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)

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!**

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.

FAQ

Q: Why English?

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.

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?

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.)

Machine learning in general

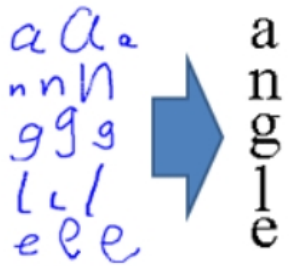
- Machine learning = construction of systems that may learn their functionality from data

  (... 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

Machine learning in general

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

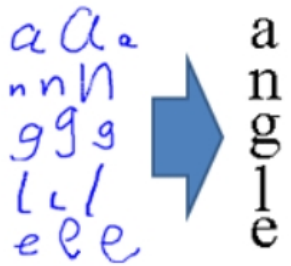
  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labelled" emails
    - consequently is able to distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labelled by their correct meaning
    - consequently is able to recognize text



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

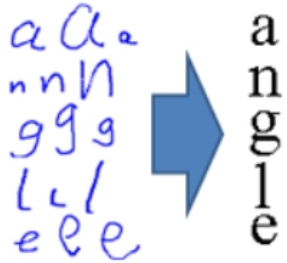


  - ...
  - and lots of much much more sophisticated applications ...

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.

Machine learning in general

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

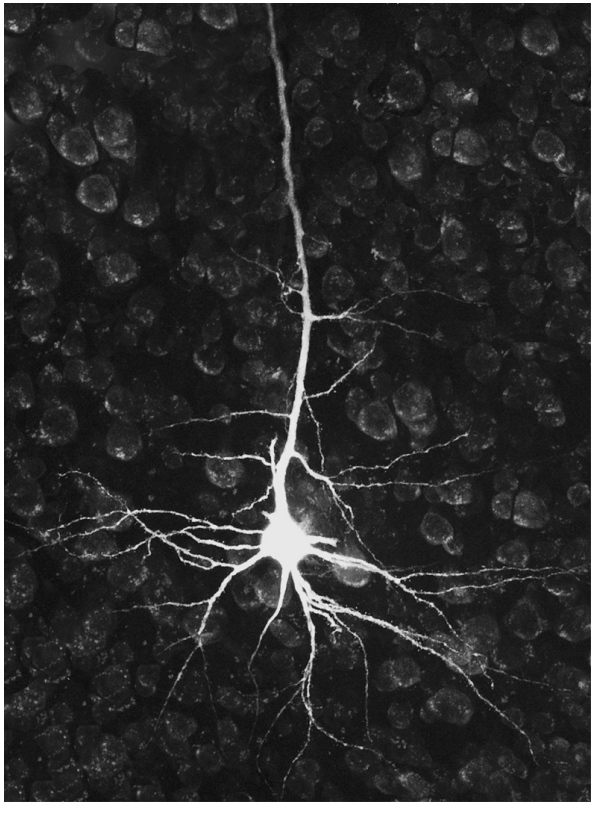
There are many types of models:

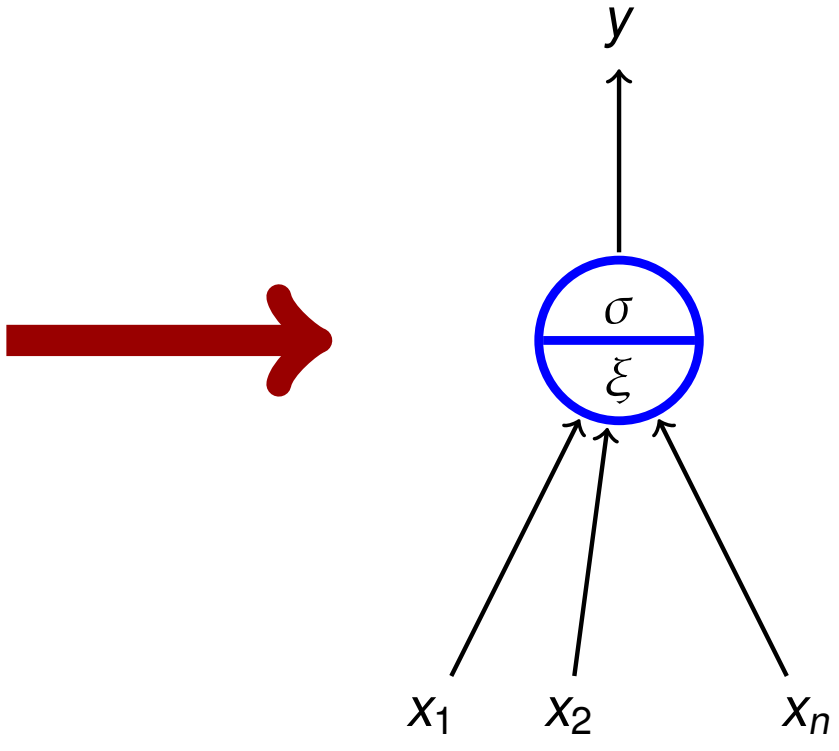
- decision trees
- support vector machines
- hidden Markov models
- Bayes networks and other graphical models
- **neural networks**
- ...

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?
  - ...

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).

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

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

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

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

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)

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)

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)

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)

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, Keras)

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.

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).

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.

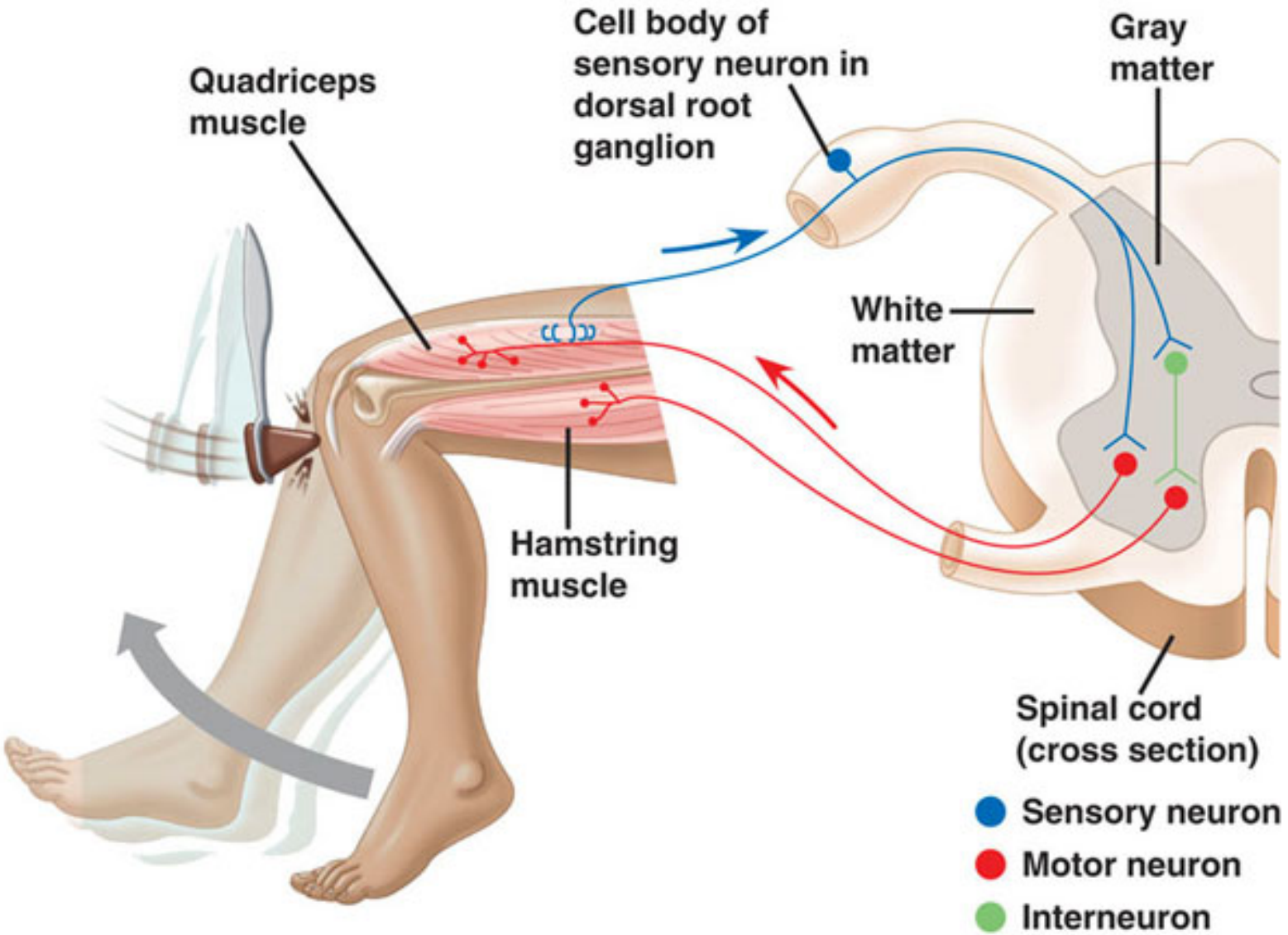
Biological neural network

- 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

Summation

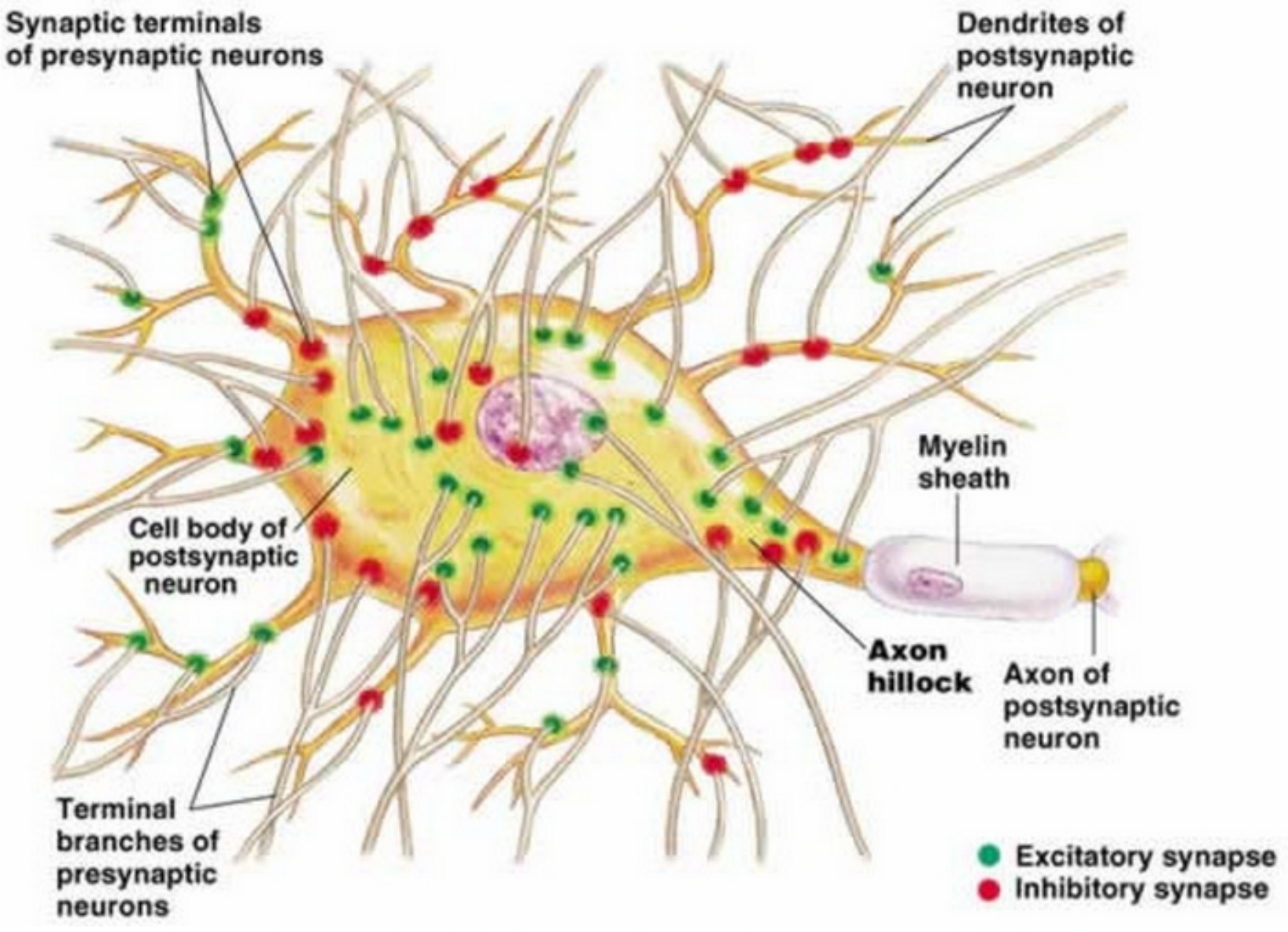
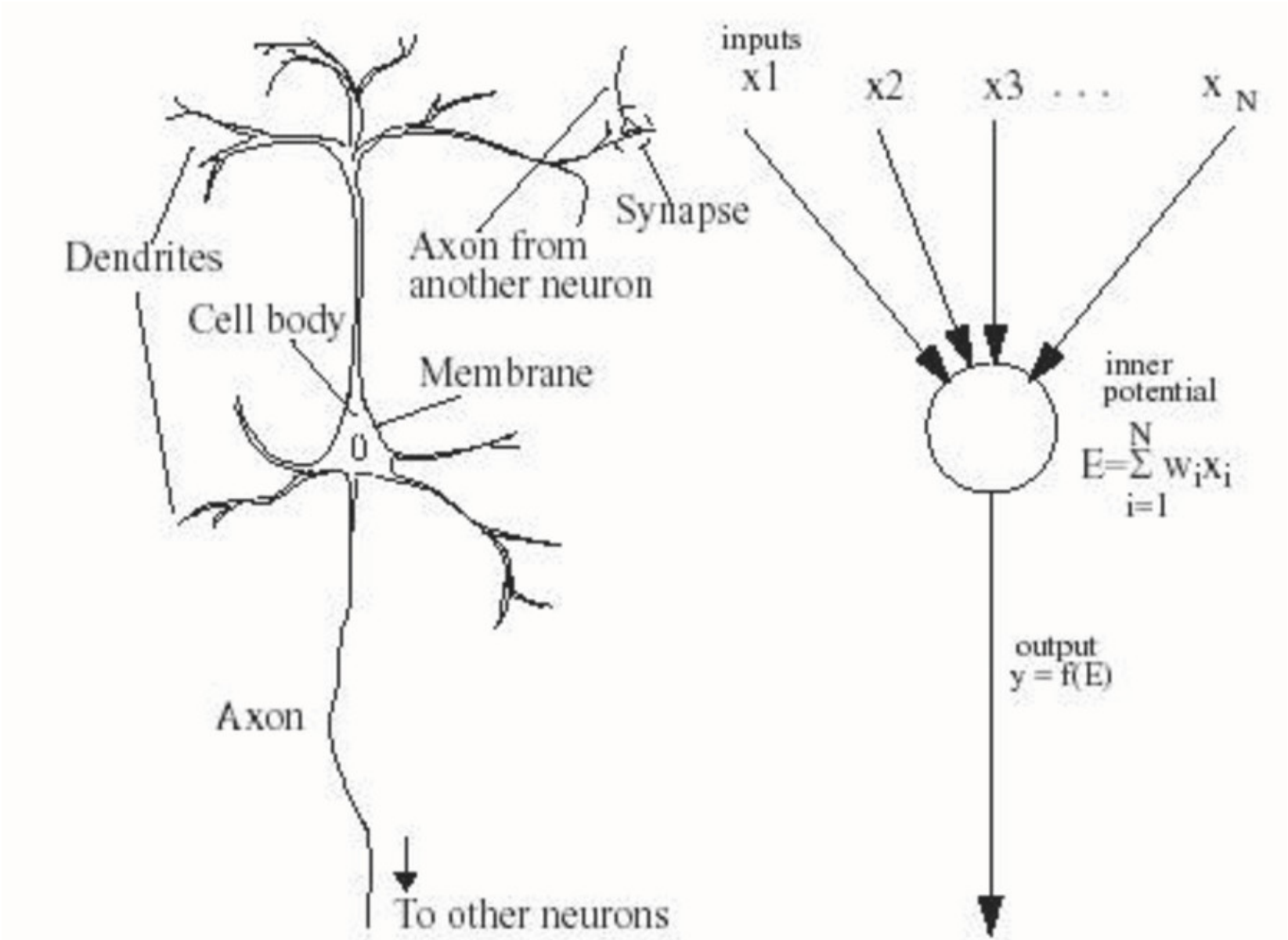
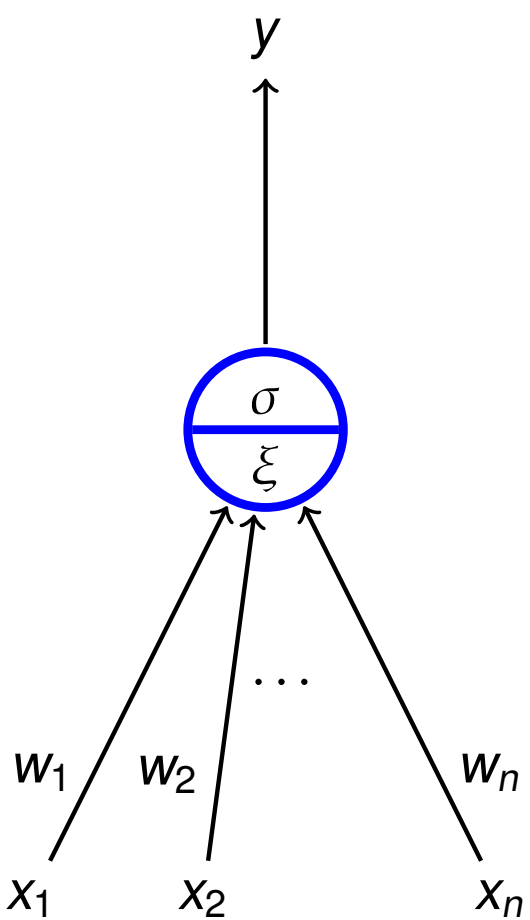


Figure 48.11(a), page 972, Campbell's *Biology, 5th Edition*

Biological and Mathematical neurons

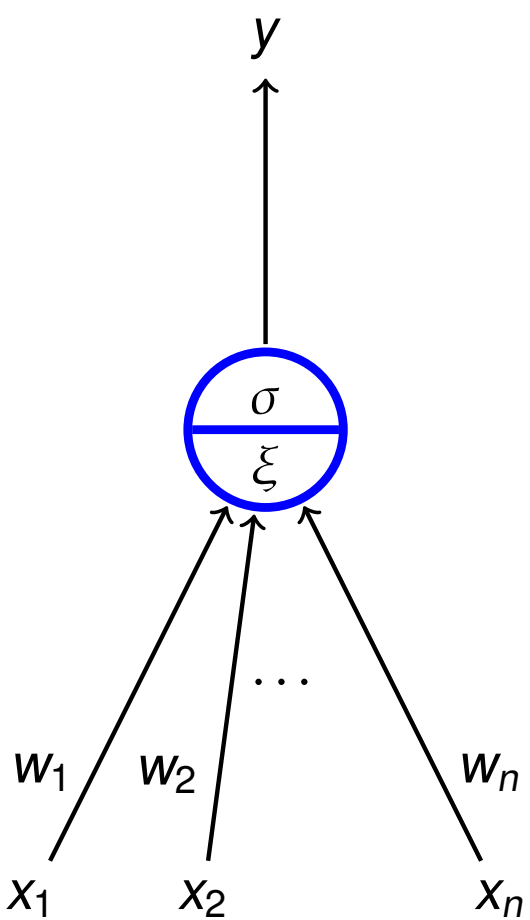


Formal neuron (without bias)



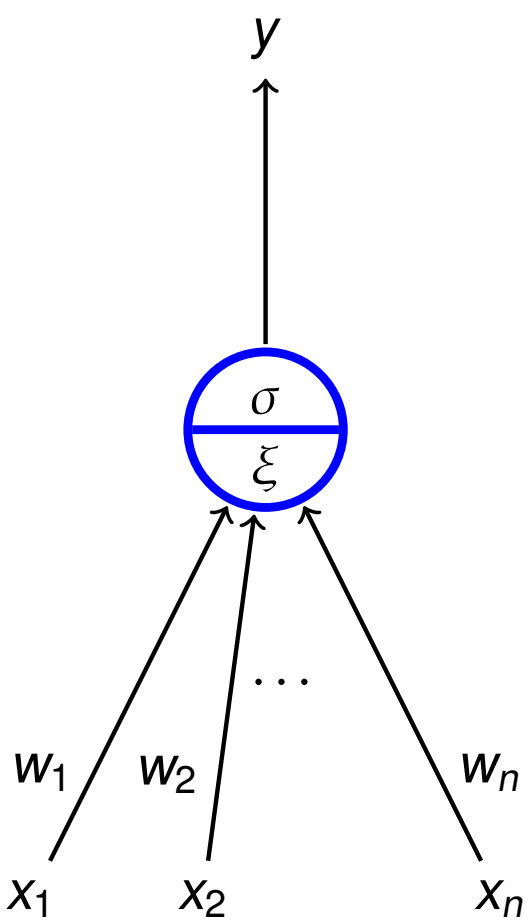
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**

