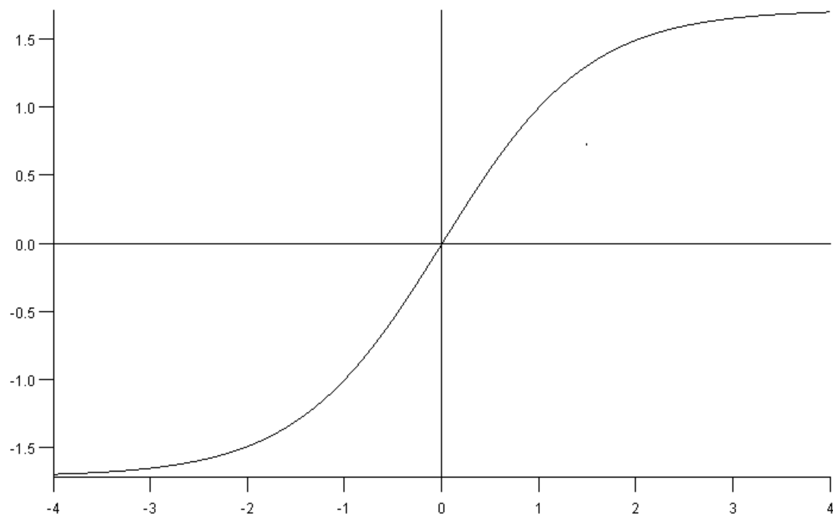
   (local extrema of activation functions induce local extrema of the error function)

4. "linearity"

   (i.e. preserve as much linearity as possible; linear models are easiest to fit; find the "minimum" non-linearity needed to solve a given task)

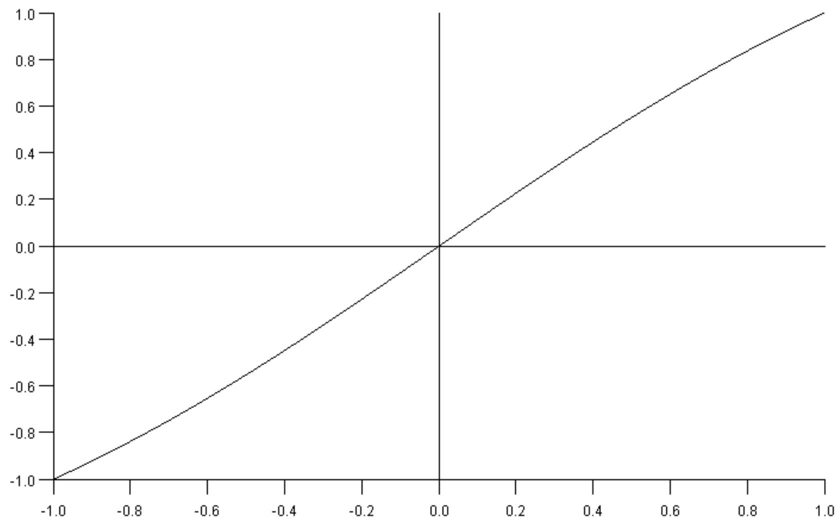
The choice of activation functions is closely related to input preprocessing and the initial choice of weights. I will illustrate the reasoning on sigmoidal functions; say few words about other activation functions later.

Activation functions – tanh



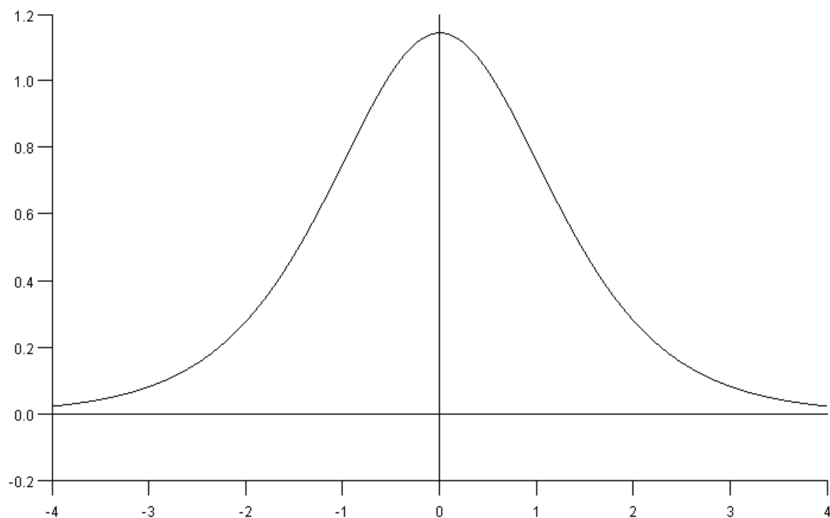
$\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$, we have $\lim_{\xi \to \infty} \sigma(\xi) = 1.7159$ and $\lim_{\xi \to -\infty} \sigma(\xi) = -1.7159$

Activation functions – tanh



$\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$ is almost linear on $[-1, 1]$

Activation functions – tanh



first derivative: $\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$

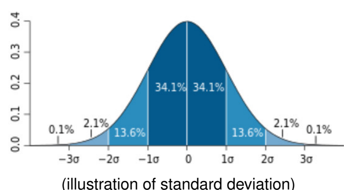
Input preprocessing

- Some inputs may be much larger than others.

  E.g..: Height vs weight of a person, maximum speed of a car (in km/h) vs its price (in CZK), etc.

- Large inputs have greater influence on the training than the small ones. In addition, too large inputs may slow down learning (saturation of activation functions).
- Typical standardization:
  - average $= 0$ (subtract the mean)
  - variance $= 1$ (divide by the standard deviation)

  Here the mean and standard deviation may be estimated from data (the training set).

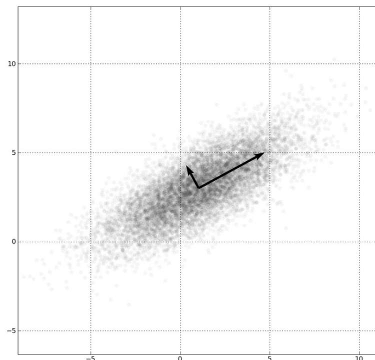


(illustration of standard deviation)

Input preprocessing

- Individual inputs should not be correlated.
- Correlated inputs can be removed as a part of *dimensionality reduction*.

  (Dimensionality reduction and decorrelation can be implemented using neural networks. There are also standard methods such as PCA.)



Initial weights (for tanh)

- Typically, the weights are chosen randomly from an interval $[-w, w]$ where w depends on the number of inputs of a given neuron.
- Consider the activation function $\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$ for all neurons.
  - σ is almost linear on $[-1, 1]$,
  - extreme values of σ'' are close to -1 and 1,
  - σ saturates out of the interval $[-4, 4]$ (i.e. it is close to its limit values and its derivative is close to 0.

  Thus
  - for too small w we may get (almost) linear model.
  - for too large w (i.e. much larger than 1) the activations may get saturated and the learning will be very slow.

  Hence, we want to choose w so that the inner potentials of neurons will be roughly in the interval $[-1, 1]$.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data. Assume that individual inputs are (almost) uncorrelated.
- Consider a neuron j from the first layer with d inputs. Assume that its weights are chosen uniformly from $[-w, w]$.
- **The rule:** choose w so that the *standard deviation* of ξ_j (denote by o_j) is close to the border of the interval on which σ_j is linear.

  In our case: $o_j \approx 1$.
- Our assumptions imply: $o_j = \sqrt{\frac{d}{3}} \cdot w$.

  Thus we put $w = \frac{\sqrt{3}}{\sqrt{d}}$.
- The same works for higher layers, d corresponds to the number of neurons in the layer one level lower.

Glorot & Bengio initialization

The previous heuristics for weight initialization ignores variance of the gradient (i.e. it is concerned only with the "size" of activations in the forward pass.

Glorot & Bengio (2010) presented a **normalized initialization** by choosing w uniformly from the interval:

$$\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Here m is the number of inputs to the neuron, m is the number of outputs of the neuron.

This is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance.

The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no non-linearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its non-linear counterparts.

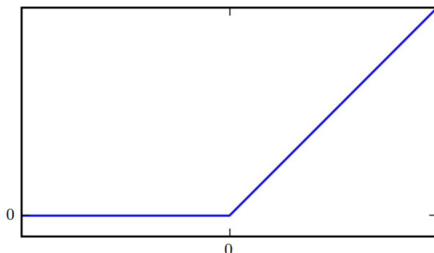
Target values (tanh)

- Target values d_{kj} should be chosen in the range of the output activation functions, in our case $[-1.716, 1.716]$.
- Target values too close to extrema of the output activations, in our case ± 1.716, may cause that the weights will grow indefinitely (slows down learning).
- Thus it is good to choose target values from the interval $[-1.716 + \delta, 1.716 - \delta]$.

  As before, ideally $[-1.716 + \delta, 1.716 - \delta]$ should span the interval on which the activation function is linear, i.e. d_{kj} should be taken from $[-1, 1]$.

Modern activation functions

For hidden neurons sigmoidal functions are often substituted with piece-wise linear activations functions. Most prominent is ReLU:



$\sigma(\xi) = \max\{0, \xi\}$

- THE default activation function recommended for use with most feedforward neural networks.
- As close to linear function as possible; very simple; does not saturate for large potentials.

Output neurons

The choice of activation functions for output units depends on the concrete applications.

For regression (function approximation) the output is typically linear (or sigmoidal).

For classification, the current activation functions of choice are

- logistic sigmoid or tanh – binary classification
- softmax:

  $$\sigma_j(\xi_j) = \frac{e^{\xi_j}}{\sum_{i \in Y} e^{\xi_i}}$$

  for multi-class classification.

For some reasons the error function used with softmax (assuming that the target values d_{kj} are from $\{0, 1\}$) is typically **cross-entropy**:

$$-\frac{1}{p} \sum_{k=1}^{p} \sum_{j \in Y} \left[d_{kj} \ln(y_j) + (1 - d_{kj}) \ln(1 - y_j) \right]$$

... which somewhat corresponds to the maximum likelihood principle.