Formal neuron (without bias)



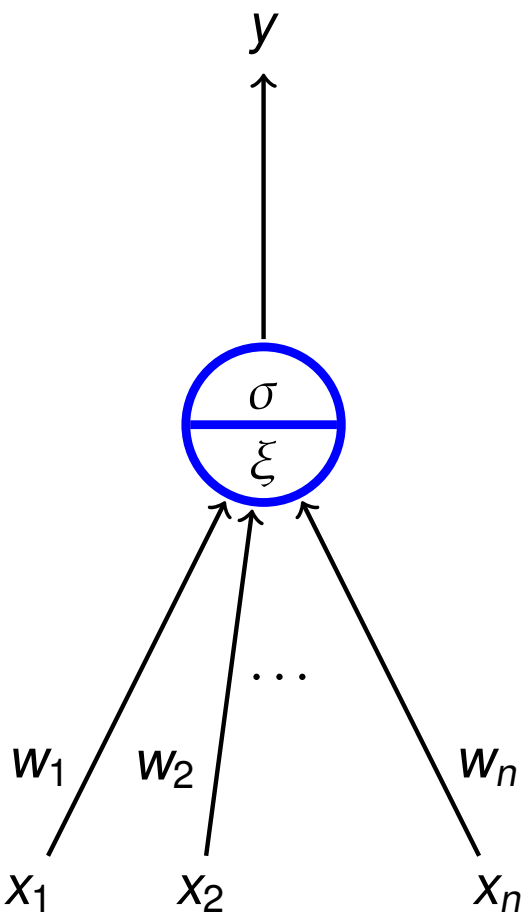
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

Formal neuron (without bias)



- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**; almost always $\xi = \sum_{i=1}^{n} w_i x_i$

Formal neuron (without bias)



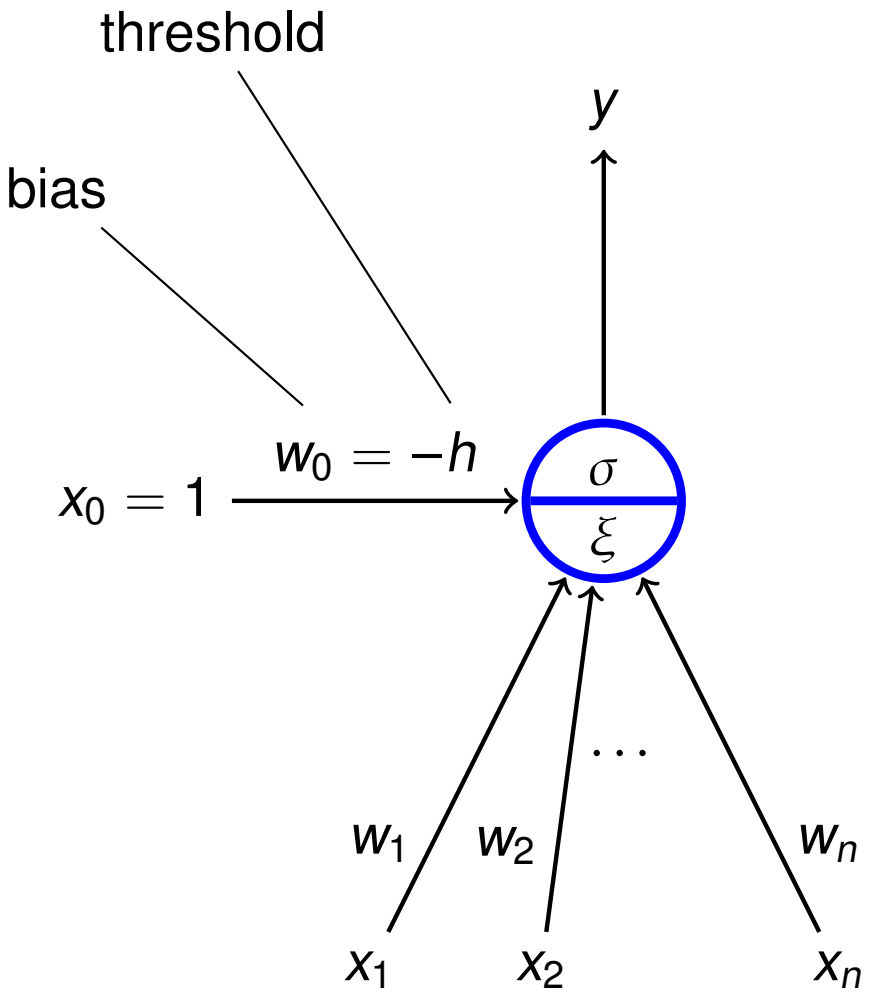
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**; almost always $\xi = \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$ where σ is an **activation function**; e.g. a *unit step function*

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

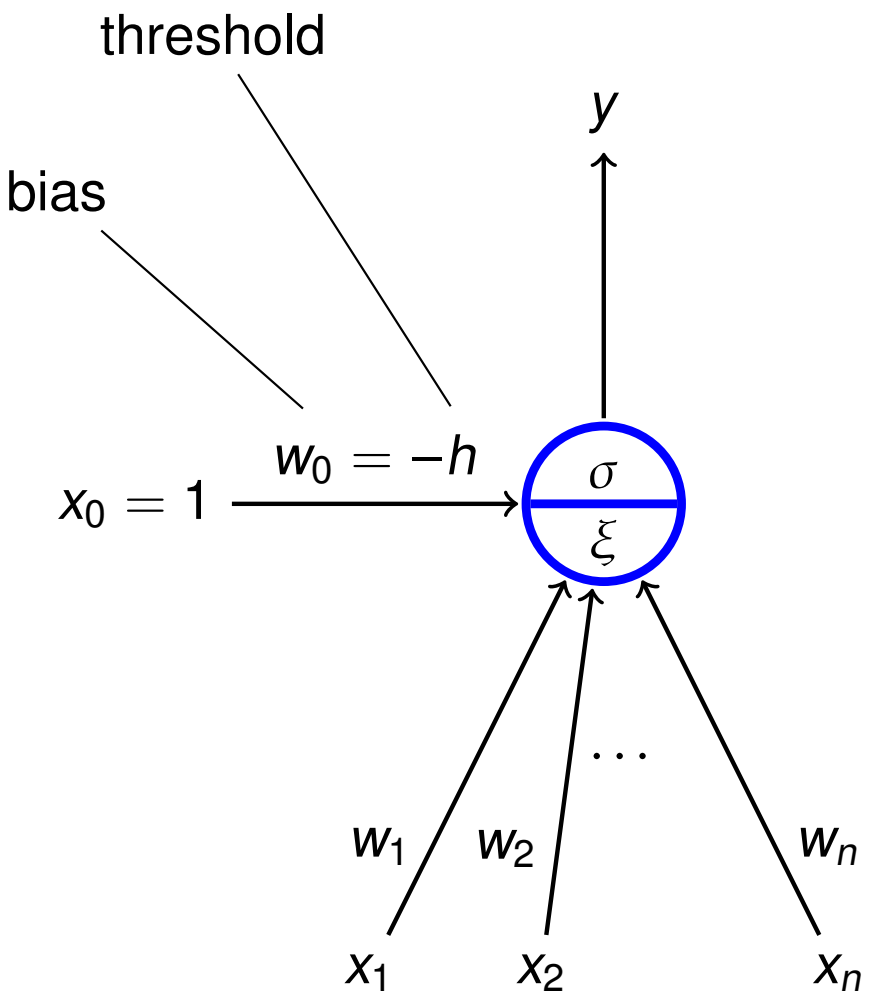
where $h \in \mathbb{R}$ is a *threshold*.

Formal neuron (with bias)

- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**

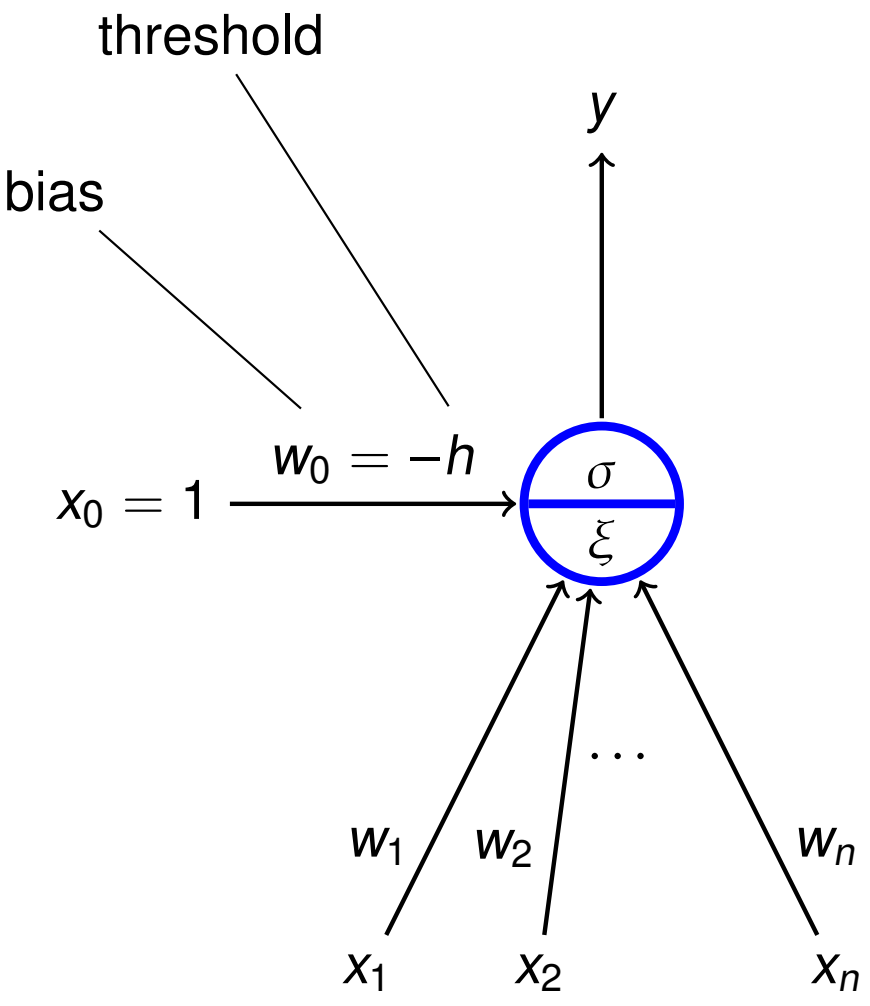


Formal neuron (with bias)



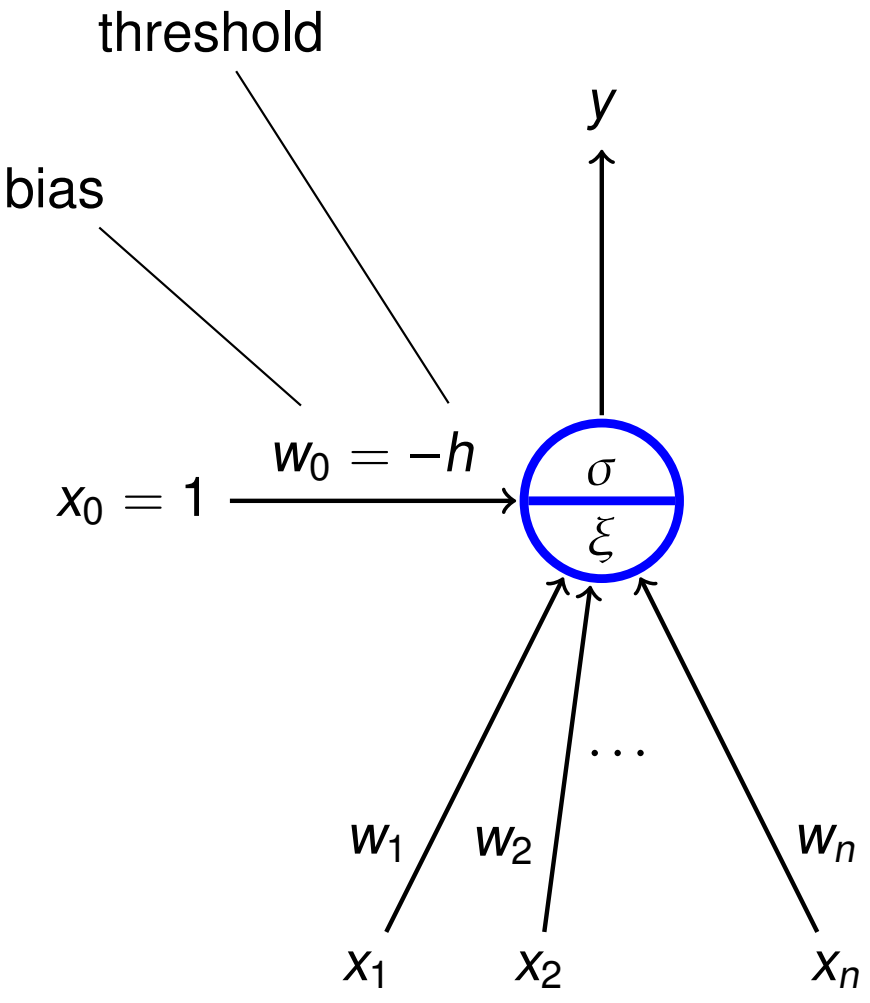
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

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

Formal neuron (with bias)



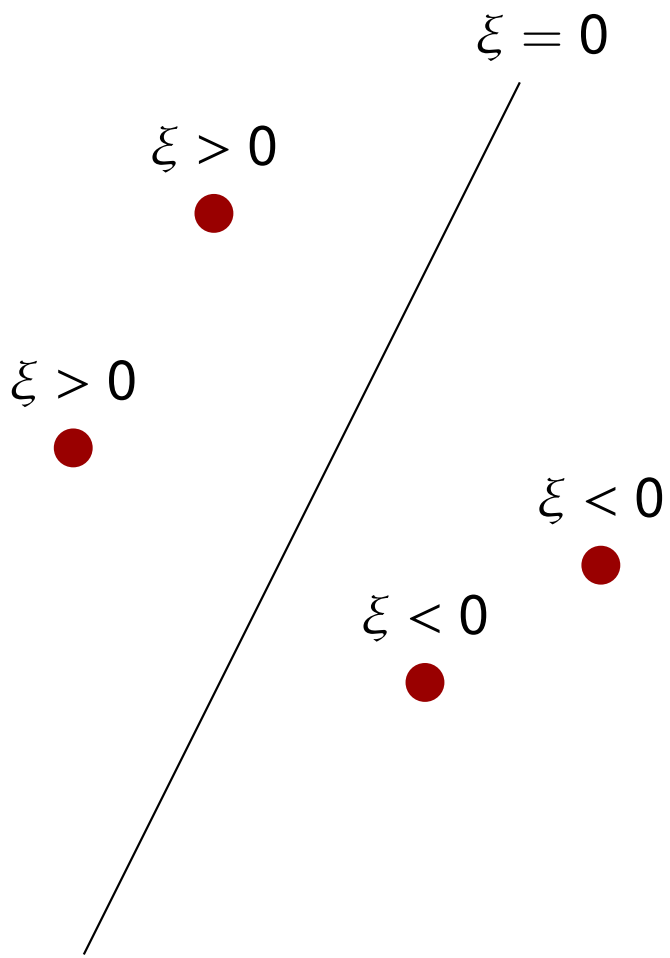
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**; almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$ where σ is an **activation function**;

  e.g. a *unit step function*

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

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

Neuron and linear separation



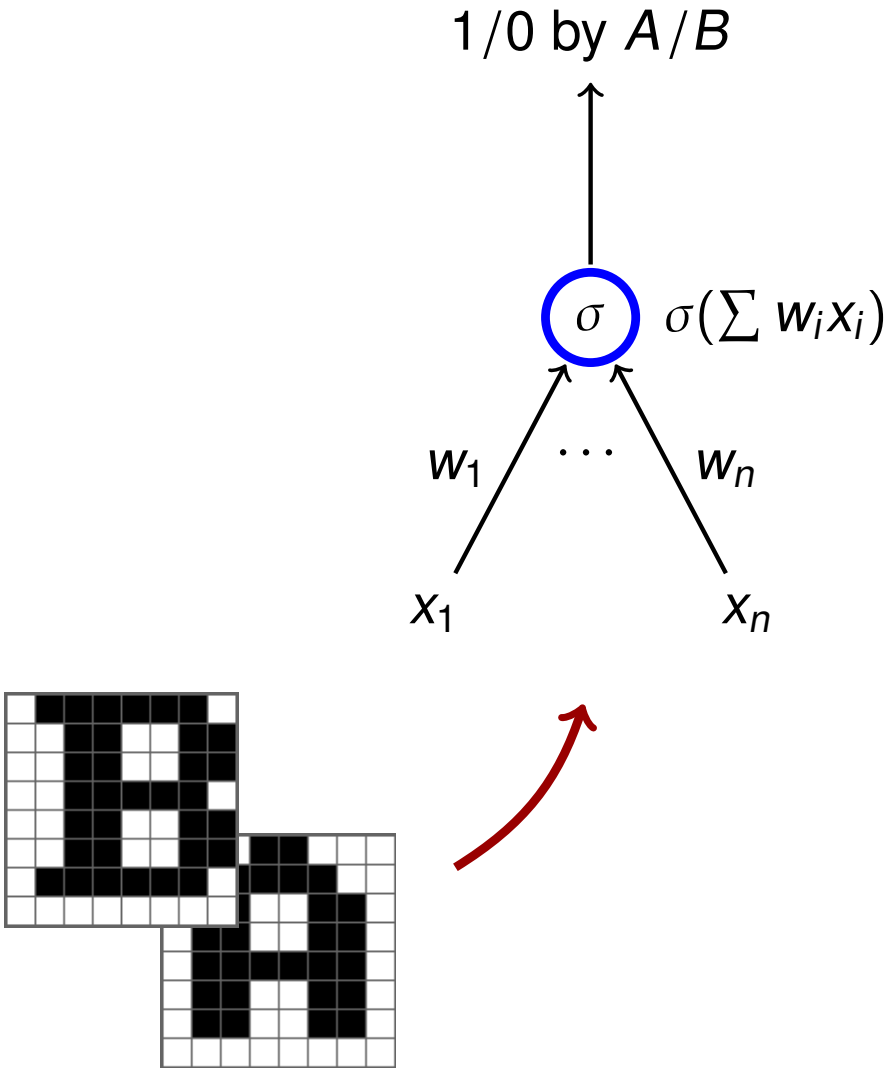
- inner potential

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

determines a separation hyperplane in the n-dimensional **input space**

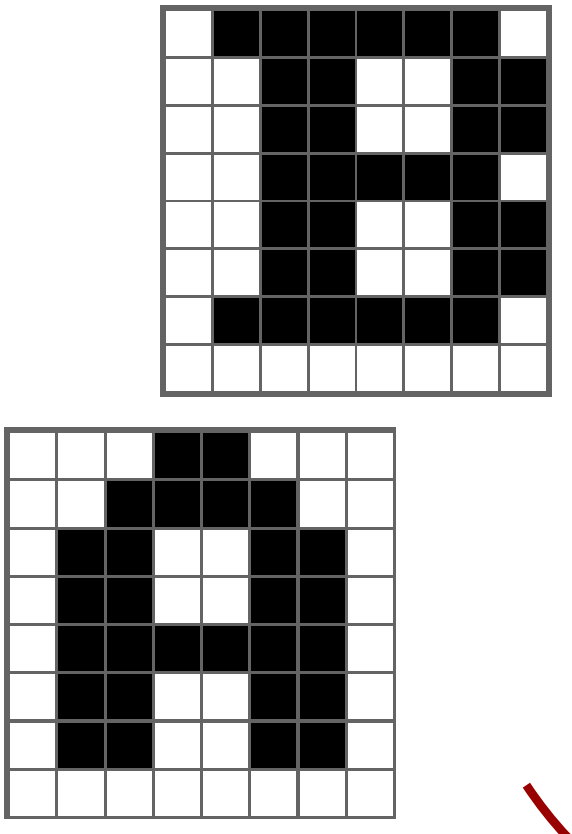
- in 2d line
- in 3d plane
- ...

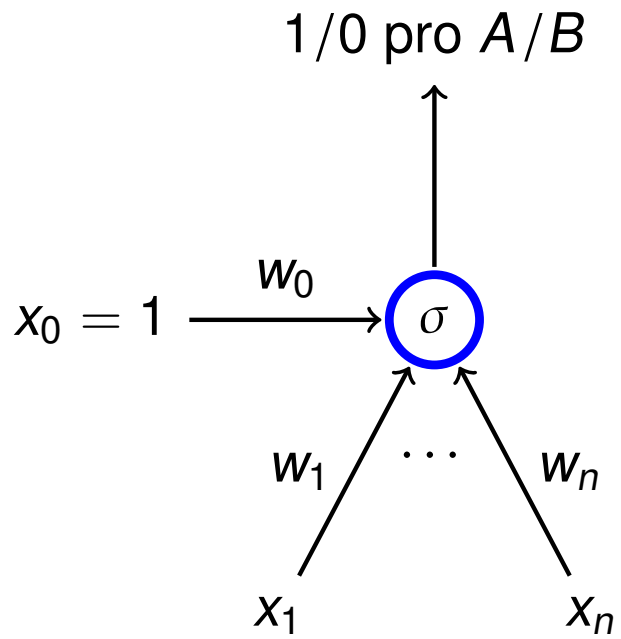
Neuron and linear separation



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

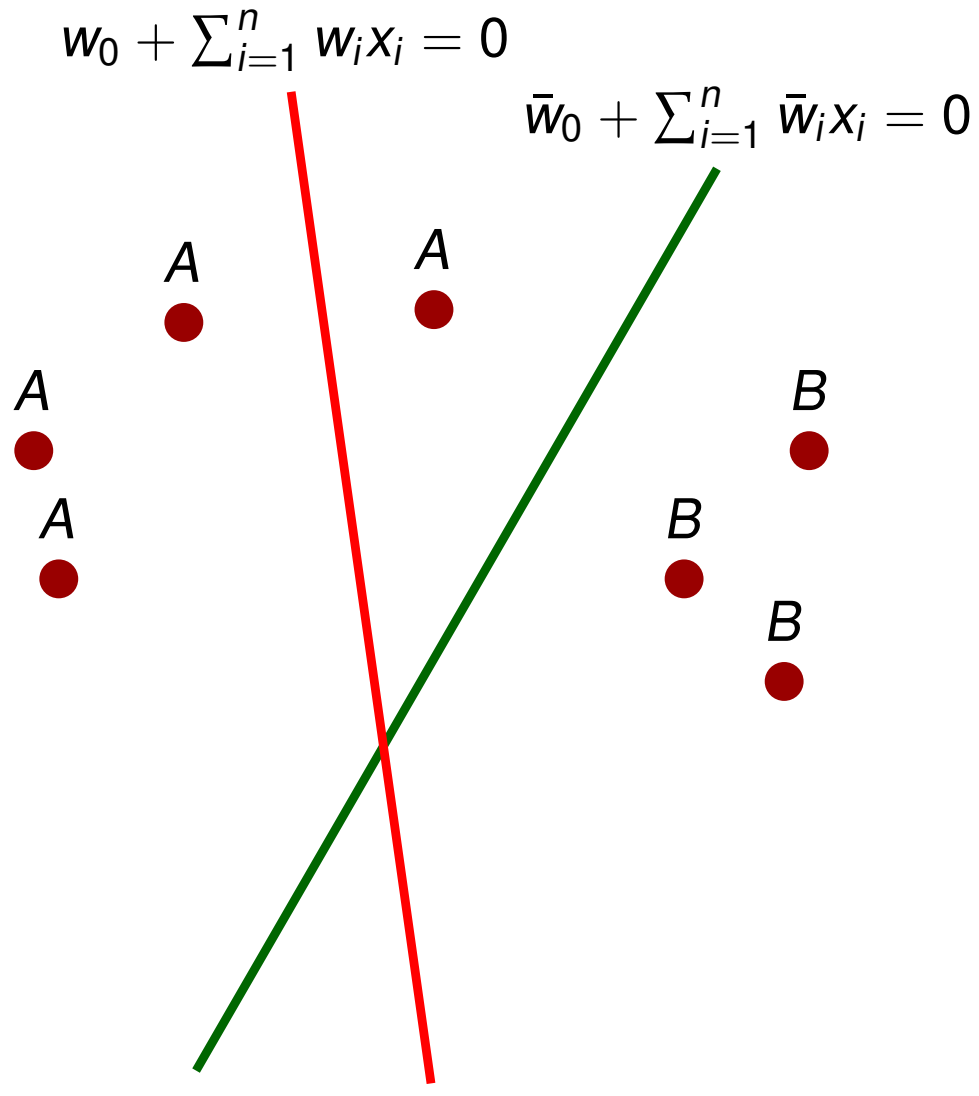
Neuron and linear separation





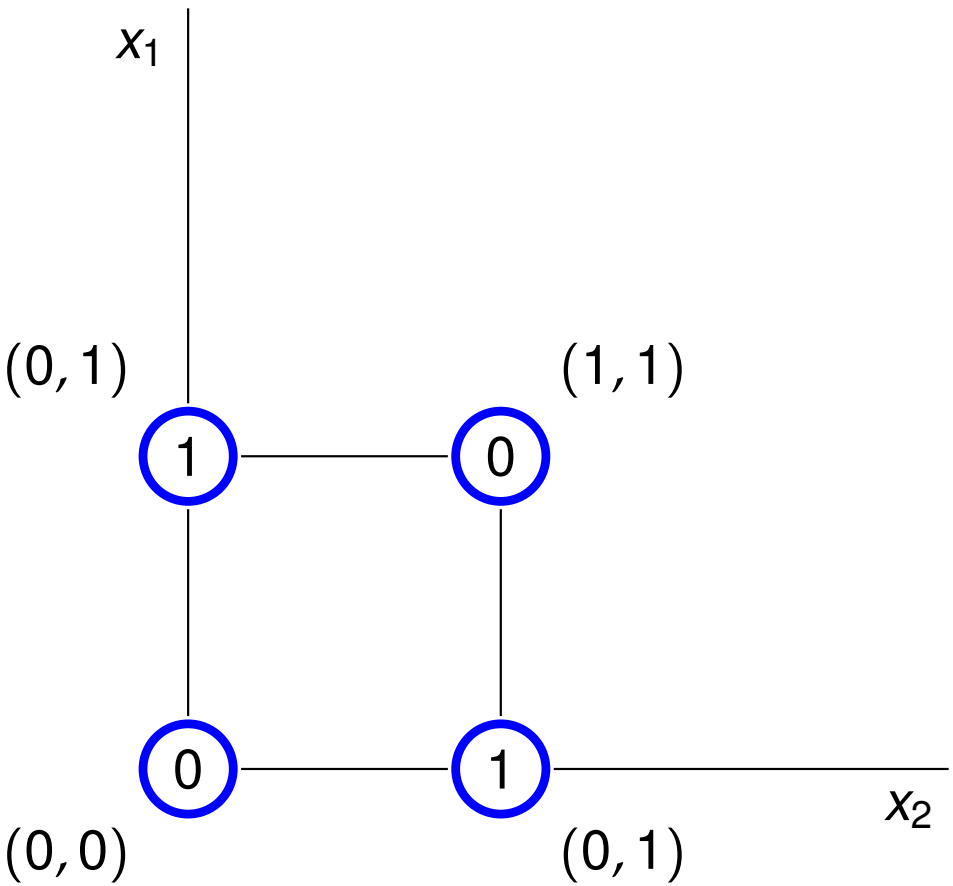
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

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.

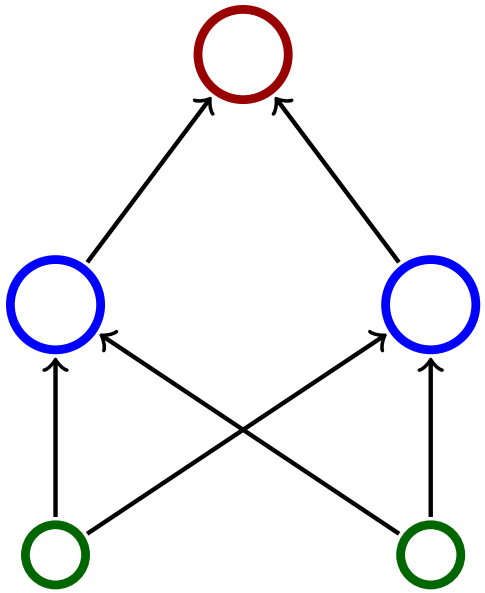
Architecture

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

We distinguish several categories of neurons:

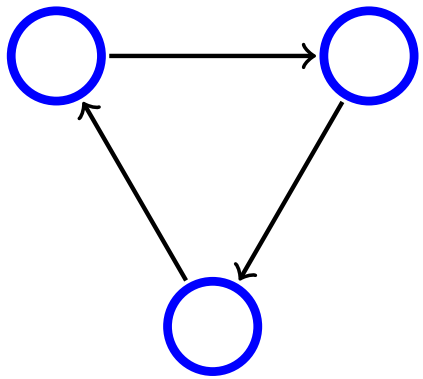
- Output neurons
- Hidden neurons
- Input neurons

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



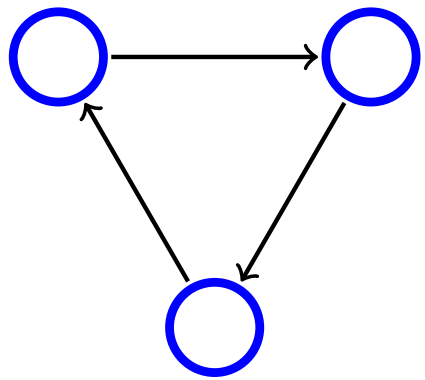
Architecture – Cycles

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

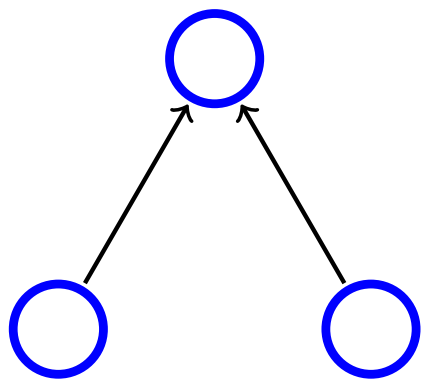


Architecture – Cycles

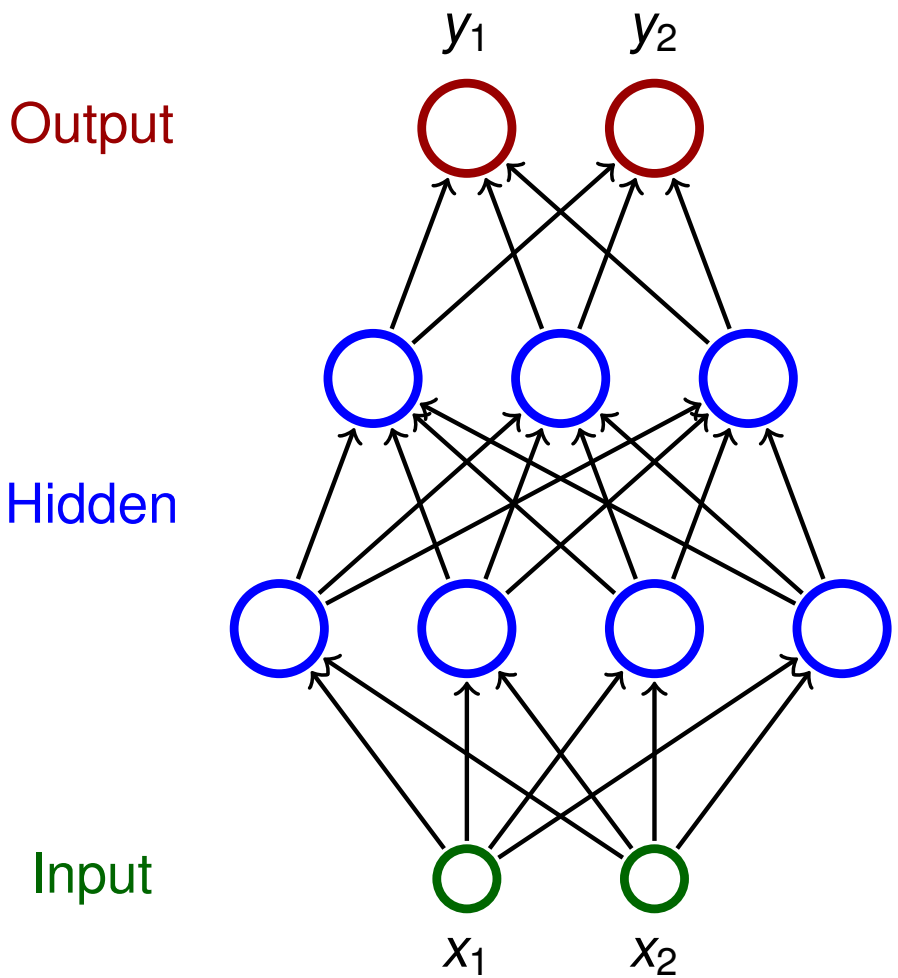
- A network is **cyclic** (recurrent) if its architecture contains a directed cycle.



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

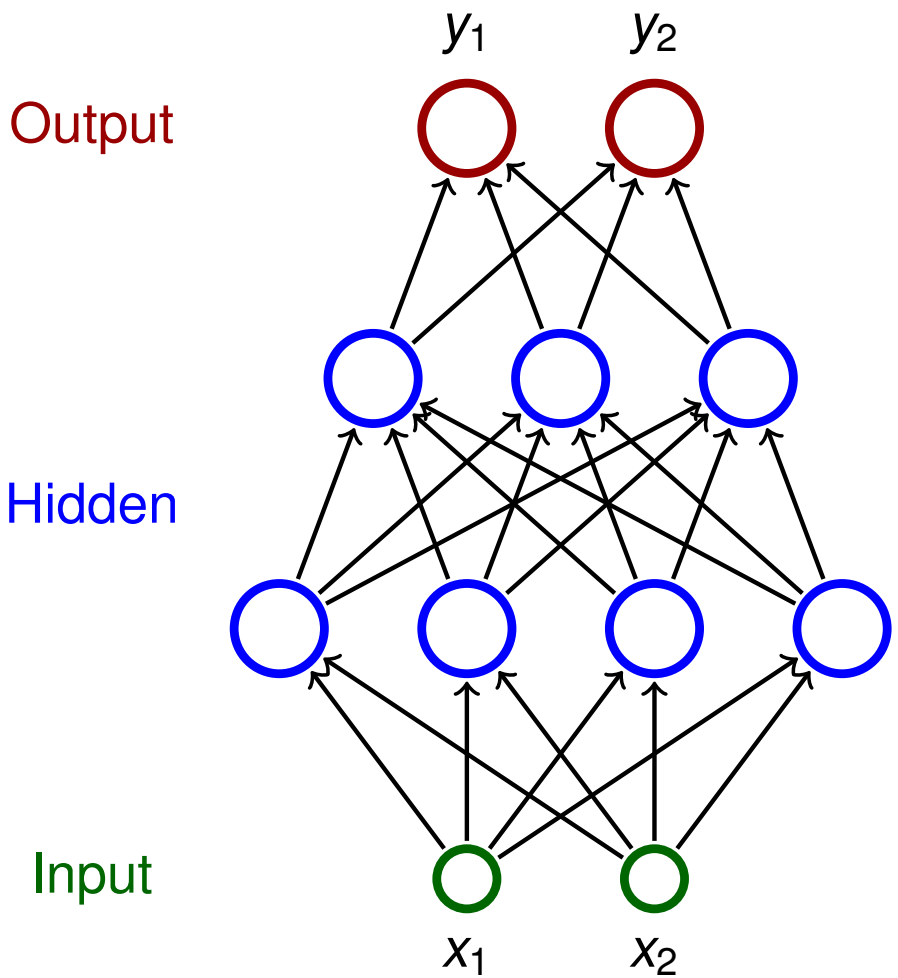


Architecture – Multilayer Perceptron (MLP)



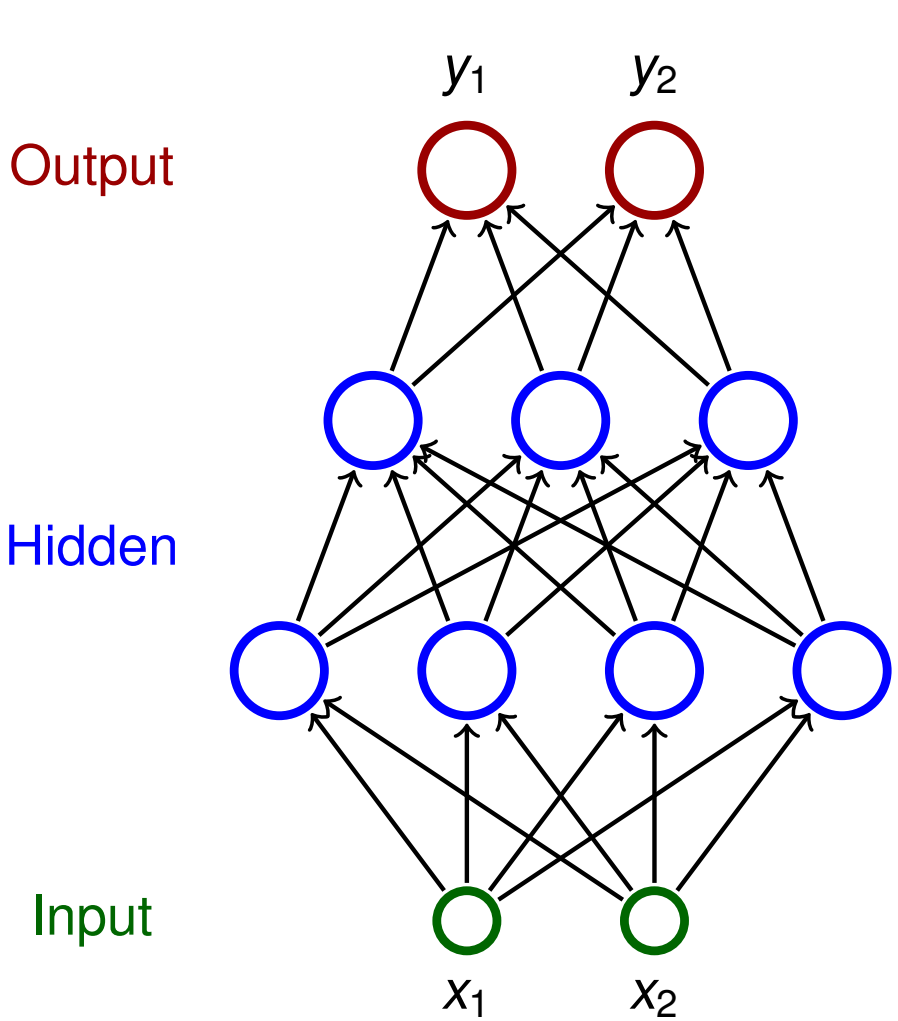
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers

Architecture – Multilayer Perceptron (MLP)



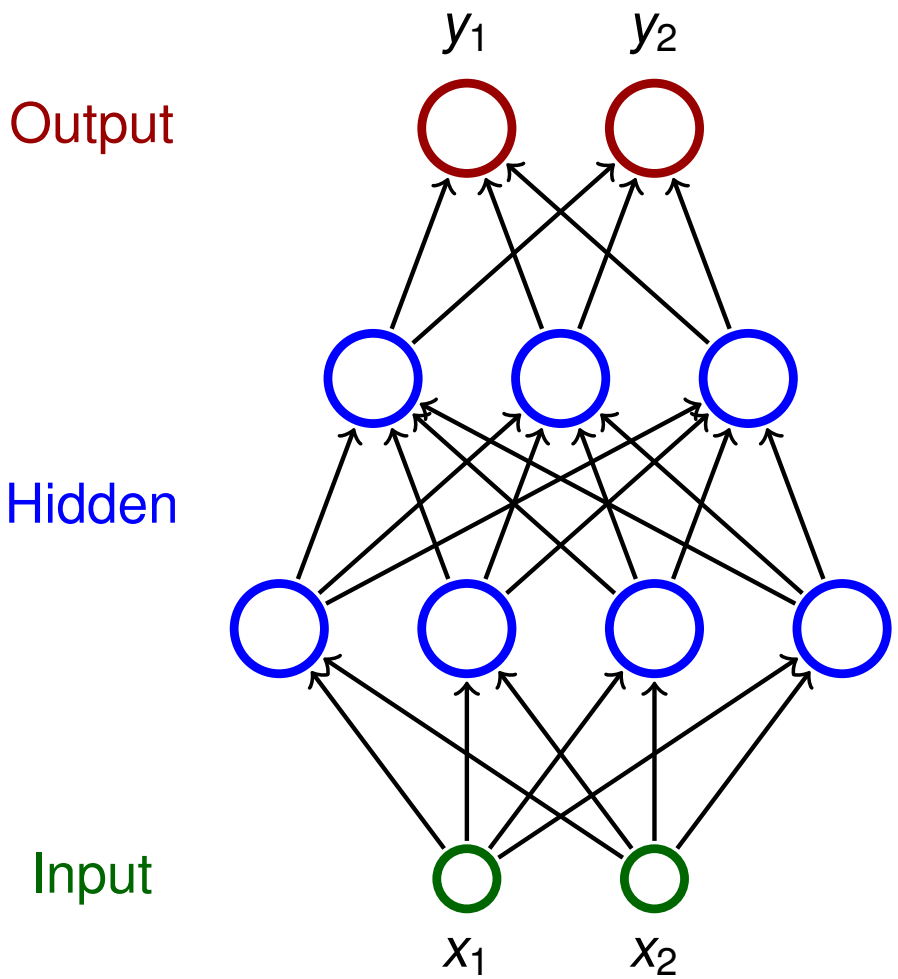
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer

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 – 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.