Sigmoidal outputs with cross-entropy – in detail

Consider

- Binary classification, two classes $\{0, 1\}$
- One output neuron j, its activation logistic sigmoid

$$\sigma_j(\xi_j) = \frac{1}{1 + e^{-\xi_j}}$$

  The output of the network is $y = \sigma_j(\xi_j)$.
- For a training set

$$\mathcal{T} = \left\{ \left(\vec{x}_k, d_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

  (here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $d_k \in \mathbb{R}$), the cross-entropy looks like this:

$$E^{cross} = -\frac{1}{p} \sum_{k=1}^{p} \left[d_k \ln(y_k) + (1 - d_k) \ln(1 - y_k) \right]$$

  where y_k is the output of the network for the k-th training input $\vec{x}_k$, and d_k is the k-th desired output.

Generalization

Intuition: Generalization = ability to cope with new unseen instances.

Data are mostly noisy, so it is not good idea to fit exactly.

In case of function approximation, the network should not return exact results as in the training set.

More formally: It is typically assumed that the training set has been generated as follows:

$$d_{kj} = g_j(\vec{x}_k) + \Theta_{kj}$$

where g_j is the "underlying" function corresponding to the output neuron $j \in Y$ and Θ_{kj} is random noise.

The network should fit g_j not the noise.

Methods improving generalization are called **regularization methods**.

Regularization

Regularization is a big issue in neural networks, as they typically use a huge amount of parameters and thus are very susceptible to overfitting.

von Neumann: **"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."**

... and I ask you prof. Neumann:

What can you fit with 40GB of parameters??

Early stopping

Early stopping means that we stop learning before it reaches a minimum of the error E.

When to stop?

In many applications the error function is not the main thing we want to optimize.

E.g. in the case of a trading system, we typically want to maximize our profit not to minimize (strange) error functions designed to be easily differentiable.

Also, as noted before, minimizing E completely is not good for generalization.

For start: We may employ standard approach of training on one set and stopping on another one.

Early stopping

Divide your dataset into several subsets:

- **training set** (e.g. 60%) – train the network here
- **validation set** (e.g. 20%) – use to stop the training
- (possibly) **test set** (e.g. 20%) – use to compare trained models

What to use as a stopping rule?

You may observe E (or any other function of interest) on the validation set, if it does not improve for last k steps, stop.

Alternatively, you may observe the gradient, if it is small for some time, stop.

(recent studies shown that this traditional rule is not too good: it may happen that the gradient is larger close to minimum values; on the other hand, E does not have to be evaluated which saves time.

To compare models you may use ML techniques such as cross-validation etc.

Size of the network

Similar problem as in the case of the training duration:

- Too small network is not able to capture intrinsic properties of the training set.
- Large networks overfit faster – bad generalization.

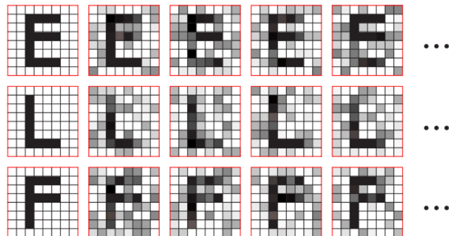
Solution: Optimal number of neurons :-)

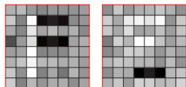
- there are some (useless) theoretical bounds
- there are algorithms dynamically adding/removing neurons (not much use nowadays)
- In practice:
  - start using a rule of thumb: the number of neurons $\approx$ ten times less than the number of training instances.
  - experiment, experiment, experiment.

Feature extraction

Consider a two layer network. Hidden neurons are supposed to represent "patterns" in the inputs.

Example: Network 64-2-3 for letter classification:





learned input-to-hidden weights

Ensemble methods

Techniques for reducing generalization error by combining several models.

The reason that ensemble methods work is that different models will usually not make all the same errors on the test set.

Idea: Train several different models separately, then have all of the models vote on the output for test examples.

Bagging:

- Generate k training sets $T_1, ..., T_k$ of the same size by *sampling from $\mathcal{T}$ uniformly with replacement*.

  If $|T_i| = |\mathcal{T}|$, then on average $|T_i| = (1 - 1/e)|\mathcal{T}|$.
- For each i, train a model M_i on T_i.
- Combine outputs of the models: for regression by averaging, for classification by (majority) voting.

Dropout

The algorithm: In every step of the gradient descent

- choose randomly a set N of neurons, each neuron is included in N independently with probability $1/2$,
  (in practice, different probabilities are used as well).
- update weights of neurons in N (in a standard way), leave weights of the other neurons unchanged.

Dropout resembles bagging: Large ensemble of neural networks is trained "at once" on parts of the data.

Dropout is not exactly the same as bagging: The models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

In the case of bagging, each model is trained to convergence on its respective training set. This would be infeasible for large networks/training sets.

Weight decay

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

In every step we decrease weights (multiplicatively) as follows:

$$w_{ji}^{(t+1)} = (1 - \zeta)(w_{ji}^{(t)} + \Delta w_{ji}^{(t)})$$

Intuition: Unimportant weights will be pushed to 0, important weights will survive the decay.

Weight decay is equivalent to the gradient descent with a constant learning rate ε and the following error function:

$$E'(\vec{w}) = E(\vec{w}) + \frac{2\zeta}{\varepsilon}(\vec{w} \cdot \vec{w})$$

Here $\frac{2\zeta}{\varepsilon}(\vec{w} \cdot \vec{w})$ penalizes large weights.

More optimization, regularization ...

There are many more practical tips, optimization methods, regularization methods, etc.

For a very nice survey see

`http://www.deeplearningbook.org/`

... and also all other infinitely many urls concerned with deep learning.

ALVINN (history)

Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina

ALVINN

Architecture:

- MLP, 960 – 4 – 30 (also 960 – 5 – 30)
- inputs correspond to pixels

Activity:

- activation functions: logistic sigmoid
- Steering wheel position determined by "center of mass" of neuron values.

ALVINN

Learning: Trained during (live) drive.

- Front window view captured by a camera, 25 images per second.
- Training samples of the form $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road
  - $\vec{d}_k$ = corresponding position of the steering wheel
- position of the steering wheel "blurred" by Gaussian distribution:

  $$d_{ki} = e^{-D_i^2/10}$$

  where D_i is the distance of the i-th output from the one which corresponds to the correct position of the wheel.

  (The authors claim that this was better than the binary output.)

ALVINN – Selection of training samples

Naive approach: take images directly from the camera and adapt accordingly.

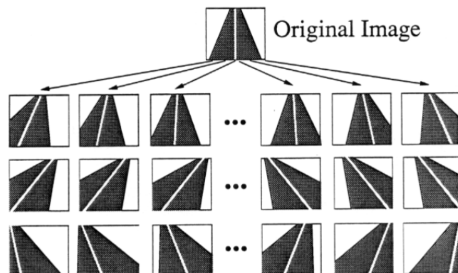
Problems:

- If the driver is gentle enough, the car never learns how to get out of dangerous situations. A solution may be
  - turn off learning for a moment, then suddenly switch on, and let the net catch on,
  - let the driver drive as if being insane (dangerous, possibly expensive).
- The real view out of the front window is repetitive and boring, the net would overfit on few examples.

ALVINN – Selection of training examples

Problem with a "good" driver is solved as follows:

- 15 distorted copies of each image:



- desired output generated for each copy

"Boring" images solved as follows:

- a buffer of 200 images (including 15 copies of the original), in every step the system trains on the buffer
- after several updates a new image is captured, 15 copies are made and they will substitute 15 images in the buffer (5 chosen randomly, 10 with the **smallest** error).

ALVINN - learning

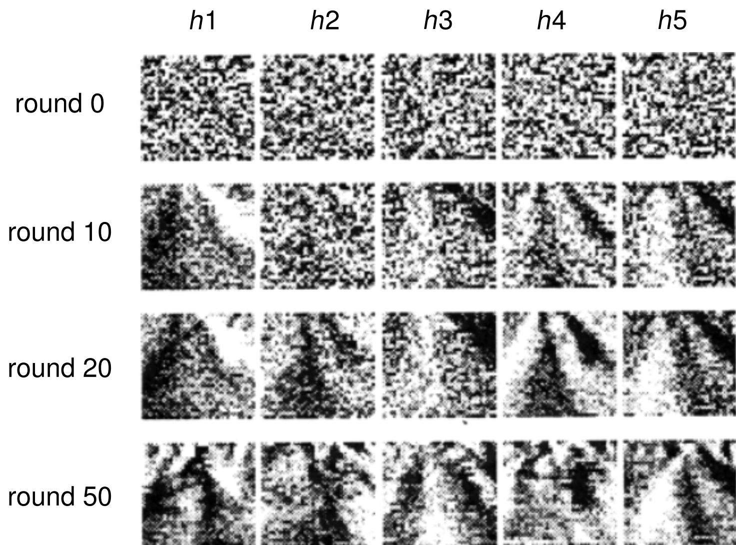
- pure backpropagation
- constant learning speed
- momentum, slowly increasing.

We used a learning rate of 0.015, a momentum term of 0.9, and we ramped up the learning rate and momentum using a rate term of 0.05. This means that the learning rate and momentum increase linearly over 20 epochs until they reach their maximum value (0.015 and 0.9, respectively). We also used a weight decay term of 0.0001.

Results:

- Trained for 5 minutes, speed 4 miles per hour.
- ALVINN was able to drive well on a new road it has never seen (in different weather conditions).
- The maximum speed was limited by the hydraulic controller of the steering wheel, not the learning algorithm.

ALVINN - weight development



Here $h1, \ldots, h5$ are hidden neurons.

ALVINN - comments

Compare ALVINN with explicit system development:
For driving you need to

- find key features for driving
  (ALVINN finds automatically)
- detect the features
  (ALVINN creates its own detectors)
- implement driving algorithm
  (ALVINN learns form the driver)

ALVINN was rather limited (but keep in mind that the net is **small**):

- just one type of road, no obstacles
- no higher level control

MNIST – handwritten digits recognition

- Database of labelled images of handwritten digits: 60 000 training examples, 10 000 testing.
- Dimensions: 28 x 28, digits are centered to the "center of gravity" of pixel values and normalized to fixed size.
- More at `http://yann.lecun.com/exdb/mnist/`

Fig. 4. Size-normalized examples from the MNIST database.

The database is used as a standard benchmark in lots of publications.

Allows comparison of various methods.

MNIST

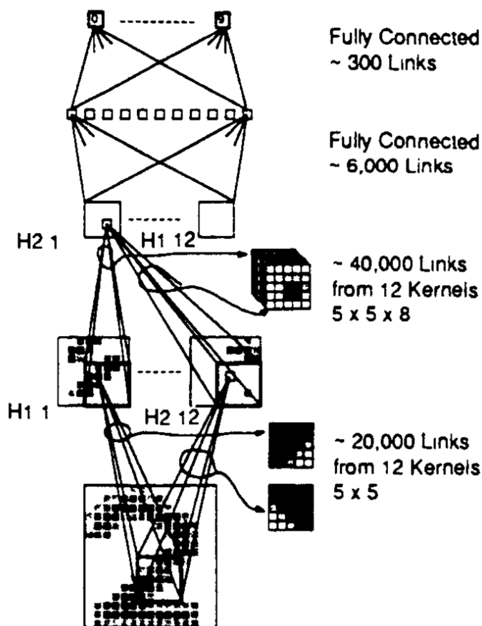
One of the best "old" results is the following:

6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU)
(Ciresan et al. 2010)

Abstrakt: Good old on-line back-propagation for plain multi-layer perceptrons yields a very low 0.35 error rate on the famous MNIST handwritten digits benchmark. All we need to achieve this best result so far are many hidden layers, many neurons per layer, numerous deformed training images, and graphics cards to greatly speed up learning.

A famous application of the first convolutional network LeNet-1 in 1998.

MNIST – LeNet1



MNIST – LeNet1

Activity: activation function: hyperbolic tangents

Interpretation of output:

- the output neuron with the highest value identifies the digit.
- the same, but if the two largest neuron values are too close together, the input is rejected (i.e. no answer).

Learning:
Inputs:

- training on 7291 samples, tested on 2007 samples

Training:

- modified backpropagation (conjugate gradients), online
- weights initialized uniformly from $[-2.4, 2.4]$, divided by the number of inputs to a given neuron

MNIST – results

- error on test set without rejection: 5%
- error on test set with rejection: 1% (12% rejected)

- compare with dense MLP with 40 hidden neurons: error 1% (19.4% rejected)

Modern convolutional networks

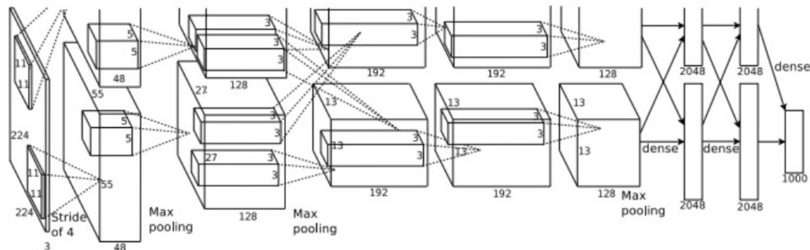
The rest of the lecture is based on the online book Neural Networks and Deep Learning by Michael Nielsen.
http://neuralnetworksanddeeplearning.com/index.html

- Convolutional networks are currently the best networks for image classification.
- Their common ancestor is LeNet-5 (and other LeNets) from nineties.

  Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998

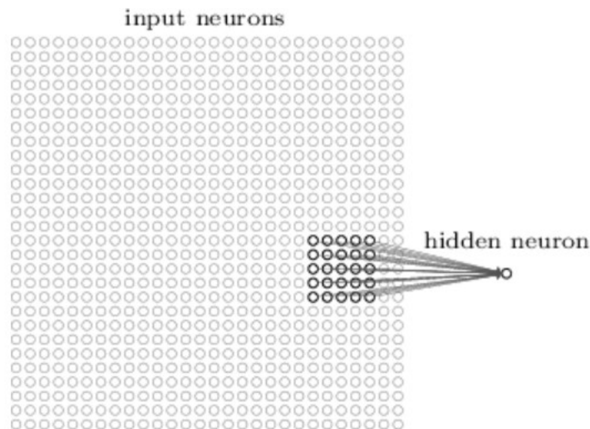
AlexNet

In 2012 this network made a breakthrough in ILVSCR competition, taking the classification error from around 28% to 16%:



A convolutional network, trained on two GPUs.

Convolutional networks - local receptive fields

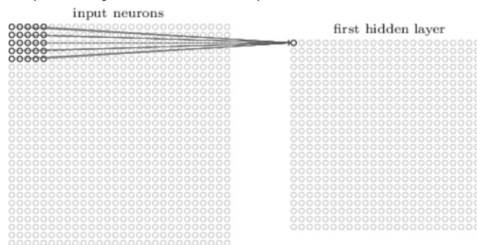


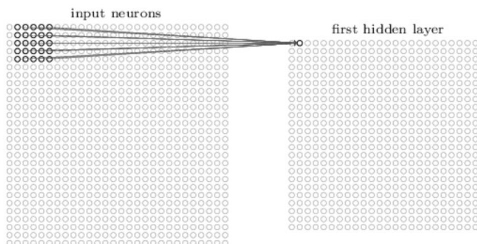
Every neuron is connected with a field of $k \times k$ (in this case 5×5) neurons in the lower layer (this filed is *receptive field*).

Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

Convolutional networks - stride length

Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:

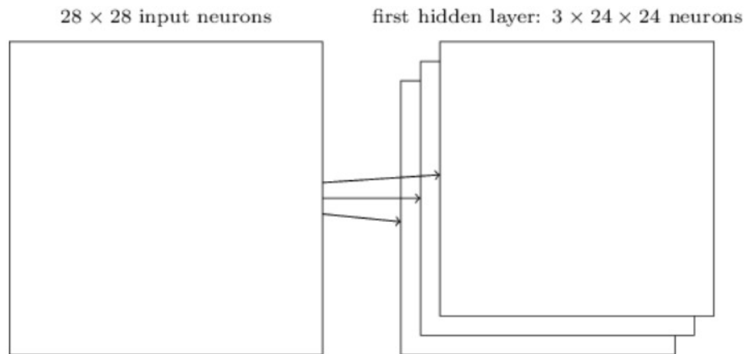




The "size" of the slide is called *stride length*.

The group of all such neurons is *feature map*. all these neurons *share weights and biases*!

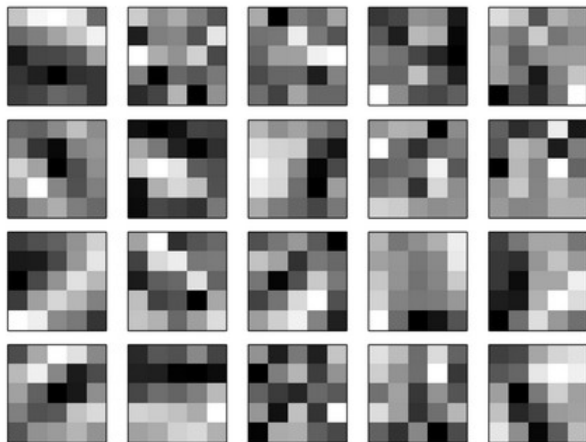
Feature maps



Each feature map represents a property of the input that is supposed to be spatially invariant.

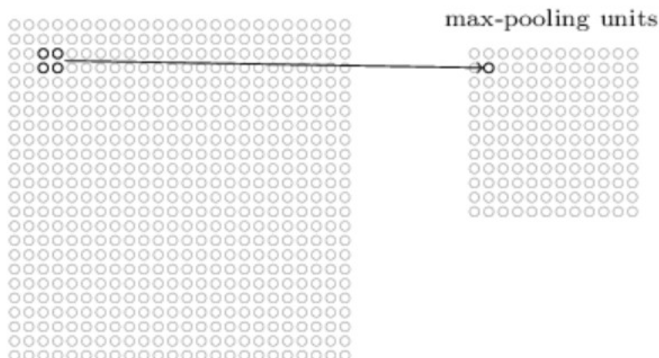
Typically, we consider several feature maps in a single layer.

Trained feature maps



(20 feature maps, receptive fields 5×5)

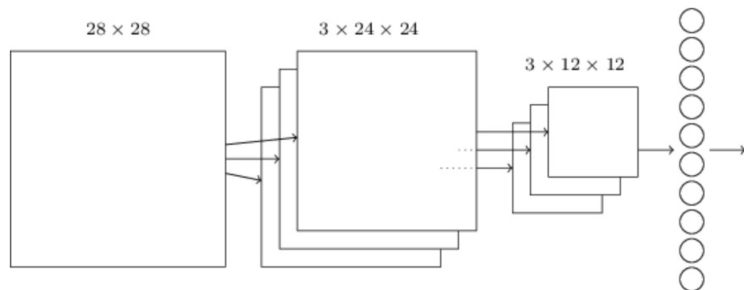
Pooling



Neurons in the pooling layer compute functions of their receptive fields:

- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- ...

Simple convolutional network

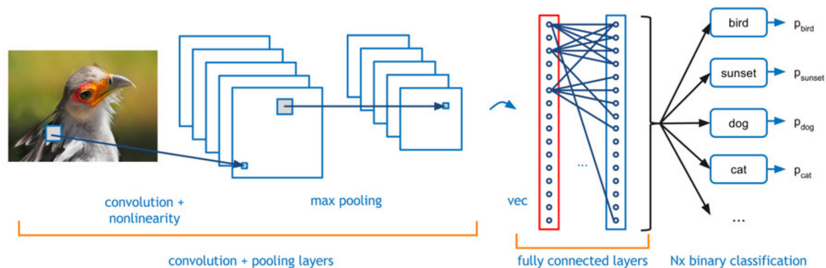


28×28 input image, 3 feature maps, each feature map has its own max-pooling (field 5×5, stride = 1), 10 output neurons.

Each neuron in the output layer gets input from each neuron in the pooling layer.

Trained using backprop, which can be easily adapted to convolutional networks.

Convolutional network



Simple convolutional network vs MNIST

two convolutional-pooling layers, one 20, second 40 feature maps, two dense (MLP) layers (1000-1000), outputs (10)

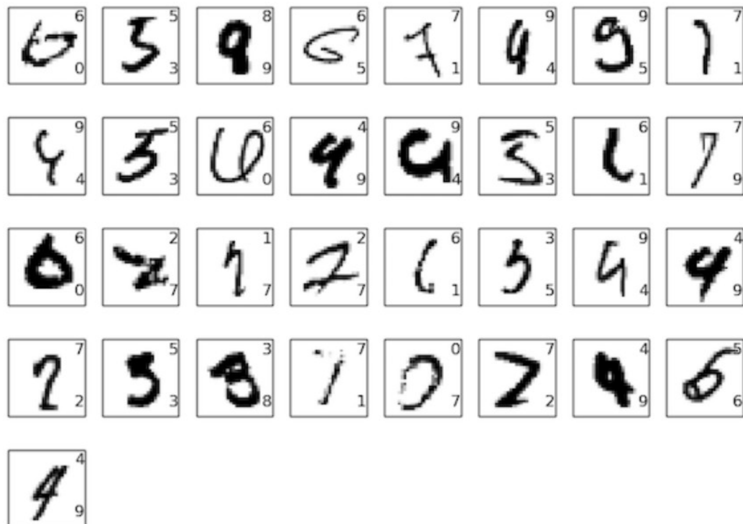
- Activation functions of the feature maps and dense layers: ReLU
- max-pooling
- output layer: soft-max
- Error function: negative log-likelihood (= cross-entropy)
- Training: SGD, mini-batch size 10
- learning rate 0.03
- L2 regularization with "weight" $\lambda = 0.1$ + dropout with prob. $1/2$
- training for 40 epochs (i.e. every training example is considered 40 times)
- Expanded dataset: displacement by one pixel to an arbitrary direction.
- Committee voting of 5 networks.