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.

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 $\mathbb{R}^k$)

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.

Activity – computation of a network

- **Computation** (typically) proceeds in discrete steps.
  In every step the following happens:

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.)

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** $\vec{x}$.

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!

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.*

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.*

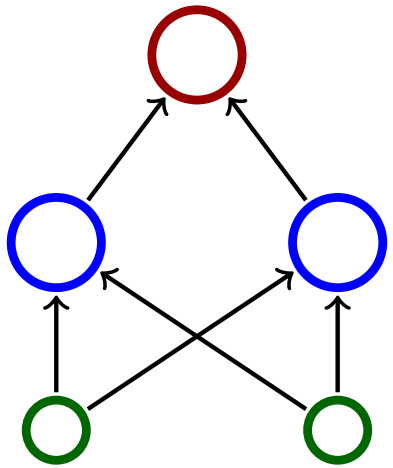
Activity – semantics of a network

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.

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.

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

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

Activity – inner potential and activation functions

There are many activation functions, typical examples:

- Unit step function

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

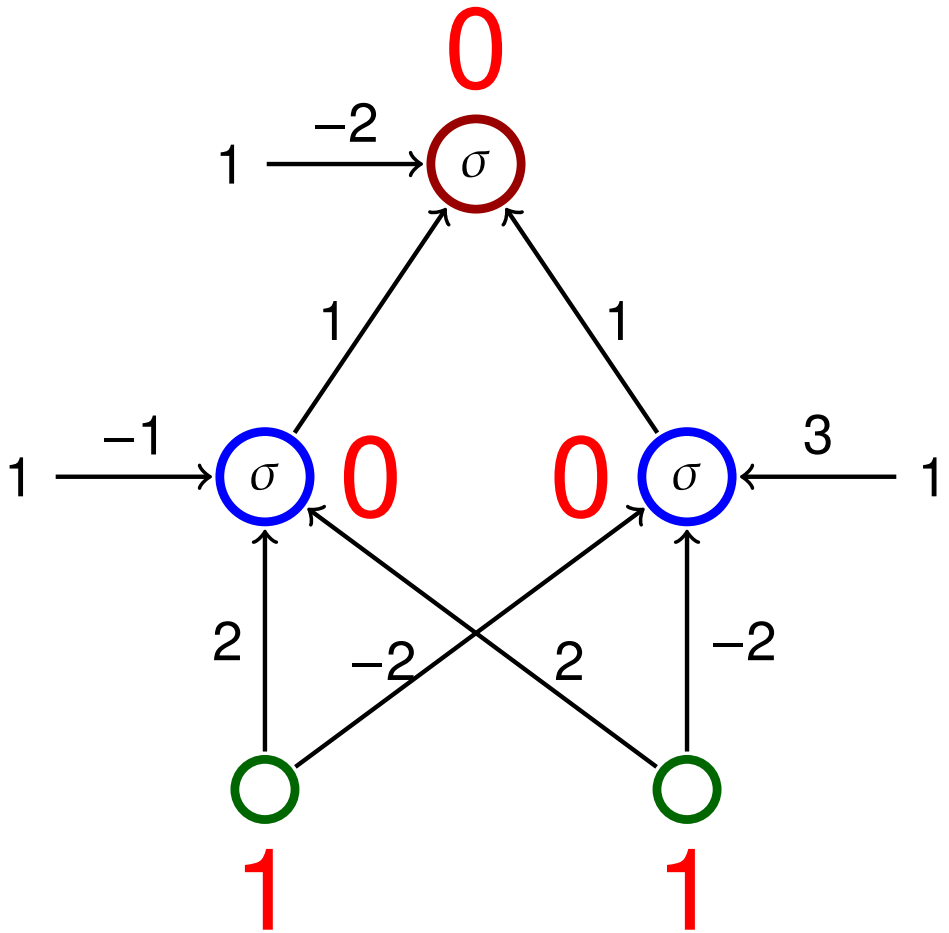
- (Logistic) sigmoid

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

- Hyperbolic tangens

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

Activity – XOR



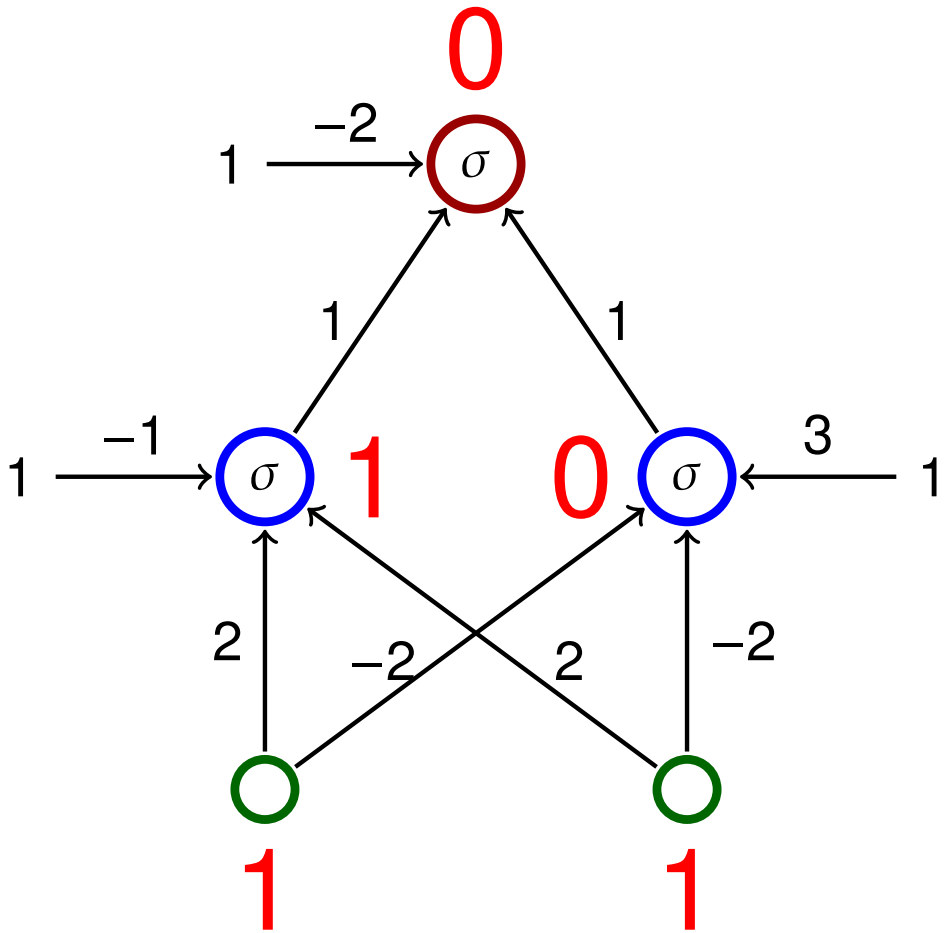
- Activation function is a unit step function

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

- 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 – XOR



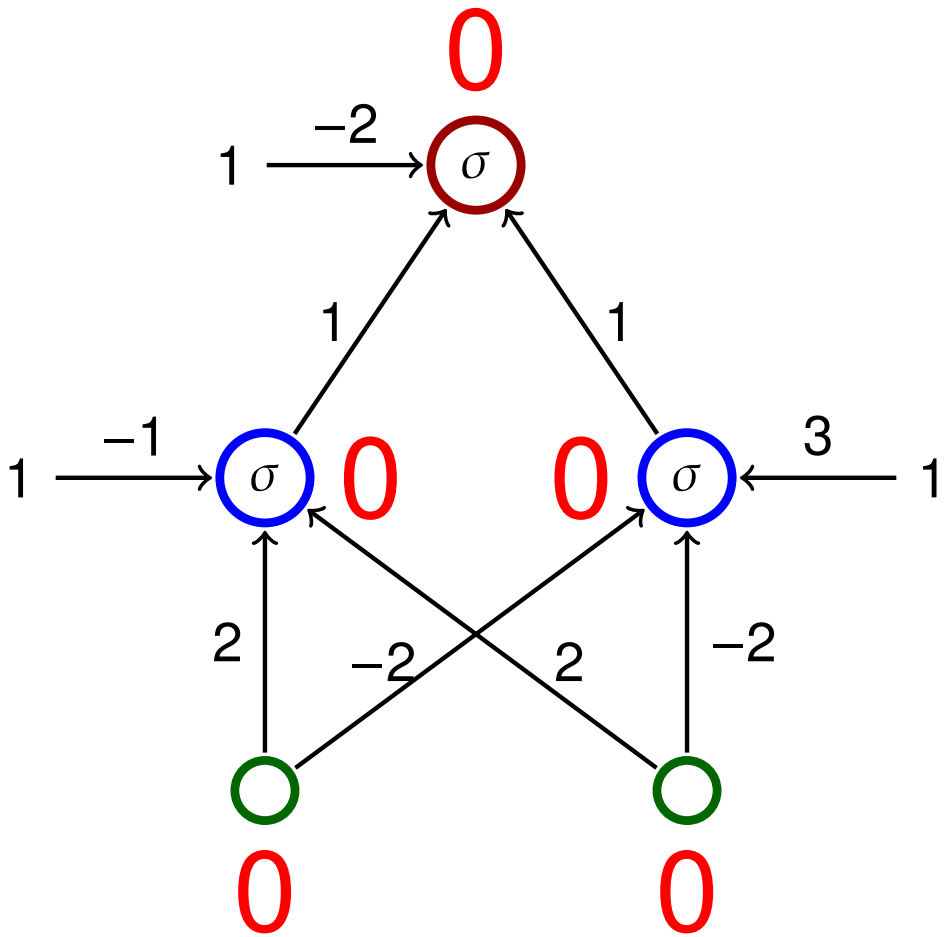
- Activation function is a unit step function

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

- 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 – XOR



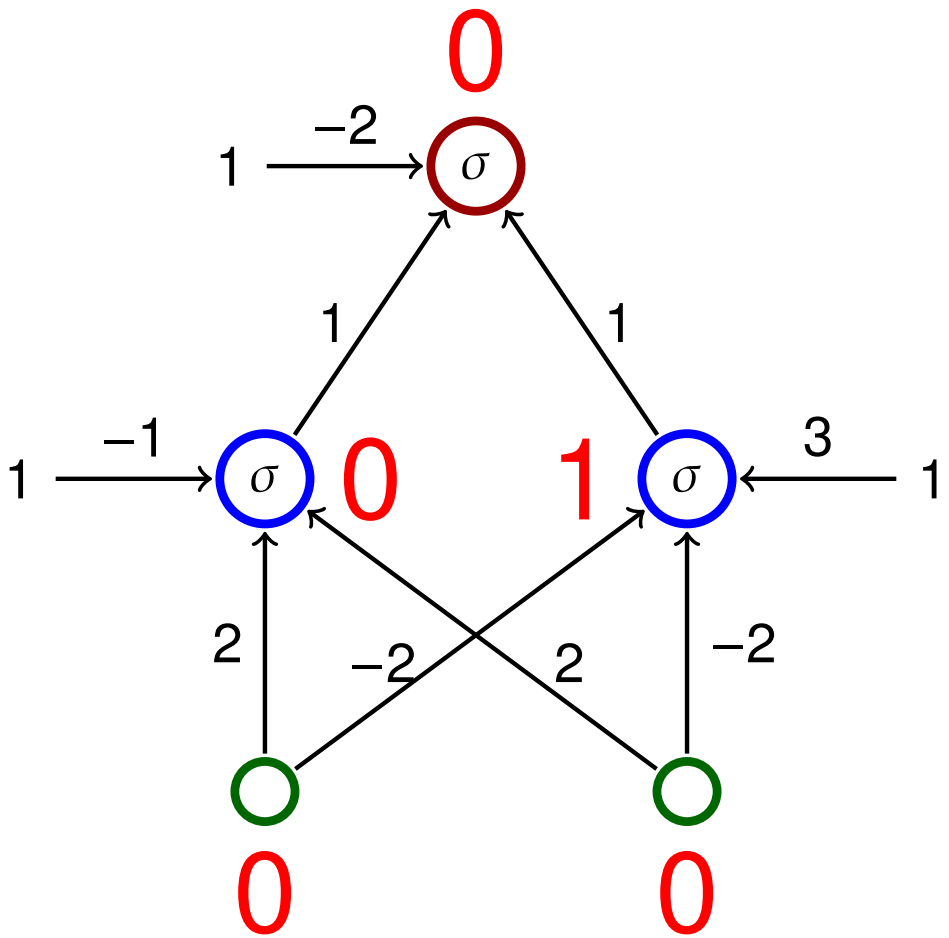
- Activation function is a unit step function

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

- 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 – XOR



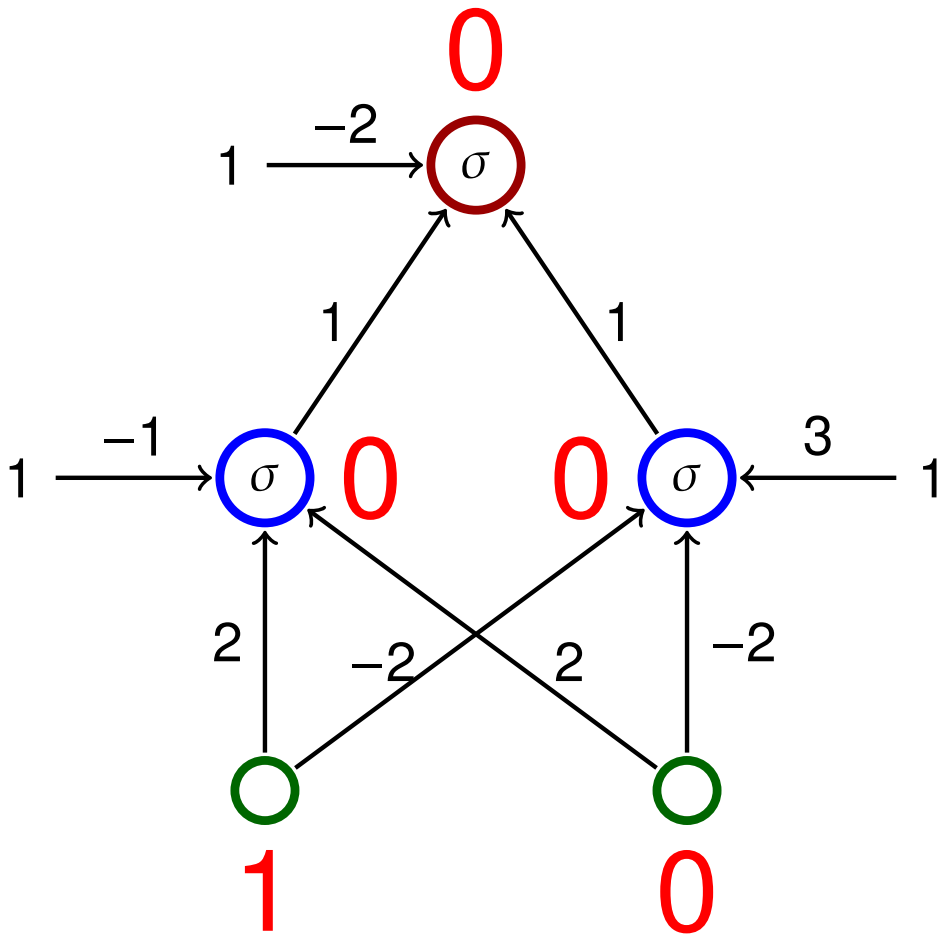
- Activation function is a unit step function

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

- 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 – XOR



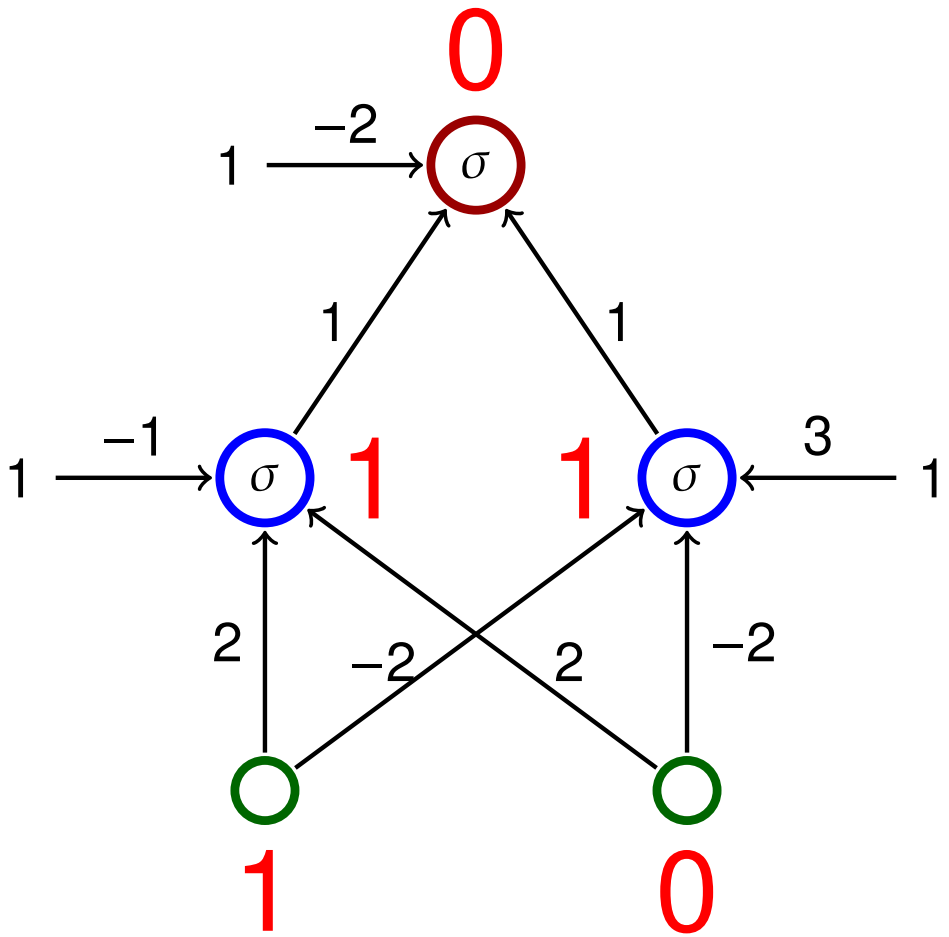
- Activation function is a unit step function

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

- 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 – XOR



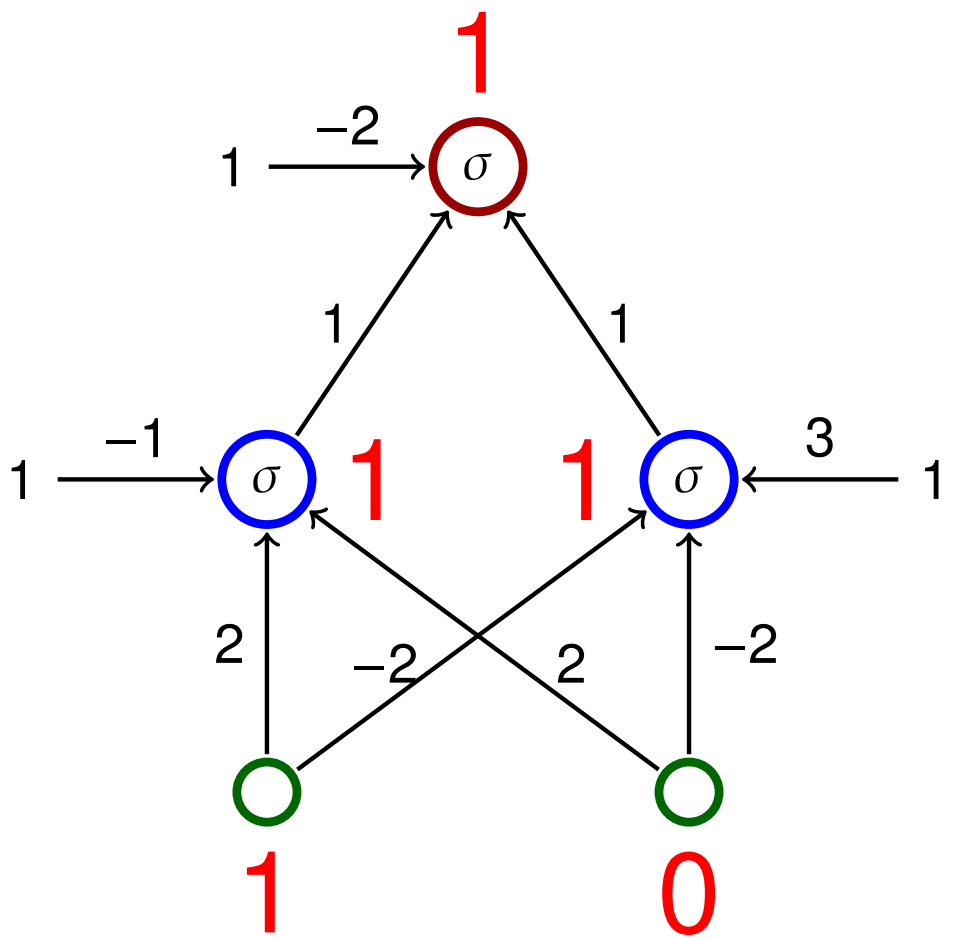
- Activation function is a unit step function

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

- 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 – XOR



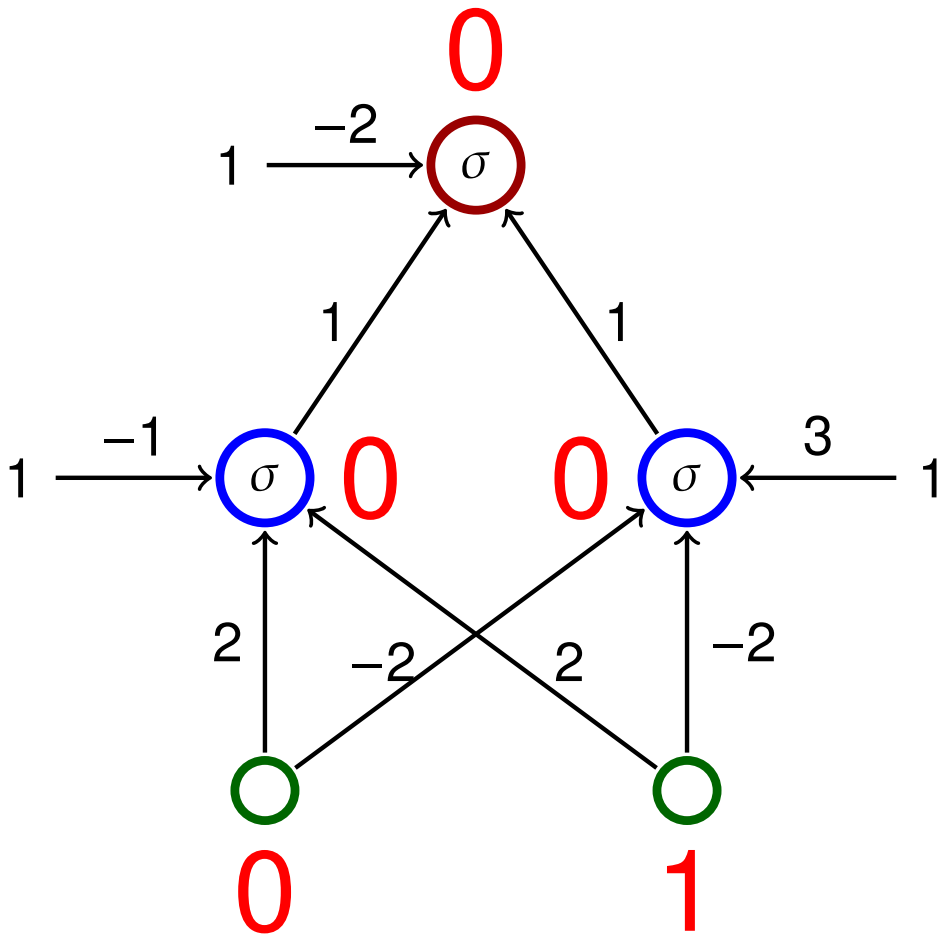
- Activation function is a unit step function

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

- 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 – XOR



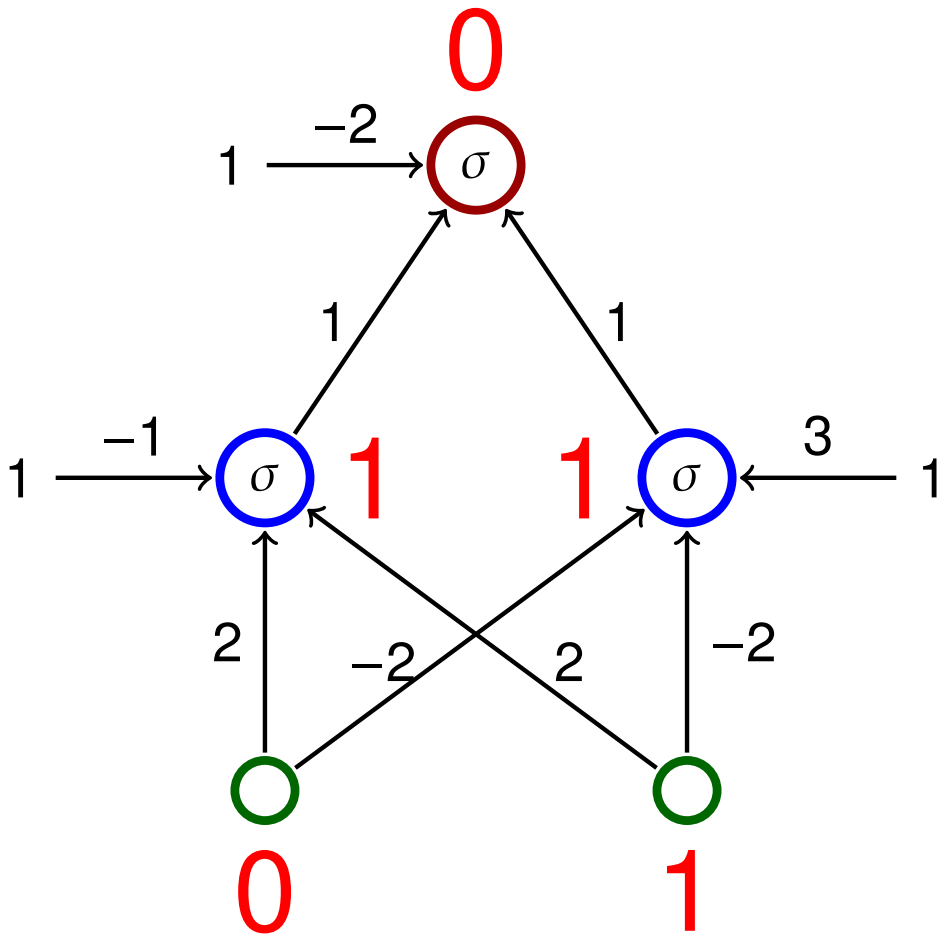
- Activation function is a unit step function

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

- 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 – XOR



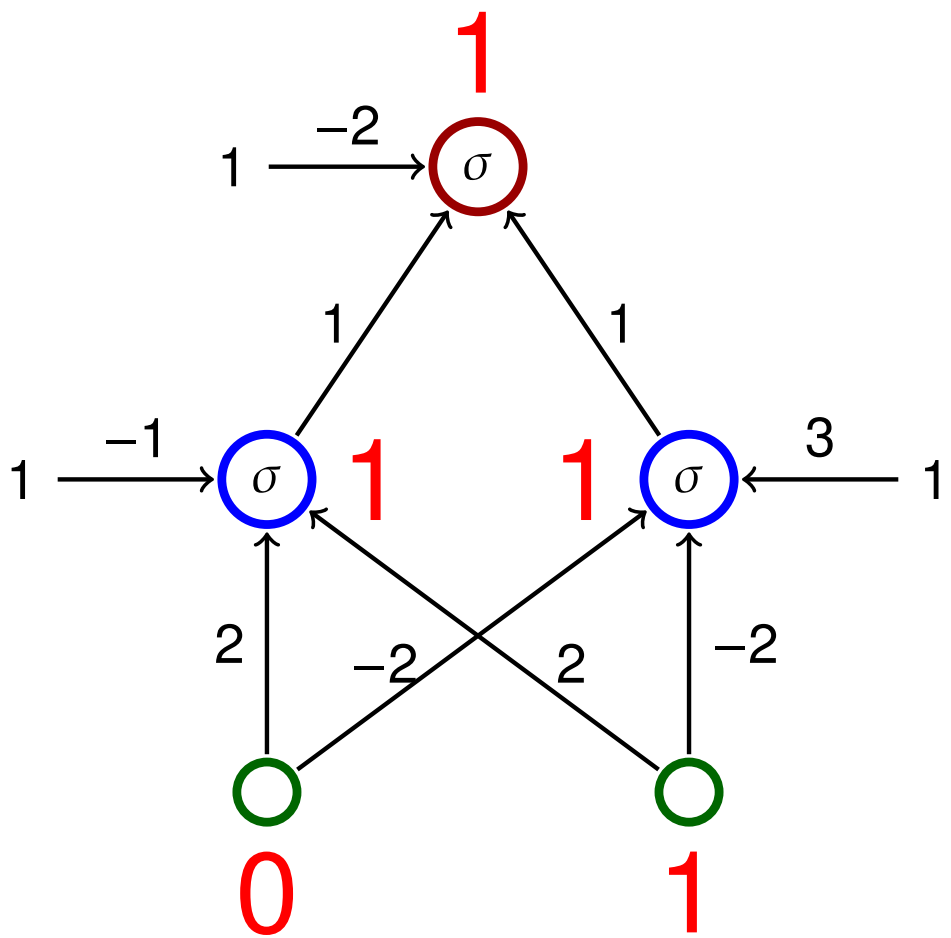
- Activation function is a unit step function

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

- 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 – XOR



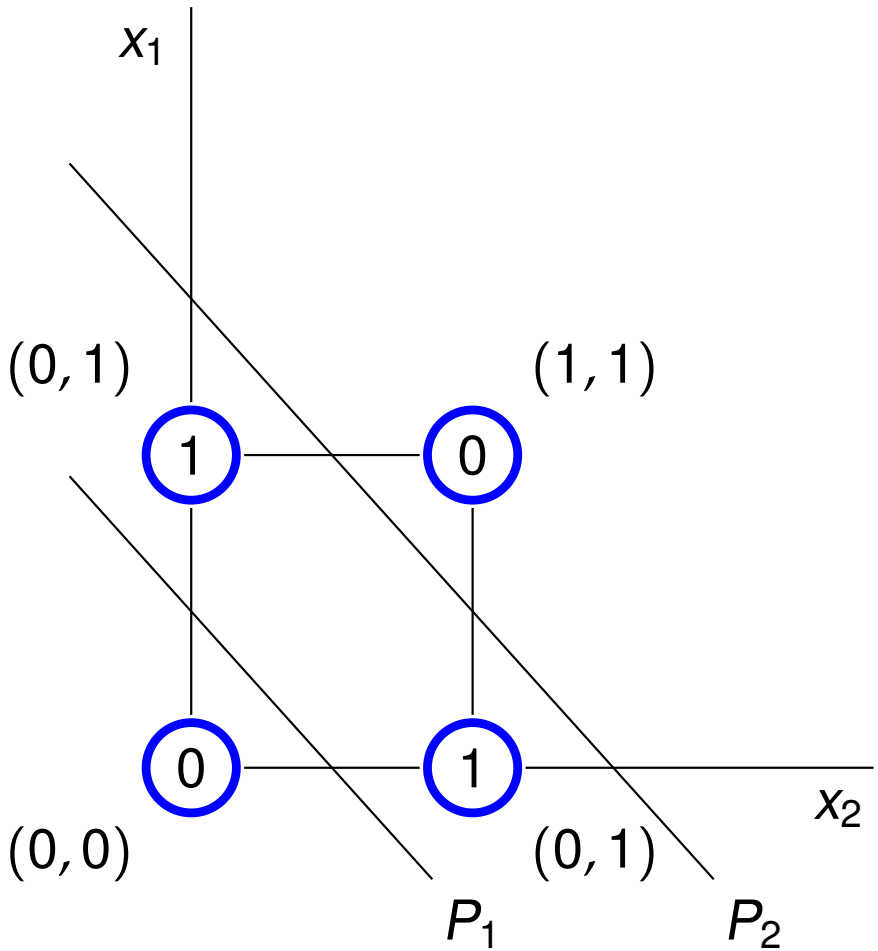
- Activation function is a unit step function

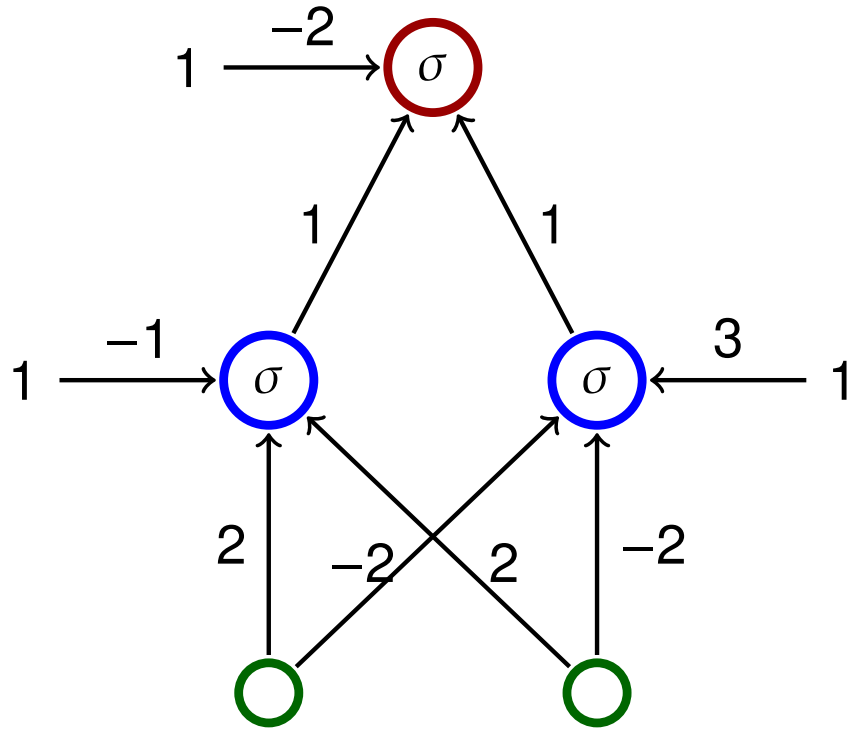
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