Simple convolutional network in Theano

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(
            n_in=40*4*4, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        FullyConnectedLayer(
            n_in=1000, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        SoftmaxLayer(n_in=1000, n_out=10, p_dropout=0.5)],
        mini_batch_size)
>>> net.SGD(expanded_training_data, 40, mini_batch_size, 0.03,
            validation_data, test_data)
```

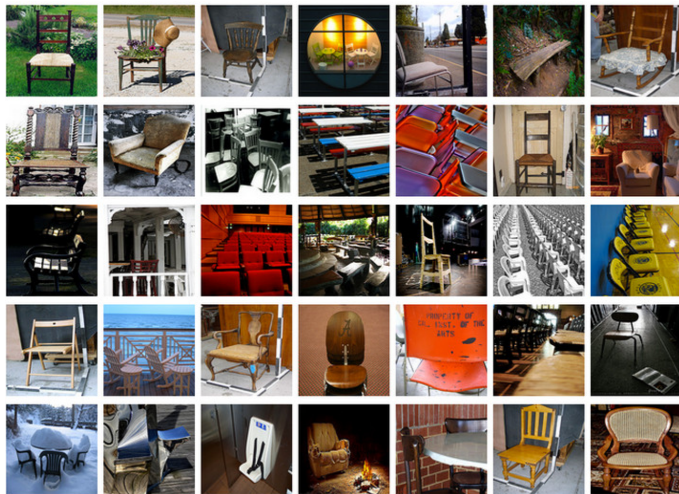
MNIST

Out of 10 000 images in the test set, only these 33 have been incorrectly classified:



More complex convolutional networks

Convolutional networks have been used for classification of images from the ImageNet database (16 million color images, 20 thousand classes)



ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

Competition in classification over a subset of images from ImageNet.

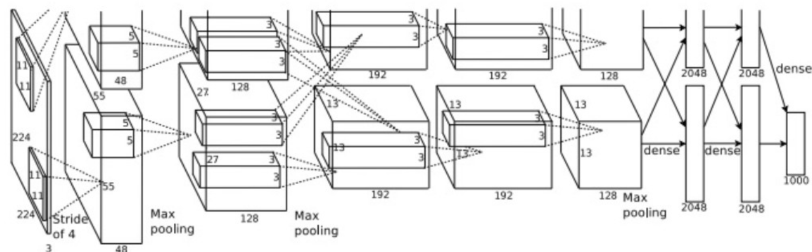
Started in 2010, assisted in breakthrough in image recognition.

Training set 1.2 million images, 1000 classes. Validation set: 50 000, test set: 150 000.

Many images contain more than one object $\Rightarrow$ model is allowed to choose five classes, the correct label must be among the five. (top-5 criterion).

AlexNet

ImageNet classification with deep convolutional neural networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton (2012).



Trained on two GPUs (NVIDIA GeForce GTX 580)

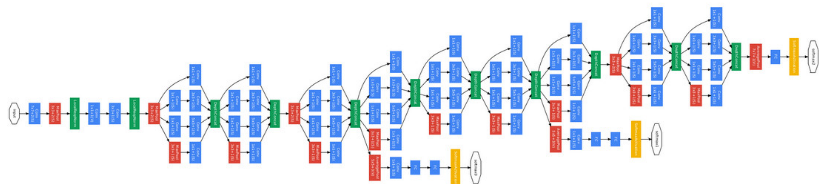
Výsledky:

- accuracy 84.7% in top-5 (second best algorithm at the time 73.8%)
- 63.3% "perfect" (top-1) classification

ILSVRC 2014

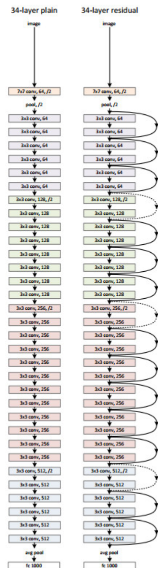
The same set as in 2012, top-5 criterion.

GoogLeNet: deep convolutional network, 22 layers



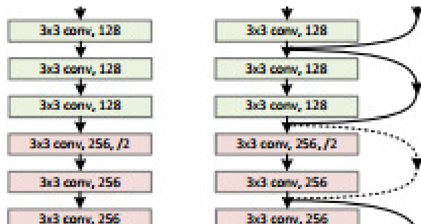
Results:

- Accuracy 93.33% top-5



- Deep convolutional network
- Various numbers of layers, the winner has 152 layers
- Skip connections implementing residual learning

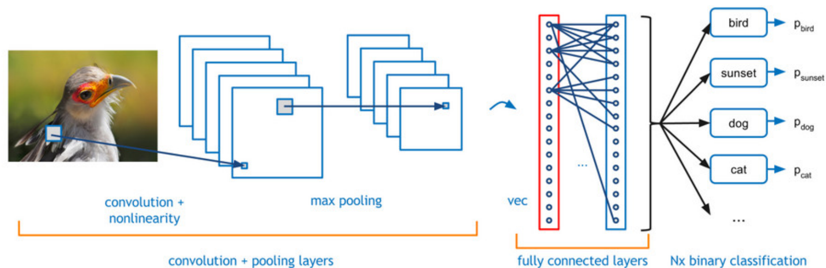
- Error **3.57%** in top-5.



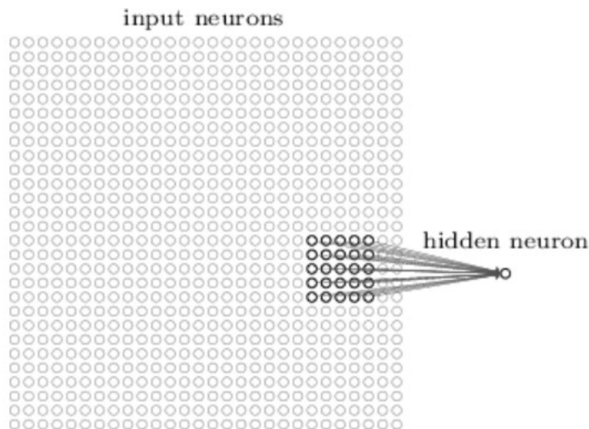
Superhuman convolutional nets?!

Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought we could recruit paid undergrads. Then I organized a labeling party of intense labeling effort only among the (expert labelers) in our lab. Then I developed a modified interface that used GoogLeNet predictions to prune the number of categories from 1000 to only about 100. It was still too hard - people kept missing categories and getting up to ranges of 13-15% error rates. In the end I realized that to get anywhere competitively close to GoogLeNet, it was most efficient if I sat down and went through the painfully long training process and the subsequent careful annotation process myself... The labeling happened at a rate of about 1 per minute, but this decreased over time... Some images are easily recognized, while some images (such as those of fine-grained breeds of dogs, birds, or monkeys) can require multiple minutes of concentrated effort. I became very good at identifying breeds of dogs... Based on the sample of images I worked on, the GoogLeNet classification error turned out to be 6.8%... My own error in the end turned out to be 5.1%, approximately 1.7% better.

Convolutional network



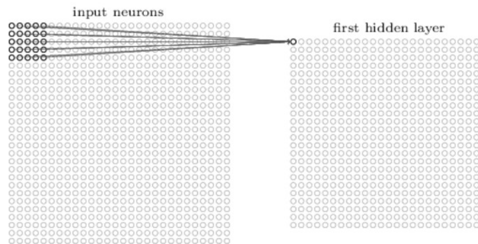
Convolutional layers

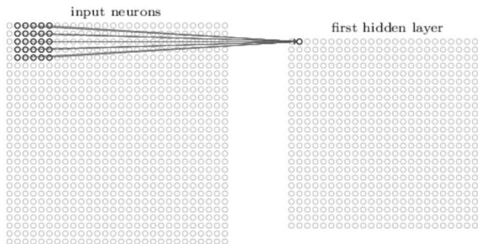


Every neuron is connected with a (typically small) *receptive field* of neurons in the lower layer.

Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

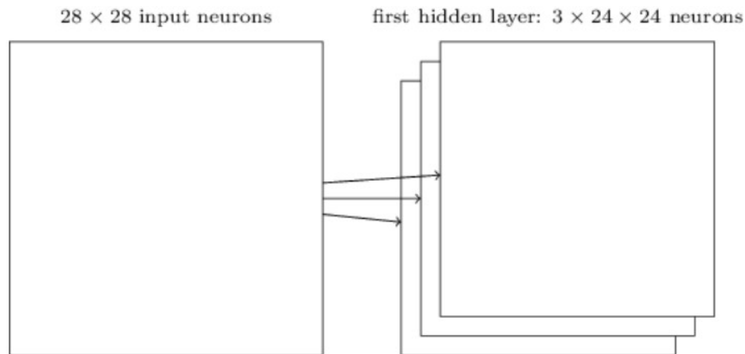
Convolutional layers





Neurons grouped into *feature maps* sharing weights.

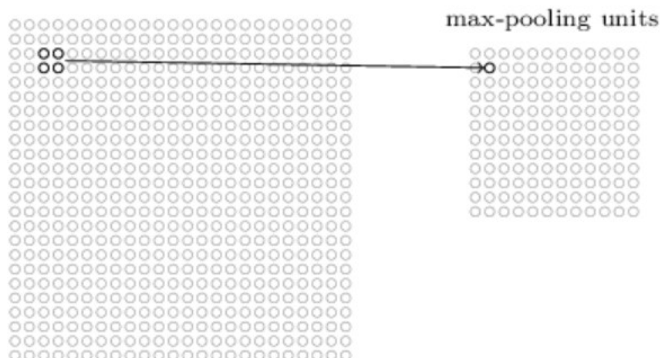
Convolutional layers



Each feature map represents a property of the input that is supposed to be spatially invariant.

Typically, we consider several feature maps in a single layer.

Pooling layers



Neurons in the pooling layer compute simple functions of their receptive fields (the fields are typically disjoint):

- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- ...

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional & pooling** layer L_m: Contains two sub-layers:
  - **convolutional layer**: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)
  - **pooling layer**: Each (convolutional) feature map F has a corresponding **pooling map** P. Neurons of P
    - have inputs only from F (typically few of them),
    - compute a simple aggregate function (such as max),
    - have *disjoint inputs*.

Convolutional networks – architecture

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)
- j_{share} is a set of neurons sharing weights with j
  i.e. neurons that belong to the same feature map as j

Convolutional networks – activity

- neurons of dense and convolutional layers:
  - inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

  - activation function σ_j for neuron j (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

- Neurons of pooling layers: Apply the "pooling" function:
  - max-pooling:

$$y_j = \max_{i \in j_{\leftarrow}} y_i$$

  - avg-pooling:

$$y_j = \frac{\sum_{i \in j_{\leftarrow}} y_i}{|j_{\leftarrow}|}$$

A convolutional network is evaluated layer-wise (as MLP), for each $j \in Y$ we have that $y_j(\vec{w}, \vec{x})$ is the value of the output neuron j after evaluating the network with weights $\vec{w}$ and input $\vec{x}$.

Convolutional networks – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

  Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function – mean square error (for example):**

$$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

  where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

Convolutional networks – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

    $$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

    where

    $$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \frac{1}{|T|} \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here T is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

Backprop

Recall that $\nabla E_k(\vec{w}^{(t)})$ is a vector of all partial derivatives of the form $\frac{\partial E_k}{\partial w_{ji}}$.

How to compute $\frac{\partial E_k}{\partial w_{ji}}$?

First, switch from derivatives w.r.t. w_{ji} to derivatives w.r.t. y_j:

- Recall that for every w_{ji} where j is in a dense layer, i.e. does not share weights:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

- Now for every w_{ji} where j is in a convolutional layer, that is shares w_{ji} with neurons of j_{share}:

$$\frac{\partial E_k}{\partial w_{ji}} = \sum_{r \in j_{share}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot y_r$$

- Neurons of pooling layers do not have weights.

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$$

This holds for the mean-square error, for other error functions the derivative w.r.t. outputs will be different.

- for every $j \in Z \setminus Y$ such that $j^{\rightarrow}$ is either a dense layer, or a convolutional layer:

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$

- for every $j \in Z \setminus Y$ such that $j^{\rightarrow}$ is max-pooling: Then $j^{\rightarrow} = \{i\}$ for a single "max" neuron and we have

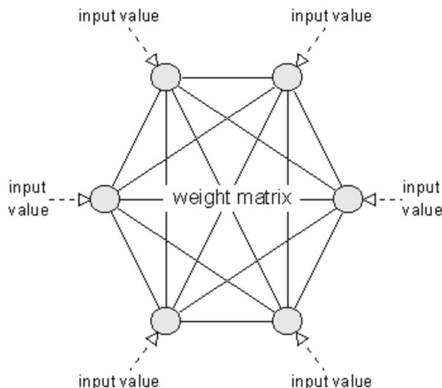
$$\frac{\partial E_k}{\partial y_j} = \begin{cases} \frac{\partial E_k}{\partial y_i} & \text{if } j = arg\ max_{r \in i_{\leftarrow}} y_r \\ 0 & \text{otherwise} \end{cases}$$

I.e. gradient can be propagated from the output layer downwards as in MLP.

Convolutional networks – conclusions

- Conv. nets. are nowadays the most used networks in image processing (and also in other areas where input has some local, "spatially" invariant properties)
- Typically trained using backpropagation.
- Due to the weight sharing allow (very) deep architectures.
- Typically extended with more adjustments and tricks in their topologies.

Recurrent networks – Hopfield network



Auto-associative network: Given an input, the network outputs a training example (encoded in its weights) "similar" to the given input.

Hopfield network

Architecture:

- complete topology, i.e. output of each neuron is input to all neurons
- all neurons are both input and output
- denote by $\xi_1, \ldots, \xi_n$ inner potentials and by $y_1, \ldots, y_n$ outputs (states) of individual neurons
- denote by w_{ji} the weight of connection from a neuron $i \in \{1, \ldots, n\}$ to a neuron $j \in \{1, \ldots, n\}$
- assume $w_{jj} = 0$ for every $j = 1, \ldots, n$
- **For now:** no neuron has a bias

Learning: Training set

$$\mathcal{T} = \{\vec{x}_k \mid \vec{x}_k = (x_{k1}, \ldots, x_{kn}) \in \{-1, 1\}^n, k = 1, \ldots, p\}$$

The goal is to "store" the training examples of $\mathcal{T}$ so that the network is able to *associate* similar examples.

Hebb's learning rule: If the inputs to a system cause the same pattern of activity to occur repeatedly, the set of active elements constituting that pattern will become increasingly strongly interassociated. That is, each element will tend to turn on every other element and (with negative weights) to turn off the elements that do not form part of the pattern. To put it another way, the pattern as a whole will become "auto-associated".

Mathematically speaking:

$$w_{ji} = \sum_{k=1}^{p} x_{kj} x_{ki} \qquad 1 \leq j \neq i \leq n$$

Intuition: "Neurons that fire together, wire together".

Hopfield network

Learning: Training set

$$\mathcal{T} = \{\vec{x}_k \mid \vec{x}_k = (x_{k1}, \ldots, x_{kn}) \in \{-1, 1\}^n, k = 1, \ldots, p\}$$

Hebb's rule:

$$w_{ji} = \sum_{k=1}^{p} x_{kj} x_{ki} \qquad 1 \leq j \neq i \leq n$$

Note that $w_{ji} = w_{ij}$, i.e. the weight matrix is symmetric.

Learning can be seen as poll about equality of inputs:

- If $x_{kj} = x_{ki}$, then the training example votes for "i equals j" by adding one to w_{ji}.
- If $x_{kj} \neq x_{ki}$, then the training example votes for "i does not equal j" by subtracting one from w_{ji}.

Hopfield network

Activity: Initially, neurons set to the network input $\vec{x} = (x_1, \ldots, x_n)$, thus $y_j^{(0)} = x_j$ for every $j = 1, \ldots, n$.

Cyclically update states of neurons, i.e. in step $t + 1$ compute the value of a neuron j such that $j = (t \mod p) + 1$, as follows:

Compute the inner potential:

$$\xi_j^{(t)} = \sum_{i=1}^{n} w_{ji} y_i^{(t)}$$

then

$$y_j^{(t+1)} = \begin{cases} 1 & \xi_j^{(t)} > 0 \\ y_j^{(t)} & \xi_j^{(t)} = 0 \\ -1 & \xi_j^{(t)} < 0 \end{cases}$$

Hopfield network – activity

The computation stops in a step t^* if the network is for the first time in a *stable* state, i.e.

$$y_j^{(t^*+n)} = y_j^{(t^*)} \qquad (j = 1, \ldots, n)$$

Theorem

Assuming symmetric weights, computation of a Hopfiled network always stops for every input.

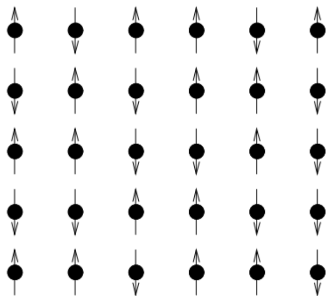
This implies that a given Hopfiled network computes a function from $\{-1,1\}^n$ to $\{-1,1\}^n$ (determined by its weights).

Denote by $\vec{y}(W,\vec{x}) = \left(y_1^{(t^*)}, \ldots, y_n^{(t^*)}\right)$ the value of the network for a given input $\vec{x}$ and a weight matrix W.
Denote by $y_j(W,\vec{x}) = y_j^{(t^*)}$ the component of the value of the network corresponding to the neuron j.

If W is clear from the context, we write only $y(\vec{x})$ a $y_j(\vec{x})$.

Ising model – an analogy

Simple models of magnetic materials resemble Hopfield network.



- atomic magnets organized into square-lattice
- each magnet may have only one of two possible orientations (in the Hopfield network $+1$ a -1)
- orientation of each magnet is influenced by an external magnetic field (input of the network) as well as orientation of the other magnets
- weights in the Hopfiled net model determine interaction among magnets

Energy function

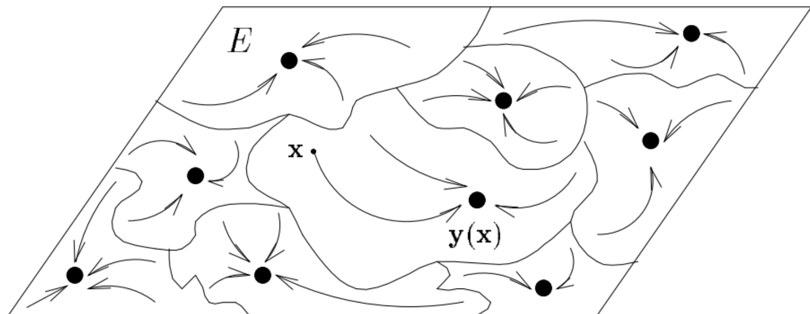
Energy function E assigns to every state $\vec{y} \in \{-1,1\}^n$ a (potential) energy:

$$E(\vec{y}) = -\frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} w_{ji} y_j y_i$$

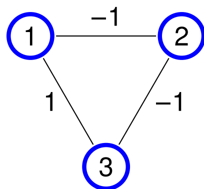
- states with low energy are stable (few neurons "want to" change their states), states with high energy are not stable
- i.e. large (positive) $w_{ji} y_j y_i$ is stable and small (negative) $w_{ji} y_j y_i$ is not stable

The energy does not increase during computation: $E(\vec{y}^{(t)}) \geq E(\vec{y}^{(t+1)})$, stable states $\vec{y}^{(t^*)}$ correspond to local minima of E.

Energy landscape



Hopfield – example



y_1	y_2	y_3	E
1	1	1	1
1	1	−1	1
1	−1	1	−3
1	−1	−1	1
−1	1	1	1
−1	1	−1	−3
−1	−1	1	1
−1	−1	−1	1

- Hopfield network with three neurons
- trained on a single training example $(1, -1, 1)$ using Hebb's rule

  (note that $(-1, 1, -1)$ has also been "stored" into the network)

Hopfield network – convergence

Observe that

- the energy does not increase during computation: $E(\vec{y}^{(t)}) \geq E(\vec{y}^{(t+1)})$
- if the state is updated in a step $t + 1$, then $E(\vec{y}^{(t)}) > E(\vec{y}^{(t+1)})$
- there are only finitely many states, and thus, eventually, a local minimum of E is reached.

This proves that computation of a Hopfield network always stops.

Hopfield network – phantoms

The energy function E may have local minima that do not correspond to training examples (so called phantoms).

Phantoms can be "unlearned" e.g. using the following rule: Given a phantom $(x_1, \ldots, x_n) \in \{-1, 1\}^n$ and weights w_{ji}, then new weights w'_{ji} are computed by

$$w'_{ji} = w_{ji} - x_i x_j$$

(i.e. similar to Hebb's rule but with the opposite sign)

Reproduction – statistical analysis

Capacity of Hopfield network is defined as the ratio p/n of number of training examples the net is able to learn over the number of neurons.

Assume that training examples are chosen randomly: each component of $\vec{x}_k$ is set to 1 with probability $1/2$ and to -1 with probability $1/2$.

Consider a configuration W obtained by learning using the Hebb's rule.

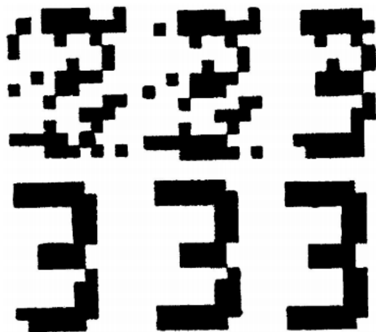
Denote

$$\beta = \mathbf{P}\left[\vec{x}_k = \vec{y}(W, \vec{x}_k) \text{ pro } k = 1, \ldots, p\right]$$

Then for $n \to \infty$ and $p \leq n/(4 \log n)$ we have $\beta \to 1$.

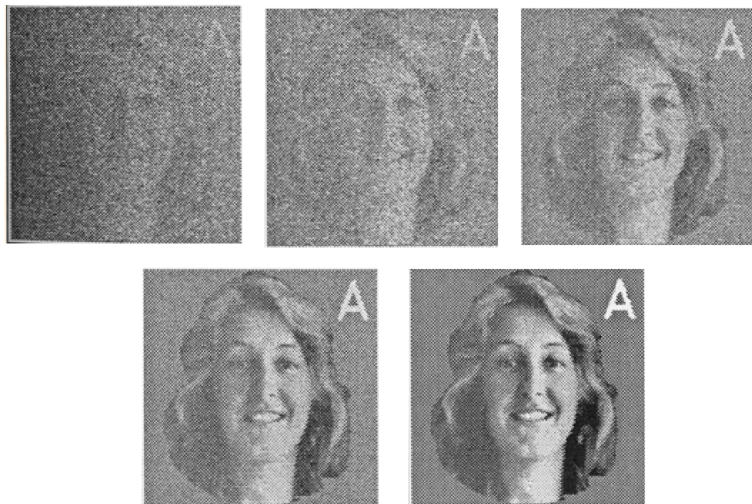
I.e. the maximum number of examples that can be effectively stored in Hopfield net is proportional to $n/(4 \log n)$.

Hopfield network – example

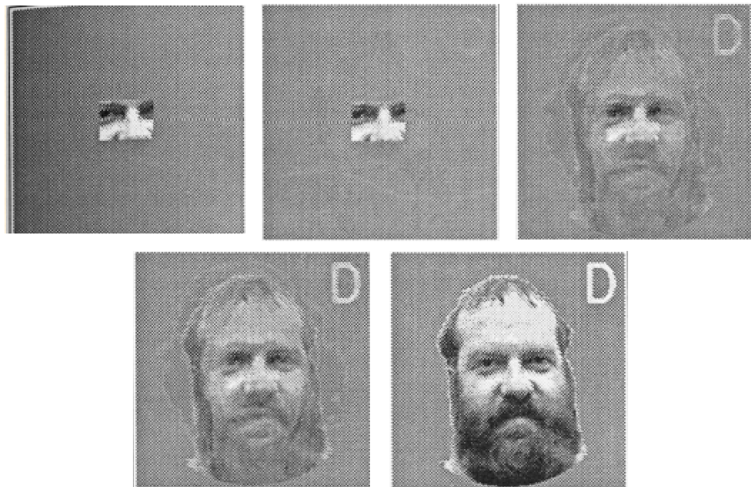


- figures 12×10
  (120 neurons, -1 is white and 1 is black)
- learned 8 figures
- input generated with 25% noise
- image shows the activity of the Hopfield network

Hopfield network – example



Hopfield network – example



Hopfield network – local minima

We look for "deep" minima of E ....

We may get suck in a shallow minimum.

Solution: In every state we allow transition to states with higher energy. This transition has a small probability (which will be higher at the beginning and decrease throughout computation).

Boltzmann activity

Activity: States of neurons initially set to values of $\{-1, 1\}$, i.e., $y_j^{(0)} \in \{-1, 1\}$ for $j \in \{1, \ldots, n\}$.

In the step $t + 1$ update value of a randomly chosen neuron $j \in \{1, \ldots, n\}$ as follows: Compute the inner potential

$$\xi_j^{(t)} = \sum_{i=1}^{n} w_{ji} y_i^{(t)}$$

choose $y_j^{(t+1)} \in \{-1, 1\}$ randomly so that