- 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

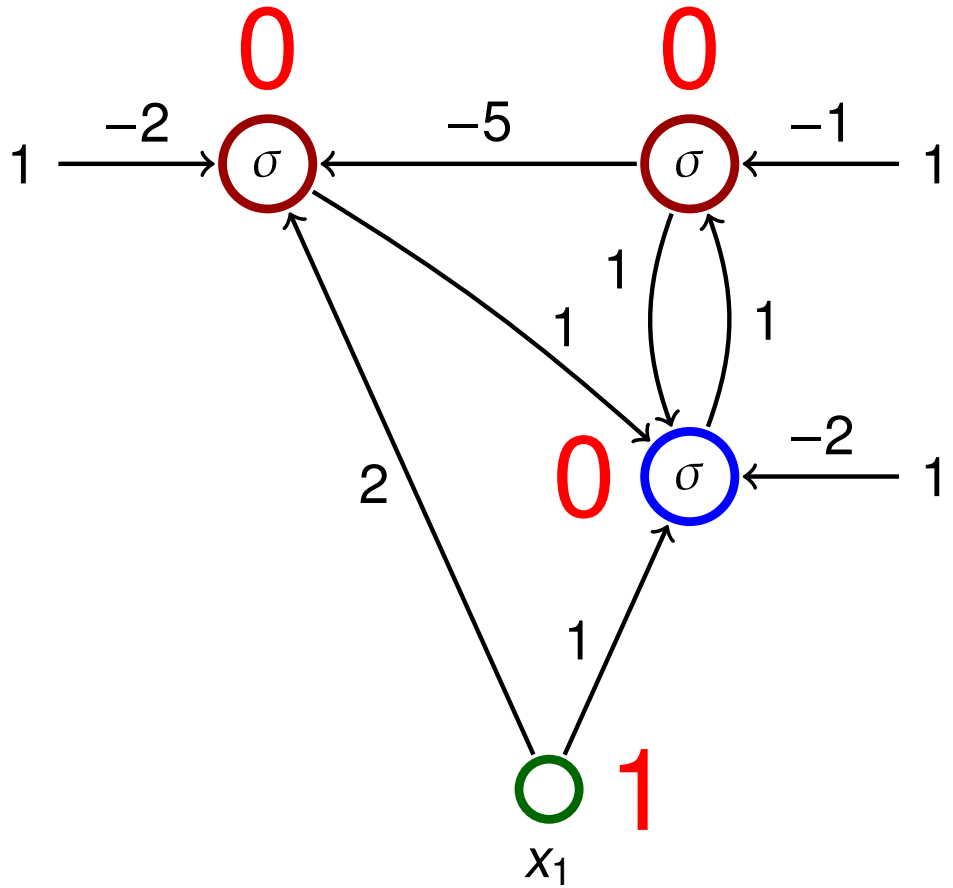
Activity – MLP and linear separation





- 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

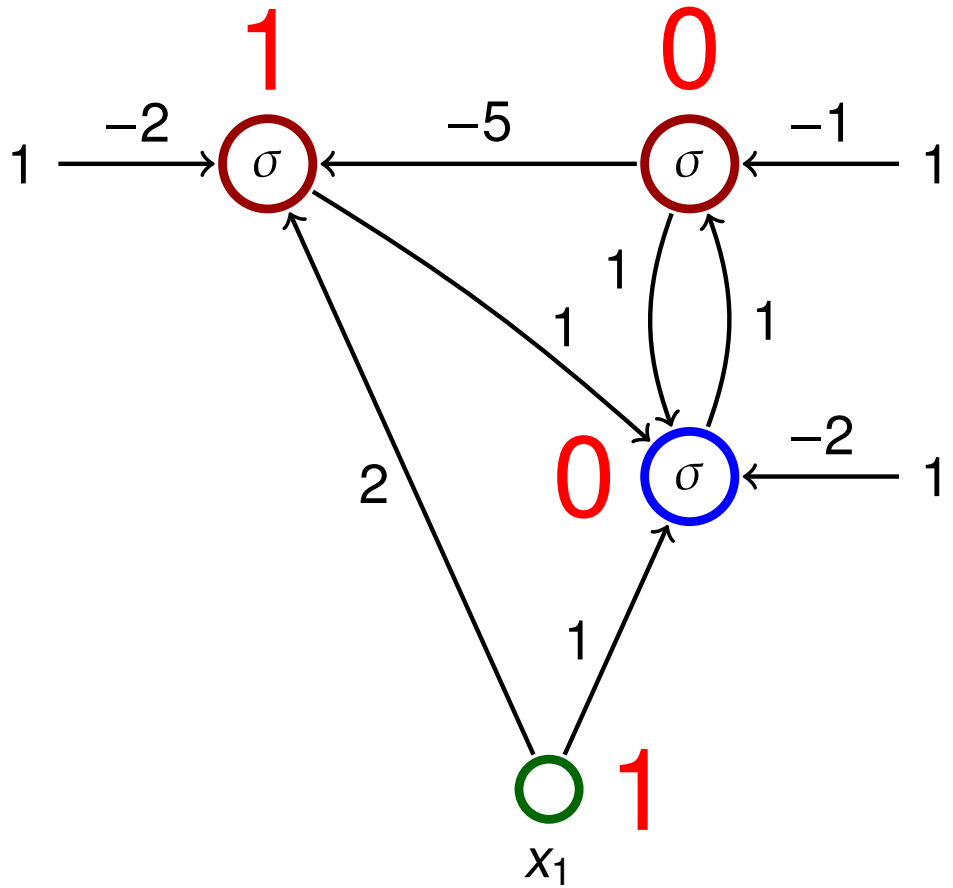


The activation function is the unit step function

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

The input is equal to 1

Activity – example

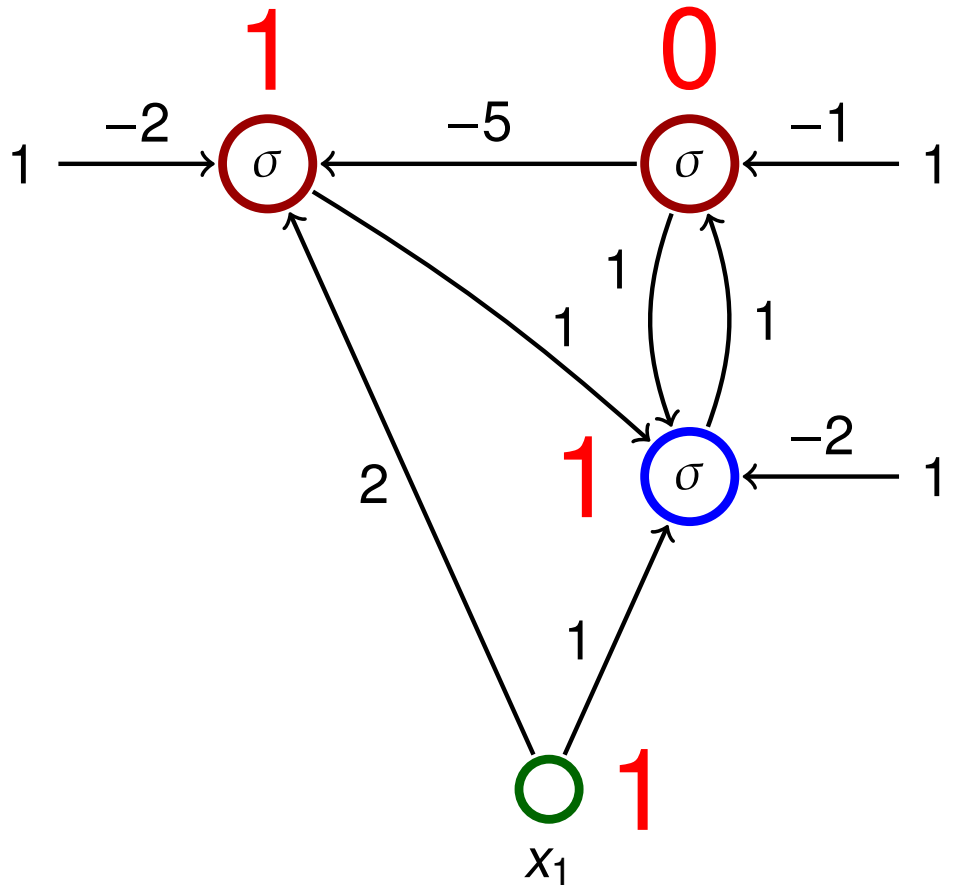


The activation function is the unit step function

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

The input is equal to 1

Activity – example

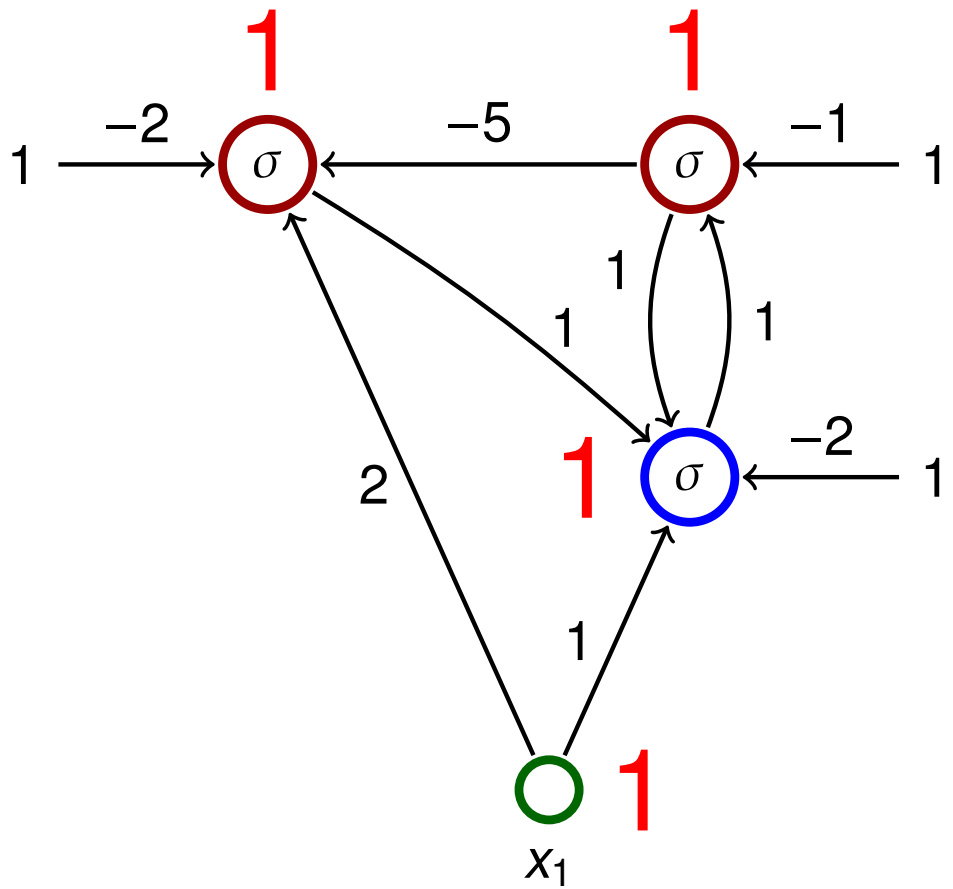


The activation function is the unit step function

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

The input is equal to 1

Activity – example

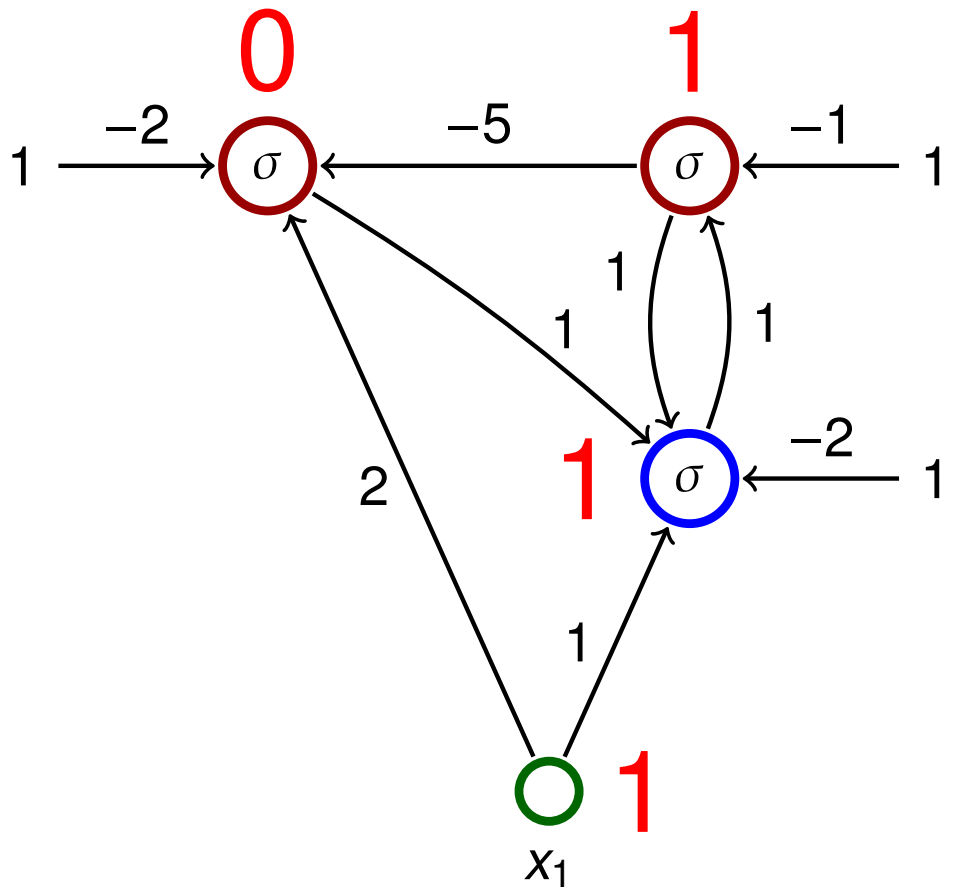


The activation function is the unit step function

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

The input is equal to 1

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.

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.

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)

Learning algorithms

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

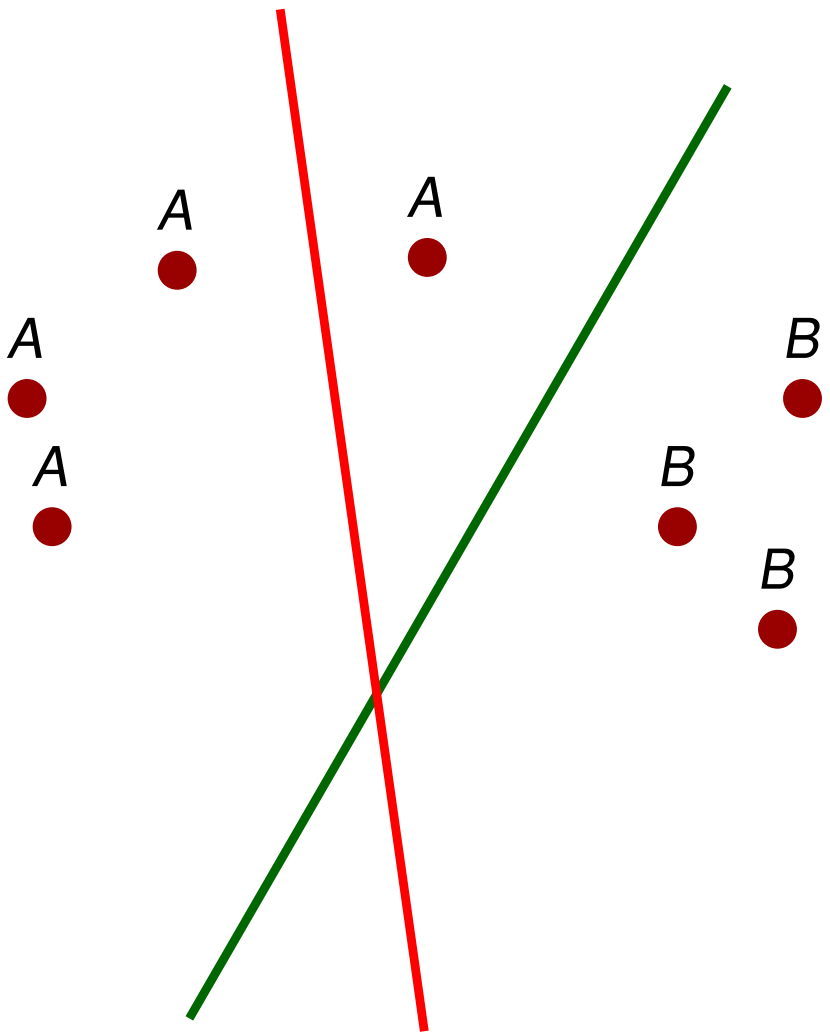
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.

Learning algorithms

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

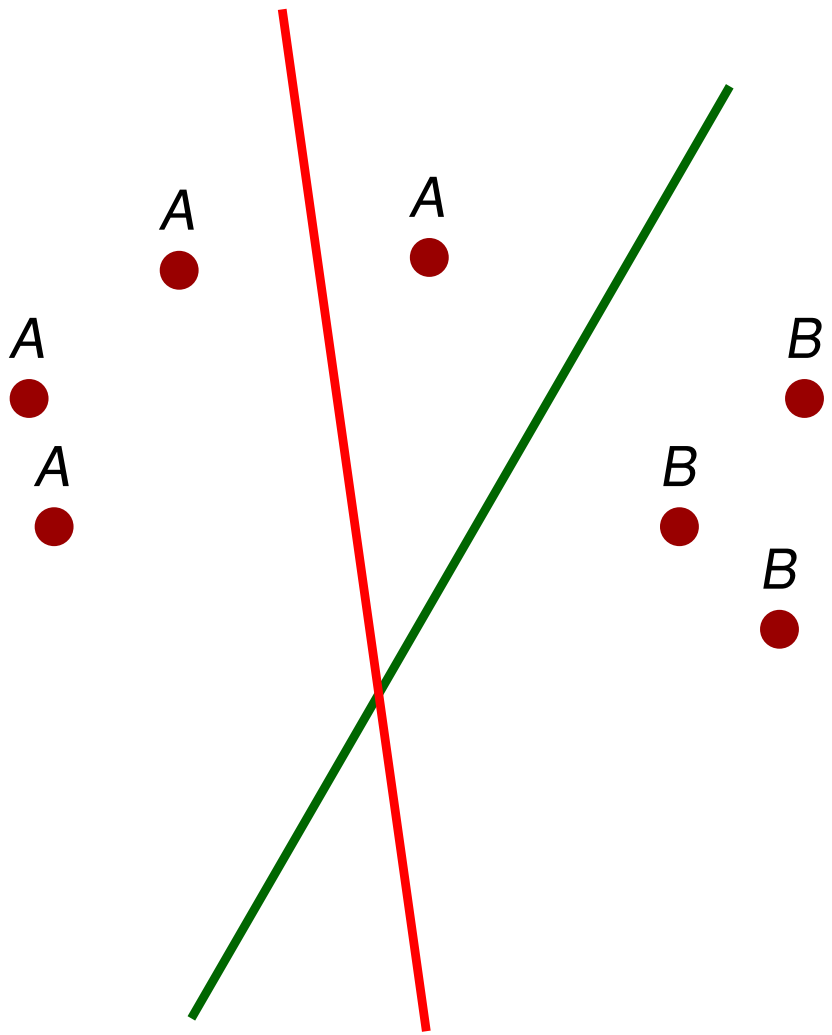
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.
- Unsupervised learning
  - The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

Supervised learning – illustration



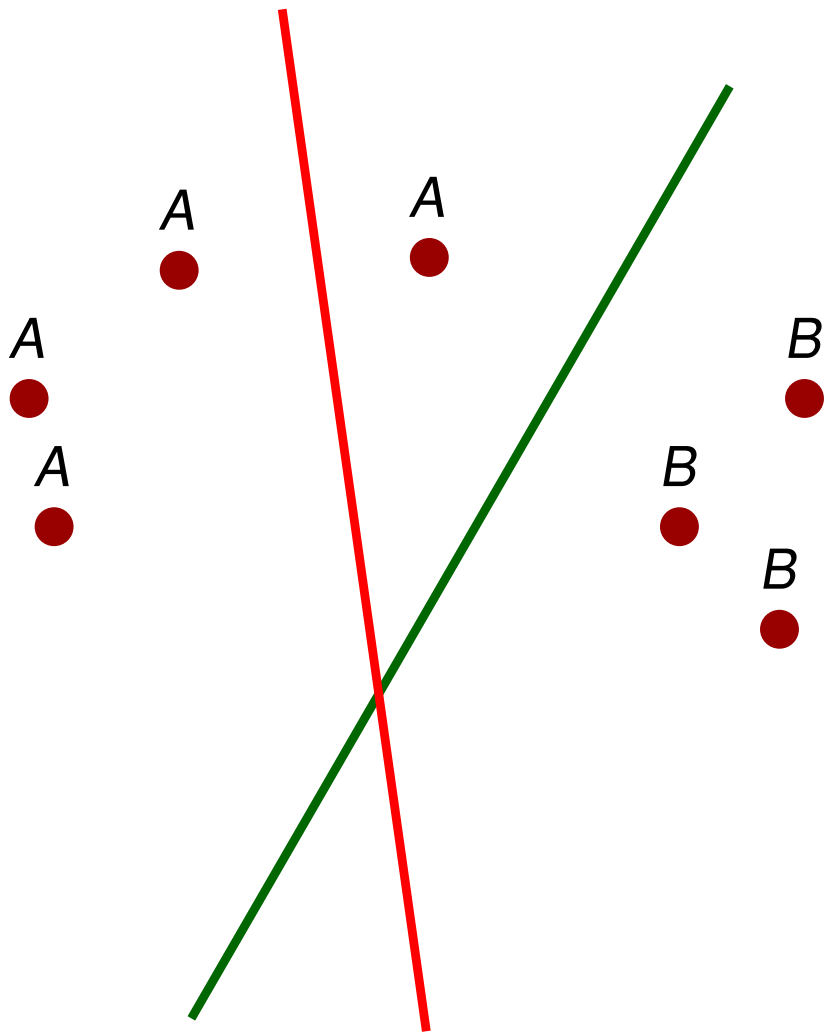
- classification in the plane using a single neuron

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*

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

Summary – Advantages of neural networks

- Massive parallelism
  - neurons can be evaluated in parallel

Summary – Advantages of neural networks

- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks

Summary – Advantages of neural networks

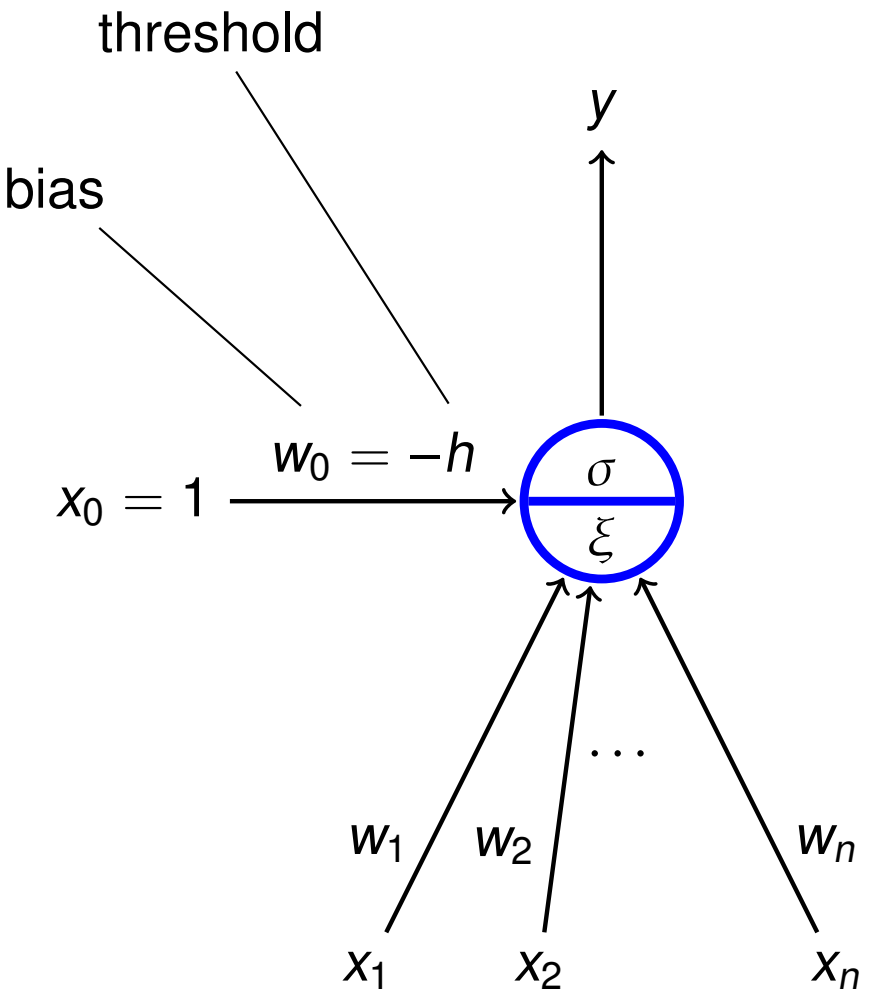
- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks
- generalization and robustness
  - information is encoded in a distributed manned in weights
  - "close" inputs typicaly get similar values

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

Expressive power of neural networks

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*

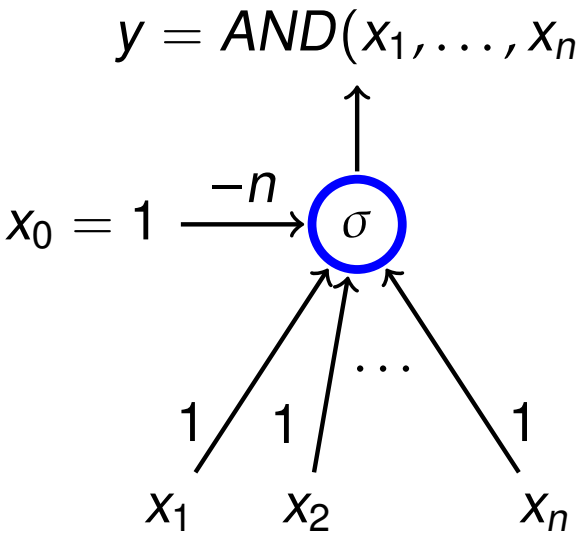
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

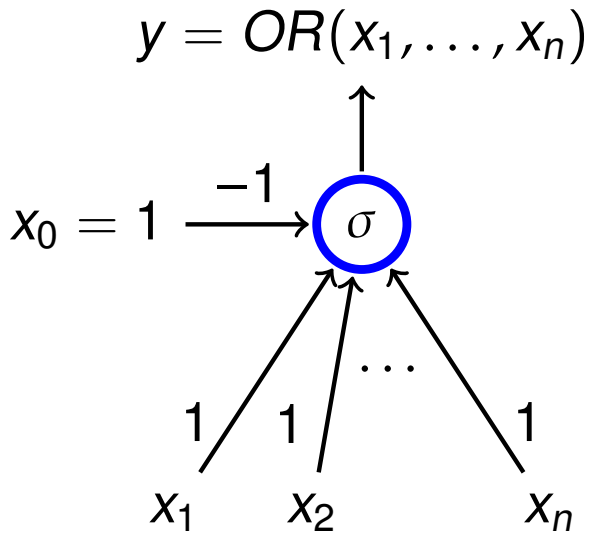
Boolean functions

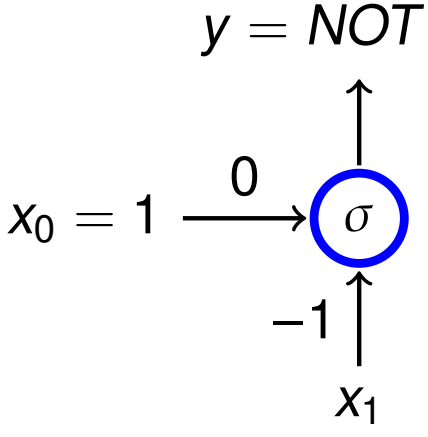
Activation function: *unit step function* $\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$

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 $F : \{0,1\}^n \to \{0,1\}$.