$$\mathbf{P}\left[y_j^{(t+1)} = 1\right] = \sigma\left(\xi_j^{(t)}\right)$$

where

$$\sigma(\xi) = \frac{1}{1 + e^{-2\xi/T(t)}}$$

The parameter $T(t)$ is called **temperature** in time t.

Temperature and energy

- High temperature $T(t)$ implies that $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx \frac{1}{2}$ and thus the network behaves almost randomly.
- Very low temperature $T(t)$ implies that either $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx 1$ or $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx 0$ depending on whether $\xi_j^{(t)} > 0$ or $\xi_j^{(t)} < 0$. Thus the network behaves almost deterministically (as in the original activity of Hopfield network).

Notes:

- Boltzmann activity = Hopfield activity + random noise,
- energy $E(\vec{y}) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ji}y_jy_i$ may jump to higher levels (with probability depending on the temperature),
- the probability of transition to higher energy decreases exponentially with the size of the "energy jump".

Simulated annealing

The following approach may help to reach deep minima of E:

- Start with higher temperature $T(t)$
- Gradually reduce the temperature, e.g. as follows:
  - $T(t) = \eta^t \cdot T(0)$ where $\eta < 1$ is close to 1
  - or $T(t) = T(0)/\log(1 + t)$

- This process resembles annealing used in metallurgy that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness.
- It also extends physical motivation of Hopfield networks: magnet orientation is now, in addition, influenced by thermal fluctuations.

... and it gets us close to Boltzmann machines.

Boltzmann machine

Architecture:

- Neural network with cycles and symmetric connections (i.e. arbitrary graph)
- N is a *set of all neurons*.
- Denote by ξ_j the inner potential and by y_j the output (i.e. state) of neuron j.
  State of the machine: $\vec{y} \in \{-1, 1\}^{|N|}$.
- Denote by $w_{ji} \in \mathbb{R}$ the weight of the connection from i to j (and thus also from j to i).
- No bias and assume $w_{jj} = 0$ for all $j \in N$.

Boltzmann machine

Activity: States of neurons initially set to values of $\{-1, 1\}$, i.e. $y_j^{(0)} \in \{-1, 1\}$ for $j \in N$.

In the step $t + 1$ do the following:

- Choose a neuron $j \in N$ randomly with the uniform probability.
- Compute the inner potential of j:

$$\xi_j^{(t)} = \sum_{i \in j_{\leftarrow}}^{n} w_{ji} y_i^{(t)}$$

- Choose $y_j^{(t+1)} \in \{-1, 1\}$ *randomly* so that $\mathbf{P}\left[y_j^{(t+1)} = 1\right] = \sigma(\xi_j^{(t)})$ where

$$\sigma(\xi) = \frac{1}{1 + e^{-2\xi/T(t)}}$$

($T(t)$ is a **temperature** at time t.)

Boltzmann machine

- High temperature $T(t)$ implies that $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx \frac{1}{2}$ and thus the machine behaves almost randomly.
- Low temperature $T(t)$ means that either $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx 1$ or $\mathbf{P}\left[y_j^{(t+1)} = 1\right] \approx 0$ depending on whether $\xi_j^{(t)} > 0$ or $\xi_j^{(t)} < 0$. Then the machine behaves almost deterministically (as the Hopfield network).

Boltzmann machine represents probability

Goal: Construct a network representing a distribution on a set of vectors $\{-1, 1\}^{|N|}$.

Rough idea: Boltzmann machine has states in $\{-1, 1\}^{|N|}$, moves randomly from state to state during computation.

If we let the machine run for sufficiently long time (with a fixed temperature), the *relative frequencies* of visits to states will be independent of the initial state.

We consider these frequencies as probabilities of the states. This gives a probability distribution on $\{-1, 1\}^{|N|}$ represented by the machine.

During learning, a probability distribution on states of $\{-1, 1\}^{|N|}$ will be given, and we adapt weights so that the frequencies match the given probabilities.

Equilibrium

Fix a temperature T (i.e. $T(t) = T$ for $t = 1, 2, \ldots$).

Theorem

For every $\gamma^* \in \{-1, 1\}^{|N|}$ we have that

$$\lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right] = \frac{1}{Z} e^{-E(\gamma^*)/T}$$

where

$$Z = \sum_{\gamma\in\{-1,1\}^{|N|}} e^{-E(\gamma)/T} \qquad E(\gamma) = -\frac{1}{2}\sum_{i,j} w_{ij} y_i^\gamma y_j^\gamma$$

the Boltzmann distribution.

Define $p_N(\gamma^*) := \lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right]$ for every $\gamma^* \in \{-1, 1\}^{|N|}$.

Equilibrium probabilities

Note that

- p_N is a probability distribution on $\{-1, 1\}^{|N|}$ represented by the machine,
- for a state γ^*, we have that $p_N(\gamma^*)$ is the probability of γ^* in the *thermal equilibrium*,
- $p_N(\gamma^*)$ can be estimated by $\mathbf{P}\left[\vec{y}^{(t^*)} = \gamma^*\right]$ for sufficiently large t^*

  That is, in order to compute $p_N(\gamma^*)$ it is sufficient to simulate a computation several times for t^* steps and then compute the relative frequency of stopping in γ^*.
- By Markov chains theory, $p_N(\gamma^*)$ is the long-run frequency of visits to γ^*.

  This gives an alternative procedure for estimating $p_N(\gamma^*)$: Execute the machine for *very* long time, compute the relative frequency of visits to γ^* along the computation.

Boltzmann machine – learning

To be able to capture more probability distributions, we introduce hidden neurons.

Divide N into two disjoint sets:

- **visible** neurons V
- **hidden** neurons H

For $\alpha \in \{-1, 1\}^{|V|}$ denote

$$p_V(\alpha) = \sum_{\beta \in \{-1,1\}^{|H|}} p_N(\alpha, \beta)$$

the probability that the state of visible neurons in the thermal equilibrium is α.

Our goal is to adapt weights so that p_V corresponds to a given probability distribution on $\{-1, 1\}^{|V|}$.

Boltzmann machine – learning

Learning:
Let p_d be a probability distribution on the states of visible neurons, i.e. on $\{-1, 1\}^{|V|}$.

The distribution p_d can be determined by a sequence of training examples:

$$\mathcal{T} = \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m$$

then

$$p_d(\alpha) = \#(\alpha, \mathcal{T})/m$$

here $\#(\alpha, \mathcal{T})$ is the number of occurrences of α in $\mathcal{T}$.

Our goal is to find a configuration of the network W such that $p_V \approx p_d$.

Boltzmann machine – learning

A suitable measure of difference between probability distributions p_V and p_d is relative entropy weighted by probabilities of states (Kullback-Leibler divergence):

$$\mathcal{E}(W) = \sum_{\alpha \in \{-1,1\}^{|V|}} p_d(\alpha) \ln \frac{p_d(\alpha)}{p_V(\alpha)}$$

For p_d given by a training set $\mathcal{T} = \vec{x}_1, \vec{x}_2, \dots, \vec{x}_m$ we have that

minimizing $\mathcal{E}(W)$ is equivalent to maximizing likelihood of $\mathcal{T}$.

Boltzmann machine – learning

Minimize $\mathcal{E}(\vec{w})$ using gradient descent, i.e. compute a sequence of weight matrices: $W^{(0)}, W^{(1)}, \dots$

- initialise $W^{(0)}$ randomly, close to 0
- in step $t+1$ compute $W^{(t+1)}$ as follows:

$$W_{ji}^{(t+1)} = W_{ji}^{(t)} + \Delta W_{ji}^{(t)}$$

where

$$\Delta W_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial \mathcal{E}}{\partial w_{ji}}(W^{(t)})$$

is the update of the weight w_{ji} in the step $t+1$ and $0 < \varepsilon(t) \leq 1$ is the learning rate in the step $t+1$.

It remains to compute $\frac{\partial \mathcal{E}}{\partial w_{ji}}(W)$.

Boltzmann machine – learning

For sufficiently large t^* (i.e. in thermal equilibrium) we have

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} \approx -\frac{1}{T}\left(\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed} - \left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}\right)$$

- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium assuming that values of visible neurons are **fixed** at the beginning of computation according to p_d.
- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium (no values fixed).

Thus

$$\begin{aligned}\Delta w_{ji}^{(\ell)} &= -\varepsilon(\ell) \cdot \frac{\partial \mathcal{E}}{\partial w_{ji}}(W^{(\ell-1)}) \\ &= \frac{\varepsilon(\ell)}{T}\left(\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed} - \left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}\right)\end{aligned}$$

Boltzmann machine – learning

Compute $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed}$ as follows:

- Let $\mathcal{Y} := 0$ and do the following q times:

  1. choose $\alpha \in \{-1, 1\}^{|V|}$ randomly according to p_d,
  2. fix values of visible neurons to α and *do not* update them throughout the remaining steps 3. and 4.,
  3. simulate t^* steps, now the current values of neurons j and i are $y_j^{(t^*)}$ and $y_i^{(t^*)}$, respectively,
  4. add $y_j^{(t^*)} y_i^{(t^*)}$ to $\mathcal{Y}$.

- For sufficiently large q, the value $\mathcal{Y}/q$ will be a good estimate of $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed}$.

$\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ can be estimated similarly, the only difference is that the steps 1. and 2. are omitted.

Boltzmann machine – learning

For completeness, the analytic version:

$$\left\langle y_i^{(t^*)} y_j^{(t^*)} \right\rangle_{fixed} =$$

$$= \sum_{\alpha \in \{-1,1\}^{|V|}} p_d(\alpha) \sum_{\beta \in \{-1,1\}^{|S|}} \frac{p_N(\alpha,\beta)}{p_V(\alpha)} y_j^{\alpha\beta} y_i^{\alpha\beta}$$

here $y_j^{\alpha\beta}$ is the output of the neuron j in the state (α,β).

$$\left\langle y_i^{(t^*)} y_j^{(t^*)} \right\rangle_{free} = \sum_{\gamma \in \{-1,1\}^{|N|}} p_N(\gamma) y_j^{\gamma} y_i^{\gamma}$$

Restricted Boltzmann machine (RBM)

Architecture:

- Neural network with cycles and symmetric connections, neurons divided into two disjoint sets:
  - V - visible
  - H - hidden

  Connections: $V \times S$ (complete bipartite graph)
- N is a *set of all neurons*.
- Denote by ξ_j the inner potential and by y_j the output (i.e. state) of neuron j.
  State of the machine: $\vec{y} \in \{0, 1\}^{|N|}$.
- Denote by $w_{ji} \in \mathbb{R}$ the weight of the connection from i to j (and thus also from j to i).
- Consider bias: w_{j0} is the weight between j and a neuron 0 whose value y_0 is always 1.

RBM – activity

Activity: States of neurons initially set to values of $\{0, 1\}$, i.e. $y_j^{(0)} \in \{0, 1\}$ for $j \in N$.

In the step $t + 1$ do the following:

- t even: randomly choose new values of all hidden neurons, for every $j \in H$

$$\mathbf{P}\left[y_j^{(t+1)} = 1\right] = 1 \Big/ \left(1 + \exp\left(-w_{j0} - \sum_{i \in V} w_{ji} y_i^{(t)}\right)\right)$$

- t odd: randomly choose new values of all visible neurons, for every $j \in V$

$$\mathbf{P}\left[y_j^{(t+1)} = 1\right] = 1 \Big/ \left(1 + \exp\left(-w_{j0} - \sum_{i \in H} w_{ji} y_i^{(t)}\right)\right)$$

Thermal equilibrium

Fix a temperature T (i.e. $T(t) = T$ for $t = 1, 2, \ldots$).

Theorem

For every $\gamma^* \in \{0,1\}^{|N|}$ we have that

$$\lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right] = \frac{1}{Z} e^{-E(\gamma^*)/T}$$

where

$$Z = \sum_{\gamma \in \{0,1\}^{|N|}} e^{-E(\gamma)/T}$$

and

$$E(\gamma) = -\sum_{i\in V, j\in H} w_{ji} y_j^\gamma y_i^\gamma - \sum_{i \in V} w_{i0} y_i^\gamma - \sum_{j\in H} w_{j0} y_j^\gamma$$

Define $p_N(\gamma^*) := \lim_{t\to\infty} \mathbf{P}\left[\vec{y}^{(t)} = \gamma^*\right]$ for every $\gamma^* \in \{0,1\}^{|N|}$.

RBM – learning

Learning:
Let p_d be a probability distribution on states of visible neurons, i.e. on $\{0,1\}^{|V|}$.

Our goal is to find a configuration of the network W such that $p_V \approx p_d$.

A suitable measure of difference between probability distributions p_V and p_d is relative entropy weighted by probabilities of states (Kullback-Leibler divergence):

$$\mathcal{E}(W) = \sum_{\alpha \in \{0,1\}^{|V|}} p_d(\alpha) \ln \frac{p_d(\alpha)}{p_V(\alpha)}$$

RBM – learning

Minimize $\mathcal{E}(\vec{w})$ using gradient descent, i.e. compute a sequence of weight matrices: $W^{(0)}, W^{(1)}, \ldots$

- initialise $W^{(0)}$ randomly, close to 0
- in step $t+1$ compute $W^{(t+1)}$ as follows:

$$W_{ji}^{(t+1)} = W_{ji}^{(t)} + \Delta W_{ji}^{(t)}$$

where

$$\Delta W_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial \mathcal{E}}{\partial w_{ji}}(W^{(t)})$$

is the update of the weight w_{ji} in the step $t+1$ and $0 < \varepsilon(t) \leq 1$ is the learning rate in the step $t+1$.

It remains to compute $\frac{\partial \mathcal{E}}{\partial w_{ji}}(W)$.

RBM – learning

For sufficiently large t^* (i.e. in thermal equilibrium) we have

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} \approx -\frac{1}{T}\left(\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed} - \left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}\right)$$

- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{fixed}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium assuming that values of visible neurons are **fixed** at the beginning of computation according to p_d.
- $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ is the expected value of $y_j^{(t^*)} y_i^{(t^*)}$ in the thermal equilibrium (no values fixed).

Problem: Computation of $\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ takes long time.

$\left\langle y_j^{(t^*)} y_i^{(t^*)} \right\rangle_{free}$ can be estimated with $\left\langle y_j y_i \right\rangle_{recon}$, the expectation of $y_j^{(3)} y_i^{(3)}$ when values of visible neurons chosen by p_d.

RBM – learning

Thus

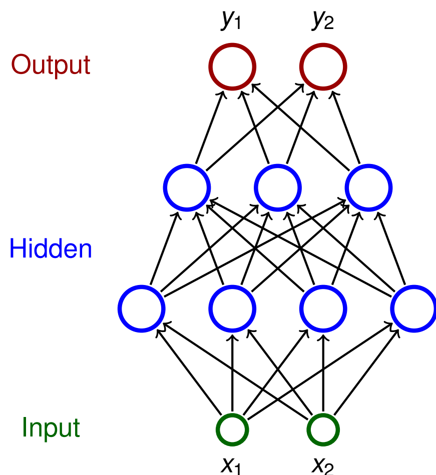
$$\Delta w_{ji}^{(t)} = \varepsilon(t) \cdot \left(\left\langle y_j y_i \right\rangle_{fixed} - \left\langle y_j y_i \right\rangle_{recon} \right)$$

- Compute $\left\langle y_j y_i \right\rangle_{fixed}$ as follows: Let $\mathcal{Y} := 0$ and repeat the following q times:
  - **fix** values of visible neurons randomly by p_d
  - simulate one step of computation, add $y_j y_i$ to $\mathcal{Y}$

  For a suitable q we have that $\mathcal{Y}/q$ estimates $\left\langle y_j y_i \right\rangle_{fixed}$ well.
- Compute $\left\langle y_j y_i \right\rangle_{recon}$ as follows: Let $\mathcal{Y} := 0$ and repeat q times:
  - choose initial values of visible neurons by p_d
  - simuate three steps, add $y_j y_i$ to $\mathcal{Y}$
    (i.e. compute values of hidden neurons, then of visible ones (reconstruction of the input) and then of hidden neurons)

  For a suitable q we have that $\mathcal{Y}/q$ estimates $\left\langle y_j y_i \right\rangle_{recon}$ well.

Deep MLP



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i+1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

Deep MLP

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)

Deep MLP – activity

- inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- activation function σ_j for neuron j (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

A deep MLP is evaluated layer-wise, for each $j \in Y$ we have that $y_j(\vec{w}, \vec{x})$ is the value of the output neuron j after evaluating the network with weights $\vec{w}$ and input $\vec{x}$.

Deep MLP – learning

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function – mean square error (for example):**

$$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

Other errors such as cross-entropy possible.

Convolutional networks – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

where

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \frac{1}{|T|} \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here T is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

Why deep networks

... if one hidden layer is able to represent an arbitrary (reasonable) function?

- One hidden layer may be very inefficient, i.e. huge amount of neurons may be needed. One can show that
  - the number of hidden neurons may be exponential w.r.t. the dimension of the input,
  - networks with multiple layers may be exponentially more succinct as opposed to single hidden layer.

... ok, so let's try to teach deep networks ... using backpropagation?

Problems:

- Gradient may vanish/explode when backpropagated through many layers.
- Deep networks (with many neurons) overfit very easily.

Deep MLP - vanishing gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma'_j(\xi_j) \cdot y_i$$

and for every $j \in Z \setminus X$ holds

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{pro } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma'_r(\xi_r) \cdot w_{rj} \qquad \text{pro } j \in Z \setminus (Y \cup X)$$

$\sigma'_r(\xi_r) \cdot w_{rj}$ **is less than one** for standard logistic sigmoid and weights initialized close to 0.

Deep MLP – pretraining

Assume k layers. Denote

- W_i the weight matrix between layers $i-1$ and i
- F_i function computed by the "lower" part of the MLP consisting of layers $0, 1, \ldots, i$

  F_1 is a function which consists of the input and the first hidden layer (which is now considered as the output layer).

Crucial observation: For every i, the layers $i-1$ and i together with the matrix W_i can be considered as a RBM (assume $T = 1$).

Denote such a RBM as B_i.

Deep MLP – pretraining

For now, consider only input vectors $\vec{x}_1, \ldots, \vec{x}_p$ where $\vec{x}_k \in \{0,1\}^n$ for all $k = 1, \ldots, p$.

- **unsupervised pretraining:** Gradually, for every $i = 1, \ldots, k$, train RBM B_i on randomly selected inputs from the training set:

  $$F_{i-1}(\vec{x}_1), \ldots, F_{i-1}(\vec{x}_p)$$

  using the training algorithm for RBM (here $F_0(\vec{x}_i) = \vec{x}_i$).

  (Thus B_i learns from training samples **transformed by the already pretrained layers** $0, \ldots, i-1$**)**

We obtain a *deep belief network* $\mathcal{D}$ representing a distribution given by $\vec{x}_1, \ldots, \vec{x}_p$.

(Recall that in such a distribution the probability of a given $\vec{x}$ is equal to the relative frequency of $\vec{x}$ in $\vec{x}_1, \ldots, \vec{x}_p$.)

Deep belief network

The network $\mathcal{D}$ can be used to sample from the distribution as follows:

- Simulate the topmost RBM for some steps (ideally to thermal equilibrium), this gives values of neurons in the two topmost layers.
- Propagate the values downwards by always simulating one step of the corresponding RBM. That is,
  - you have already computed values of neurons in layers k and $k-1$.
  - To compute values of neurons in the layer $k-2$, simulate one step of RBM B_{k-1}, that is sample values of neurons in the layer $k-2$ using RBM dynamics of B_{k-1} with values of the layer $k-1$ fixed.
  - Similarly, compute values of $k-3$ by simulating B_{k-2} ... etc.
  - ... finally obtain values of input neurons.
- Probability with which a concrete input $\vec{x}$ is sampled by the above procedure is the probability of $\vec{x}$ in the distribution represented by $\mathcal{D}$.

Deep MLP – training with pretraining

Now consider supervised learning with a training set:
$\mathcal{T} = \left\{ \left(\vec{x}_k, \vec{d}_k \right) \;\middle|\; k = 1, \ldots, p \right\}$.
Still assume that $\vec{x}_k \in \{0, 1\}^n$.

- **unsupervised pretraining:** Gradually, for every $i = 1, \ldots, k$, train RBM B_i on randomly selected inputs from the training set:

  $$F_{i-1}(\vec{x}_1), \ldots, F_{i-1}(\vec{x}_p)$$

  using the training algorithm for RBM (here $F_0(\vec{x}_i) = \vec{x}_i$).

  (Thus B_i learns from training samples **transformed by the already pretrained layers** $0, \ldots, i-1$**)**

  Obtain $\mathcal{D}$.

- Add one (or more) layer to the top of $\mathcal{D}$ and consider the result to be MLP.

  (i.e. forget the RBM dynamics and start considering the network as MLP with sigmoidal activations).

- **supervised fine-tuning:** Train in supervised mode (on the training set $\mathcal{T}$) using e.g. gradient descent + backprop.

Application – dimensionality reduction

- Dimensionality reduction: A mapping R from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.
- Standard method: PCA (there are many linear as well as non-linear variants)

Reconstruction – PCA

1024 pixels compressed to 100 dimensions (i.e. 100 numbers).

Autoencoders

Dimensionality reduction using MLP.

- Consider MLP $n - m - n$ where $m << n$.
- The same vector on the input as well as output.
- Dimensionality reduction:
  - Encoding: Compute values of hidden neurons.
  - Reconstruction: Compute values of output neurons given values of hidden neurons.

Can also be used for compression (in communication).

One can show that if linear neurons are used, the method implements PCA.

Autoencoder – historical implementation

Architecture: MLP 64 – 16 – 64

Activity: activation function: hyperbolic tangens with limits -1 and 1

Learning:
Data:

- Images 256×256, 8 bits per pixel.
- Samples: input and output is a frame 8×8, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

Learning:

- Backpropagation
- Learning rate: 0.01 for hidden, 0.1 pro output
- trained in 50 000 - 100 000 iterations

The goal was to compress images to smaller data size.

Dimensionality reduction – compression

(A)

(B)

(C)

(D)

A frame 8×8 passes through the image 256×256 (no overlap)

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

Dimensionality reduction – compression

(A)

(B)

(C)

(D)

New image (trained on the previous one):

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

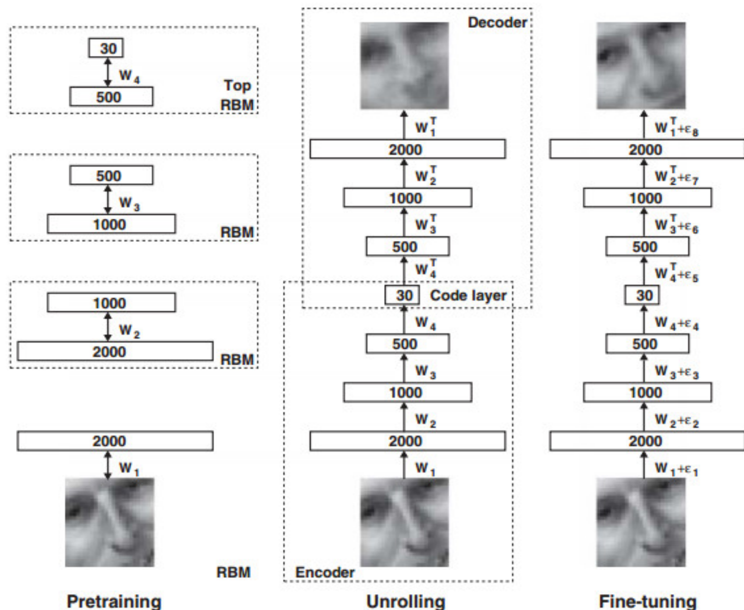
Deep MLP – dimensionality reduction

Hinton, G. E., Osindero, S. and Teh, Y. (2006)
A fast learning algorithm for deep belief nets.
Neural Computation, 18, pp 1527-1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006)
Reducing the dimensionality of data with neural networks.
Science, Vol. 313. no. 5786, pp. 504 - 507, 28 July 2006.

This basically started all the deep learning craze ...

Deep MLP – dimensionality reduction



Images – pretraining

- **Data:** 165 600 black-white images, 25×25, mean intensity 0, variance 1.

  Images obtained from Olivetti Faces database of images 64×64 using standard transformations.
- 103 500 training set, 20 700 validation, 41 400 test
- **Network:** 2000-100-500-30, training using layered RBM.

Notes:

Training of the lowest layer (2000 neurons): Values of pixels distorted using Gaussian noise, low learning rate: 0.001, 200 iterations

Training all hidden layers: Values of neurons are binary.

Training of output layer: Values computed directly using the sigmoid activation functions + noise. That is, values of output neurons are from the interval $[0, 1]$.

Images – fine-tuning

- Stochastic activation substituted with deterministic.
  That is the value of hidden neurons is not chosen randomly but directly computed by application of sigmoid on the inner potential (this gives the mean activation).
- Backpropagation.
- Error function: cross-entropy

$$-\sum_i p_i \ln \hat{p}_i - \sum_i (1 - p_i) \ln(1 - \hat{p}_i)$$

here p_i is the intensity of i-th pixel of the input and $\hat{p}_i$ of the reconstruction.

Results

1. Original
2. Reconstruction using deep networks (reduction to 30-dim)
3. Reconstruction using PCA (reduction to 30-dim)