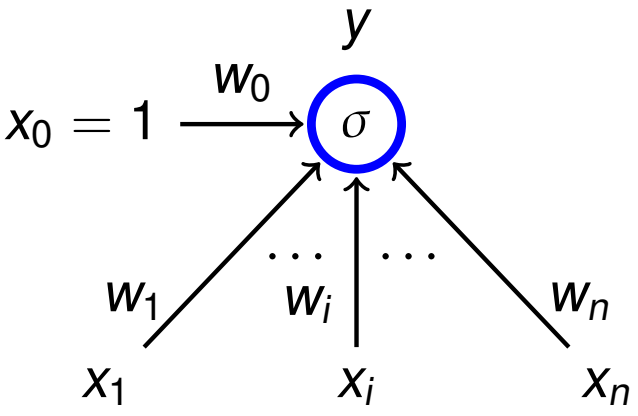
Boolean functions

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 $F : \{0,1\}^n \to \{0,1\}$.

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 $\vec{v}$:

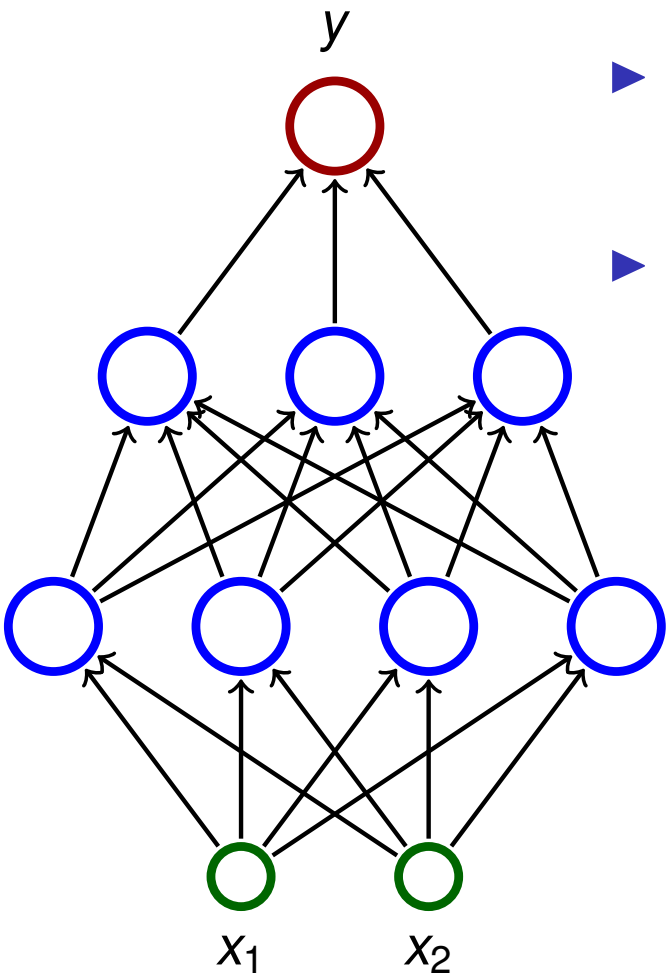


$$w_0 = -\sum_{i=1}^{n} v_i$$

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

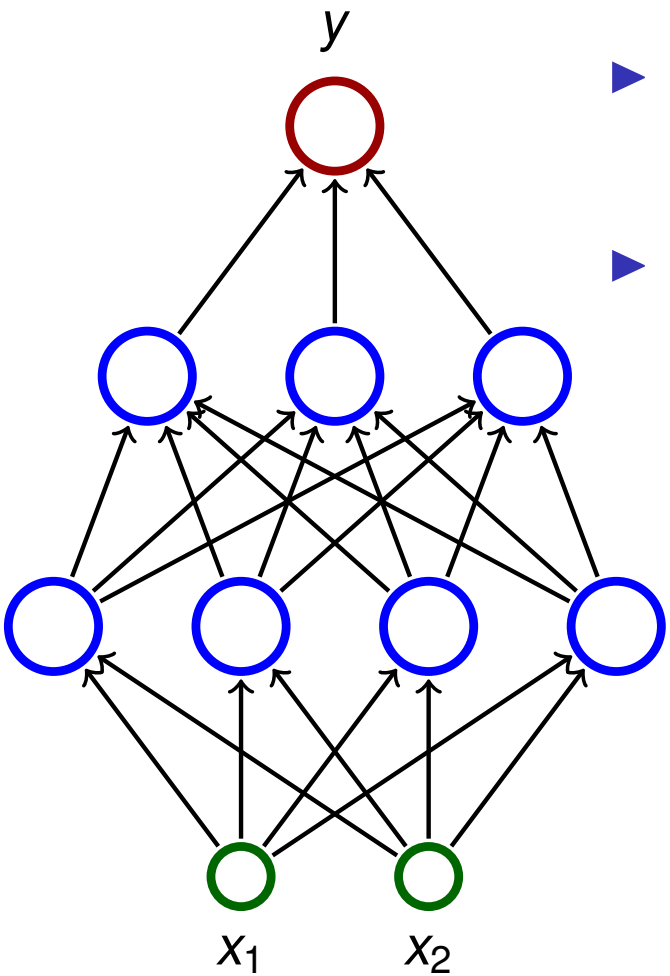
- Now let us connect all outputs of all neurons $N_{\vec{v}}$ satisfying $F(\vec{v}) = 1$ using a neuron implementing *OR*. □

Non-linear separation



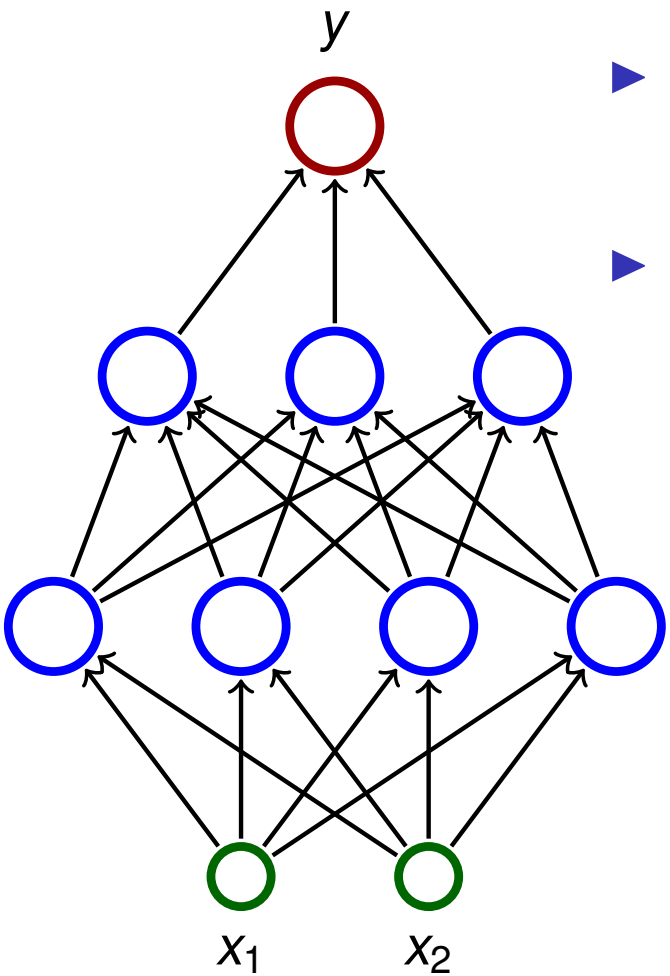
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).

Non-linear separation



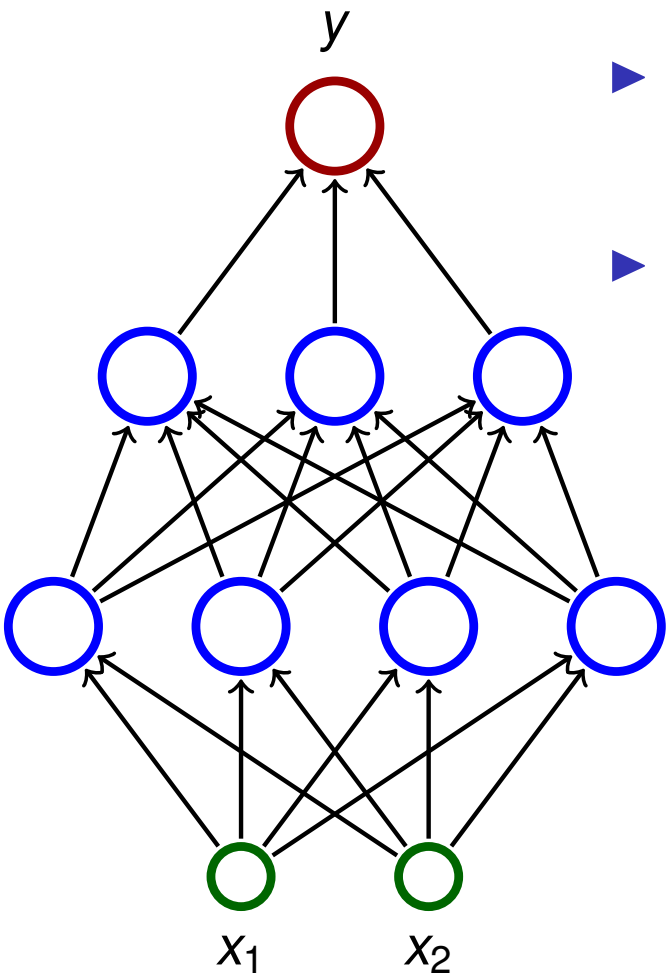
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.

Non-linear separation



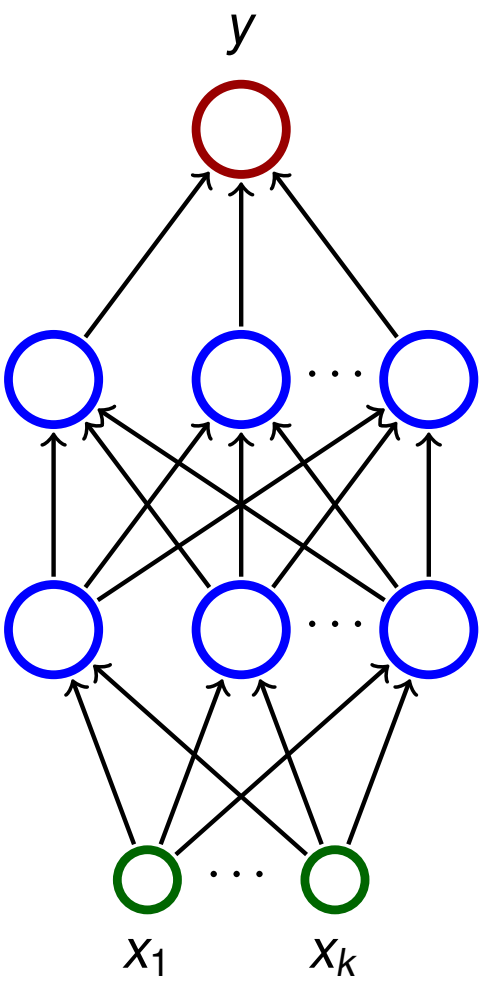
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation



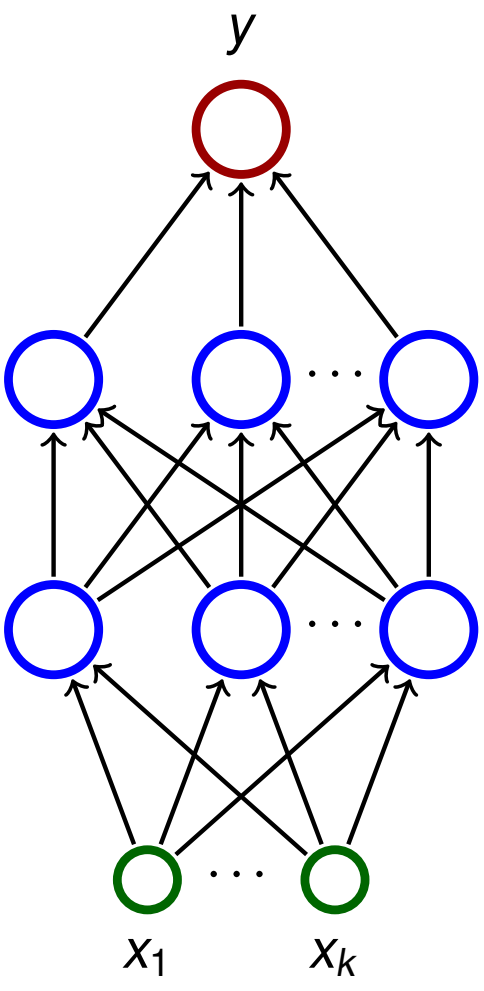
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation – illustration



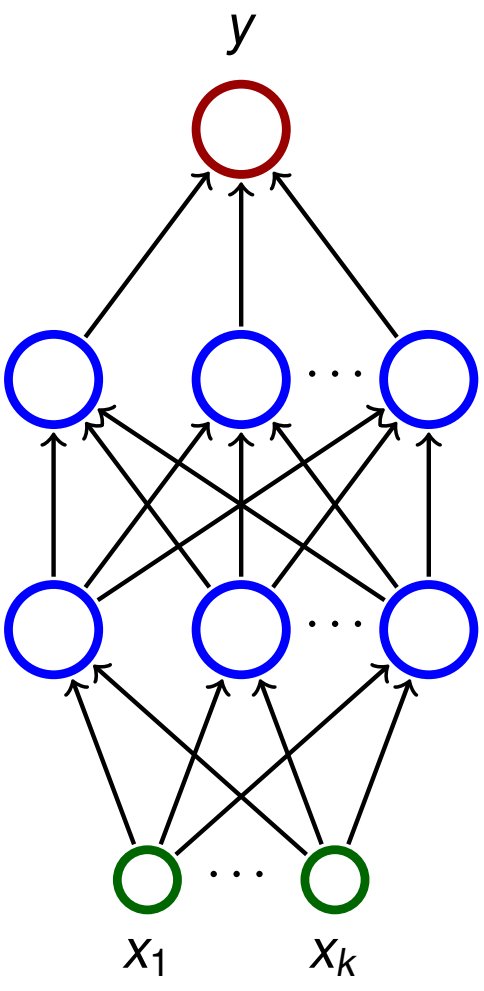
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.

Non-linear separation – illustration



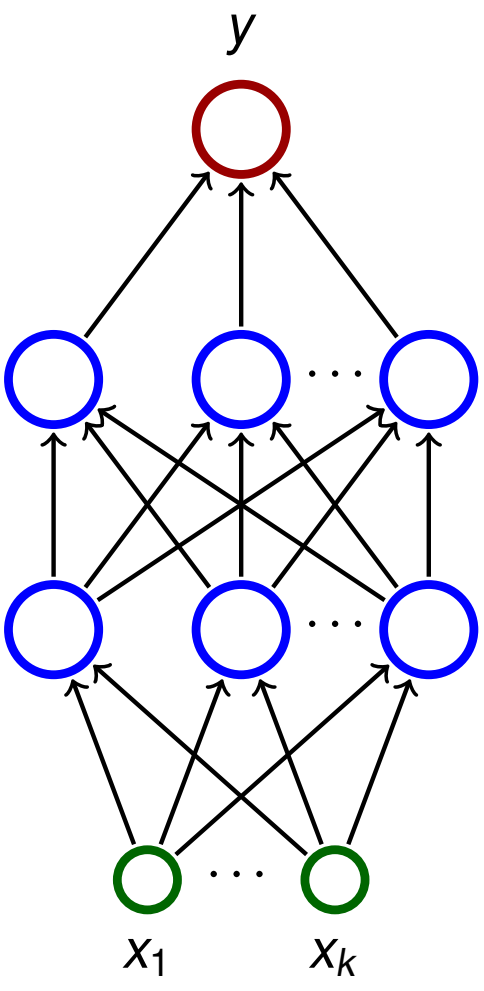
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)

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).

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

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

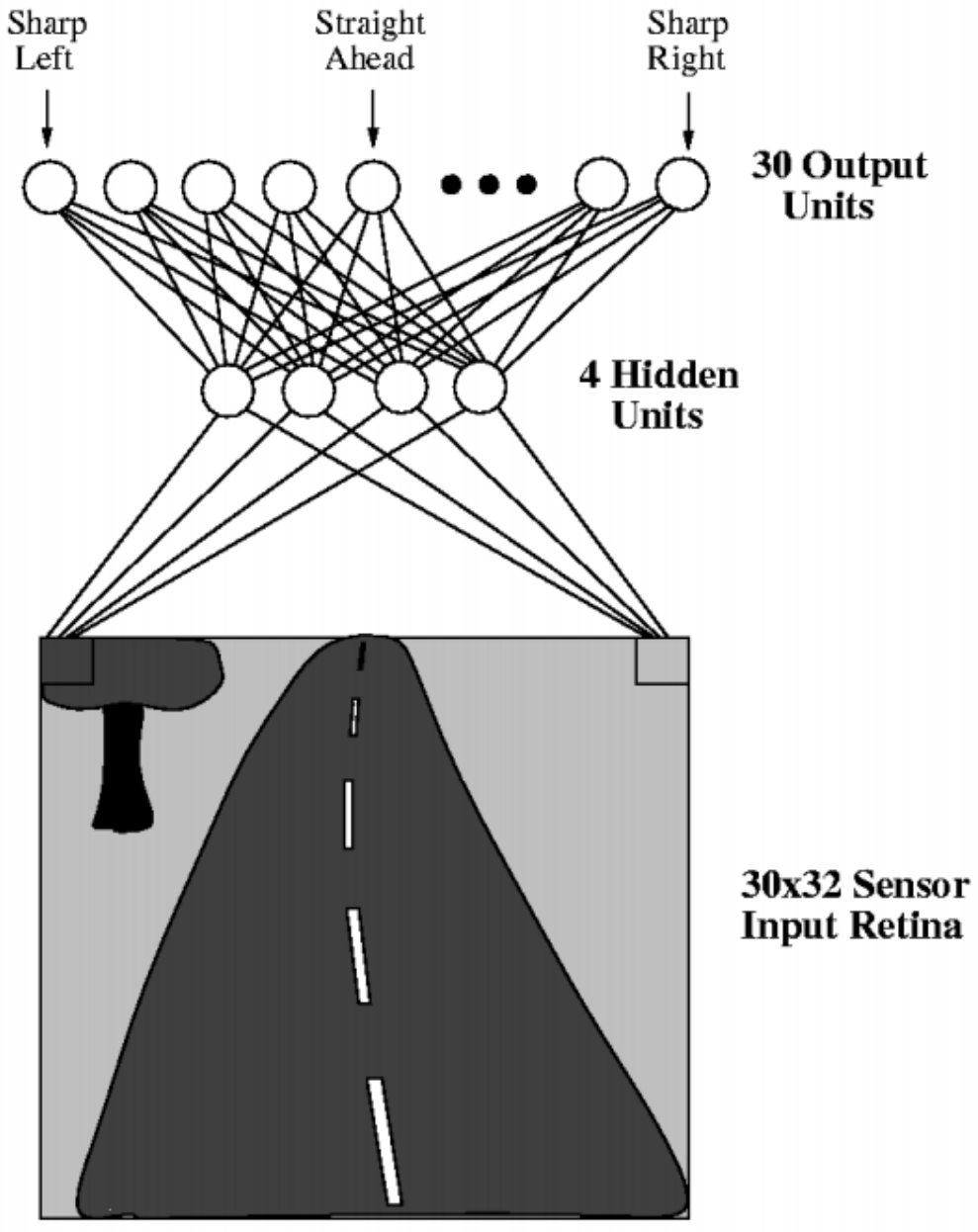
For every "reasonable" set $A \subseteq [0,1]^n$, there is a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following:*

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:

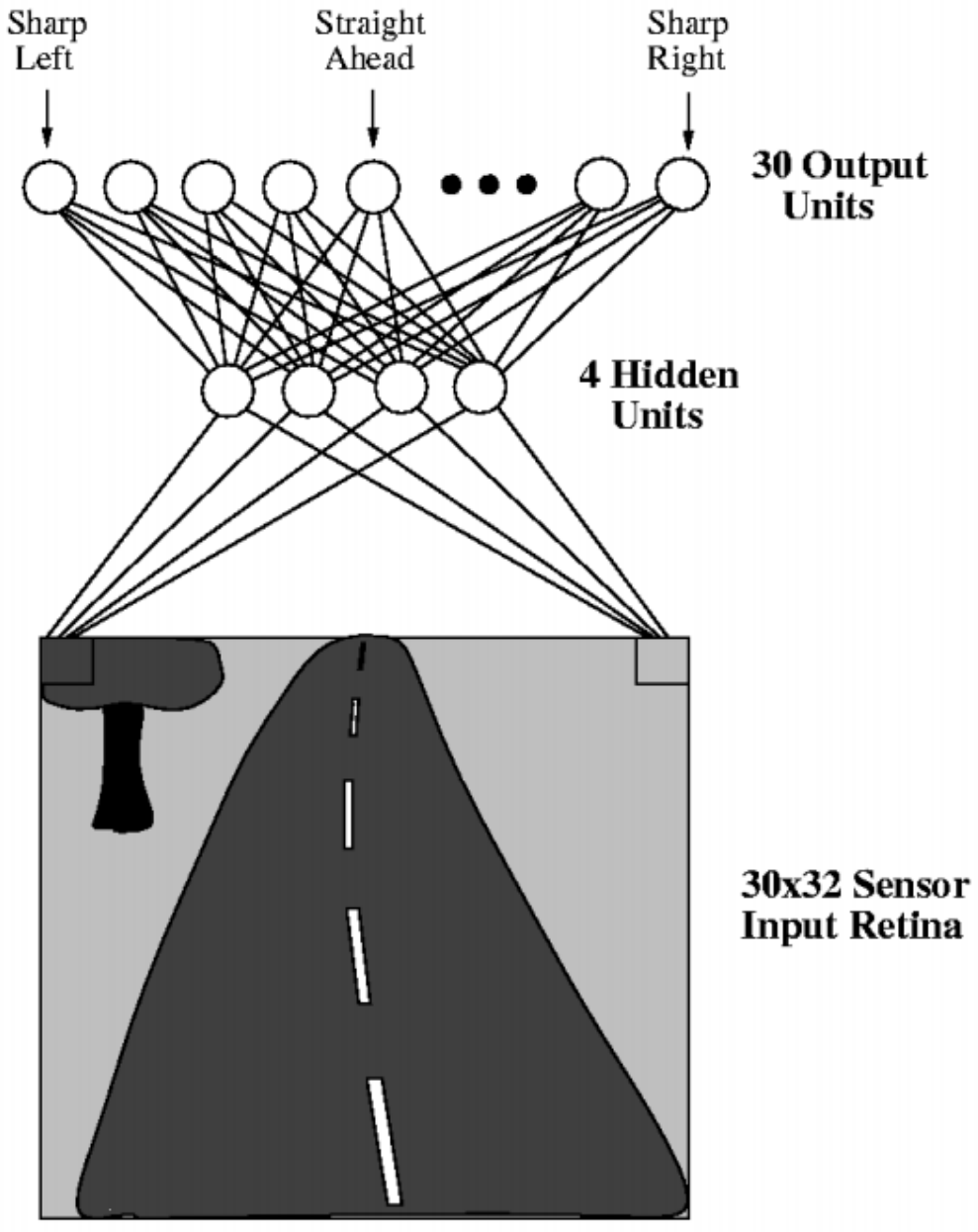
- "reasonable" means Lebesgue measurable
- "most" means that the set of incorrectly classified vectors has the Lebesgue measure smaller than a given $\varepsilon > 0$

Non-linear separation - practical illustration



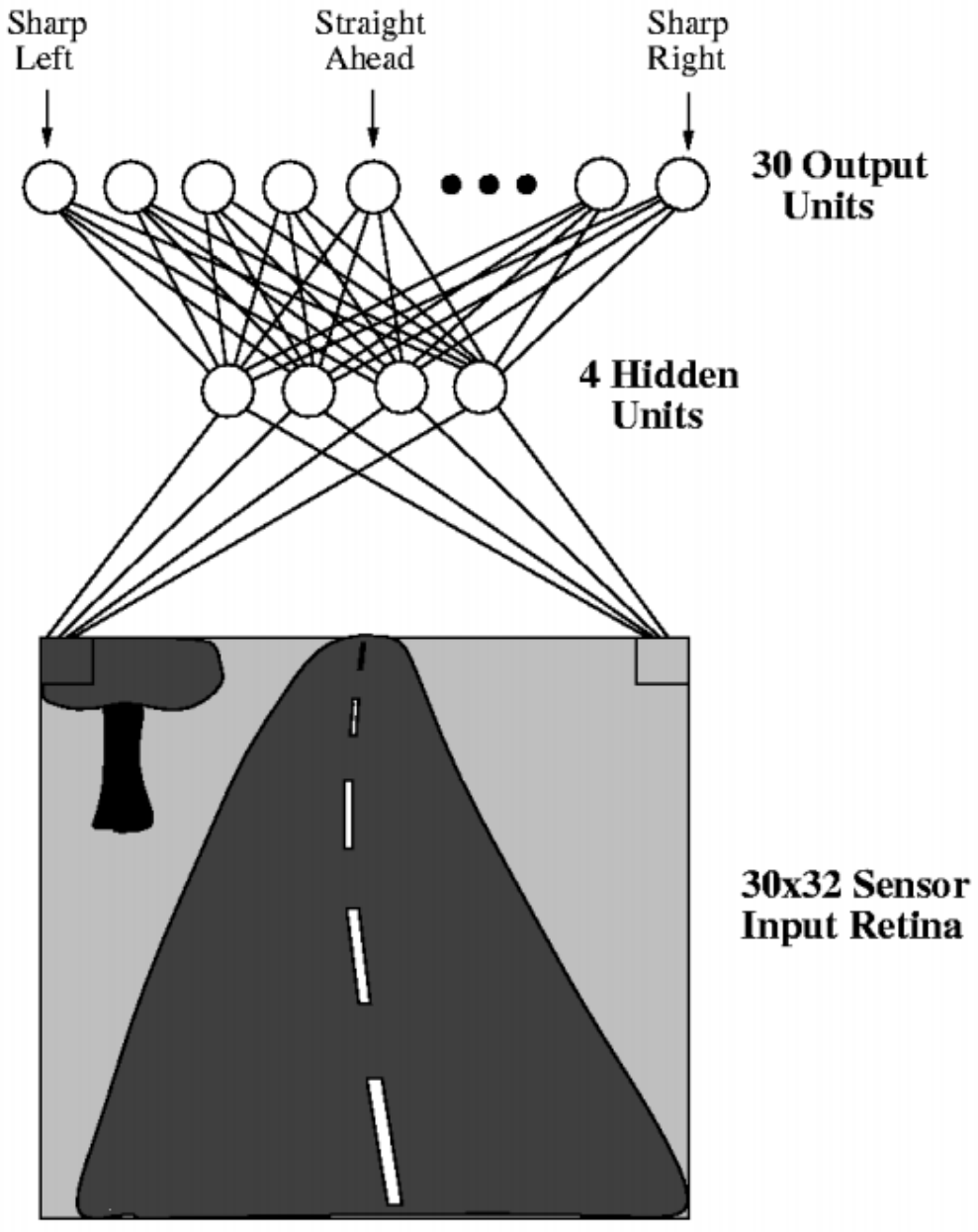
- ALVINN drives a car

Non-linear separation - practical illustration



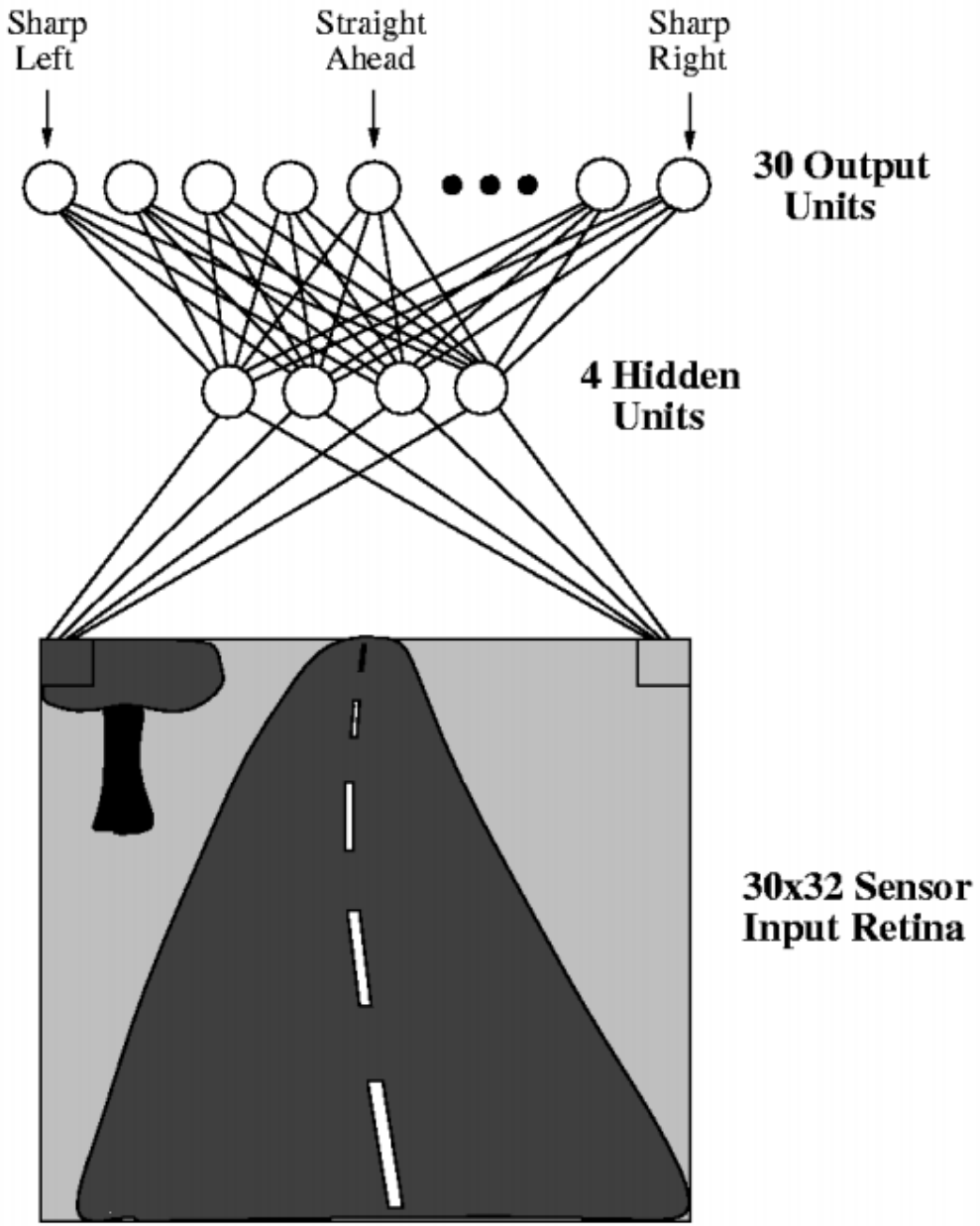
- ALVINN drives a car
- The net has $30 \times 32 = 960$ inputs (the input space is thus $\mathbb{R}^{960}$)

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.

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

- there is a linear activation in the output layer, i.e. the value of the output neuron is its inner potential ξ,

Function approximation - three layers

Let σ be a logistic sigmoid, i.e.

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

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

Function approximation - three layers

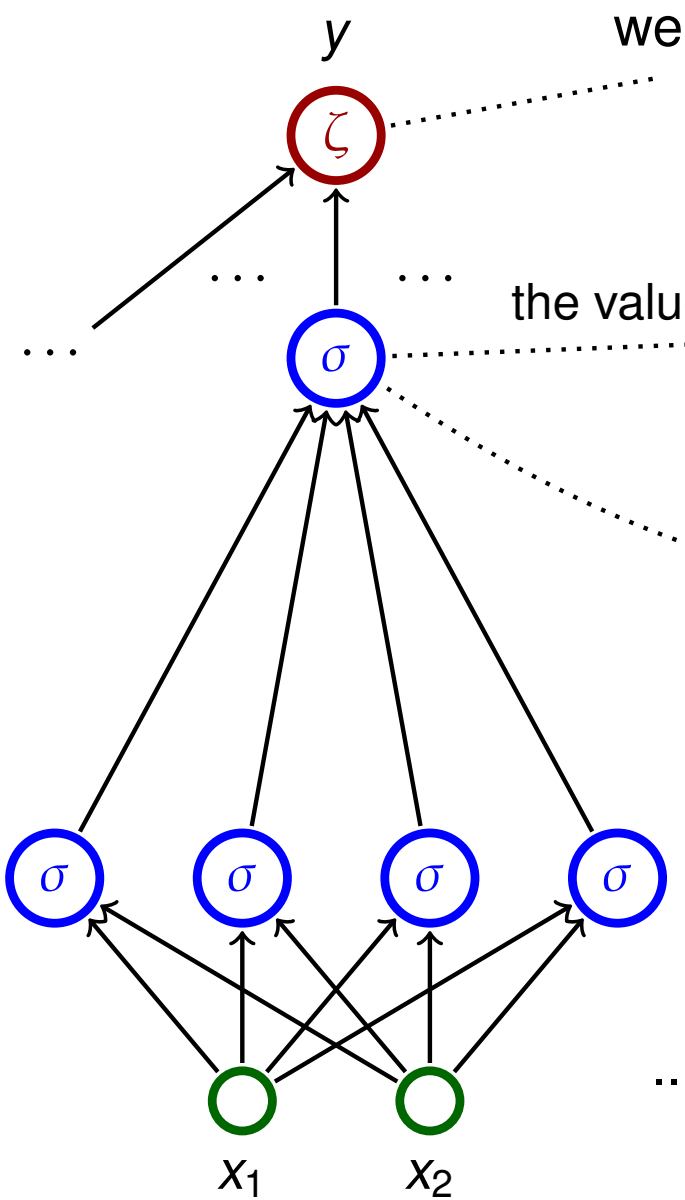
Let σ be a logistic sigmoid, i.e.

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

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

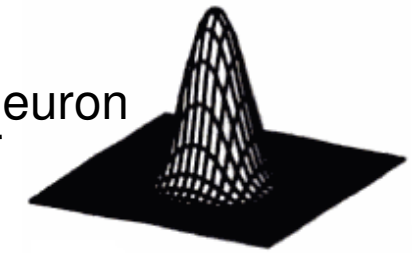
- there is a linear activation in the output layer, i.e. the value of the output neuron is its inner potential ξ,
- the remaining neurons have the logistic sigmoid σ as their activation,
- for every $\vec{v} \in [0,1]^n$ we have that $|F(\vec{v}) - f(\vec{v})| < \varepsilon$.

Function approximation – three layer networks



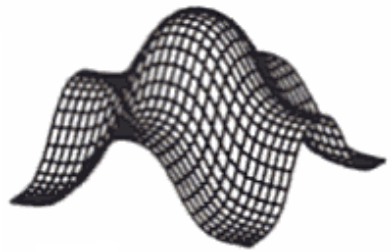
weighted sum of "spikes"

the value of the neuron



a "spike"

inner potential



... + the other two 90 degree rotations

Function approximation - two-layer networks

Theorem (Cybenko 1989)

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

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

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*

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

Neural networks and computability

- Consider recurrent networks (i.e. containing cycles)

Neural networks and computability

- Consider recurrent networks (i.e. containing cycles)
  - with real weights (in general);

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);

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);

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

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

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

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

- 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

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

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)

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

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.

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.

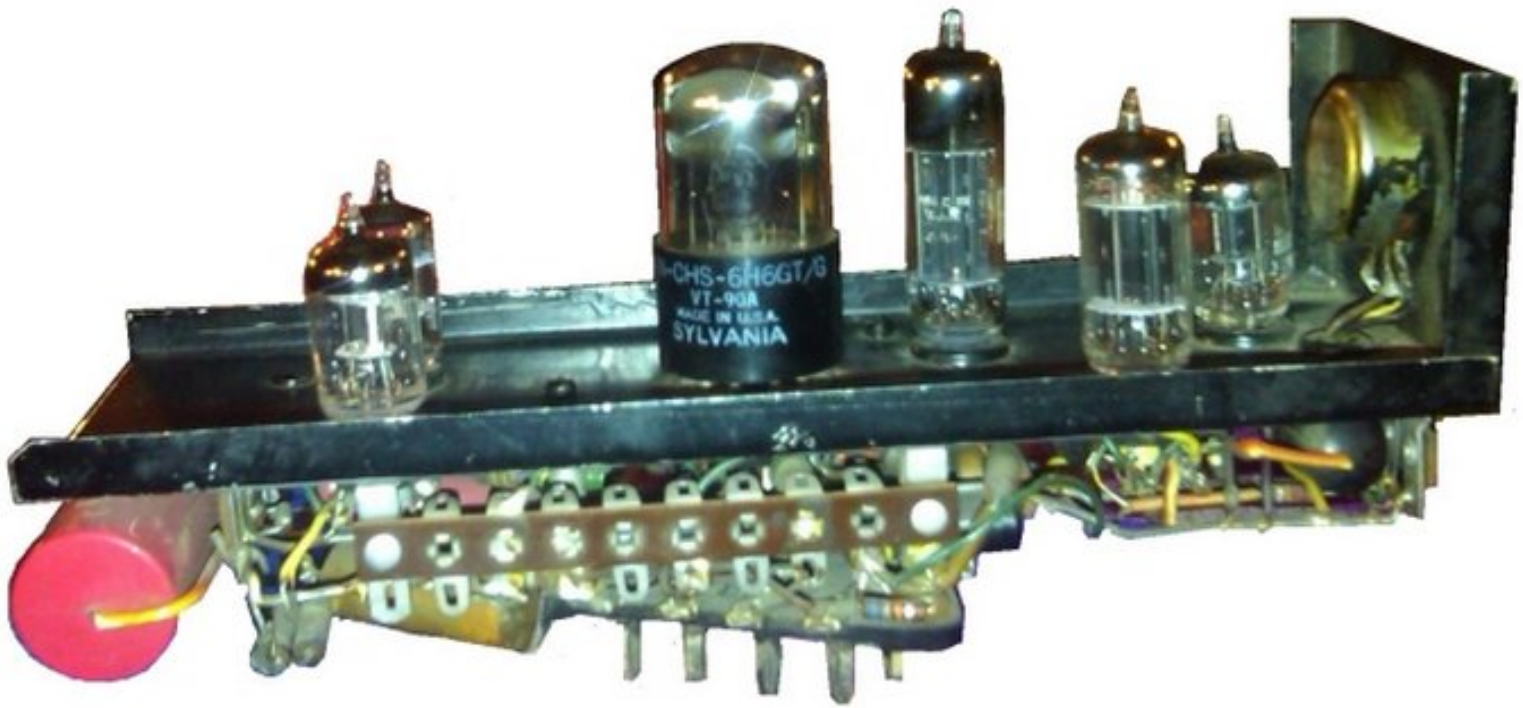
Neural networks vs classical computers

	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 & implementations

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.)



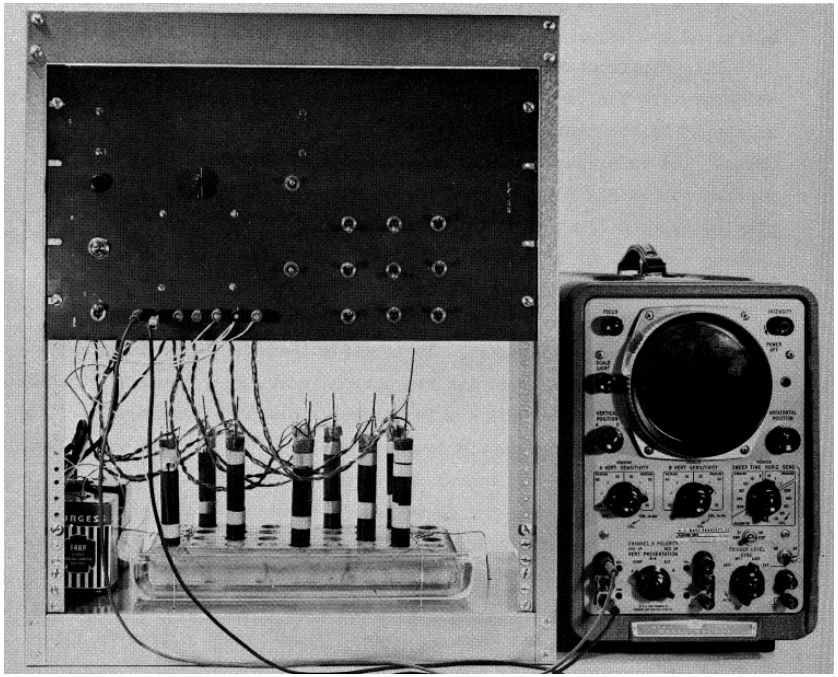
History of neurocomputers

- 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.)

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 ...

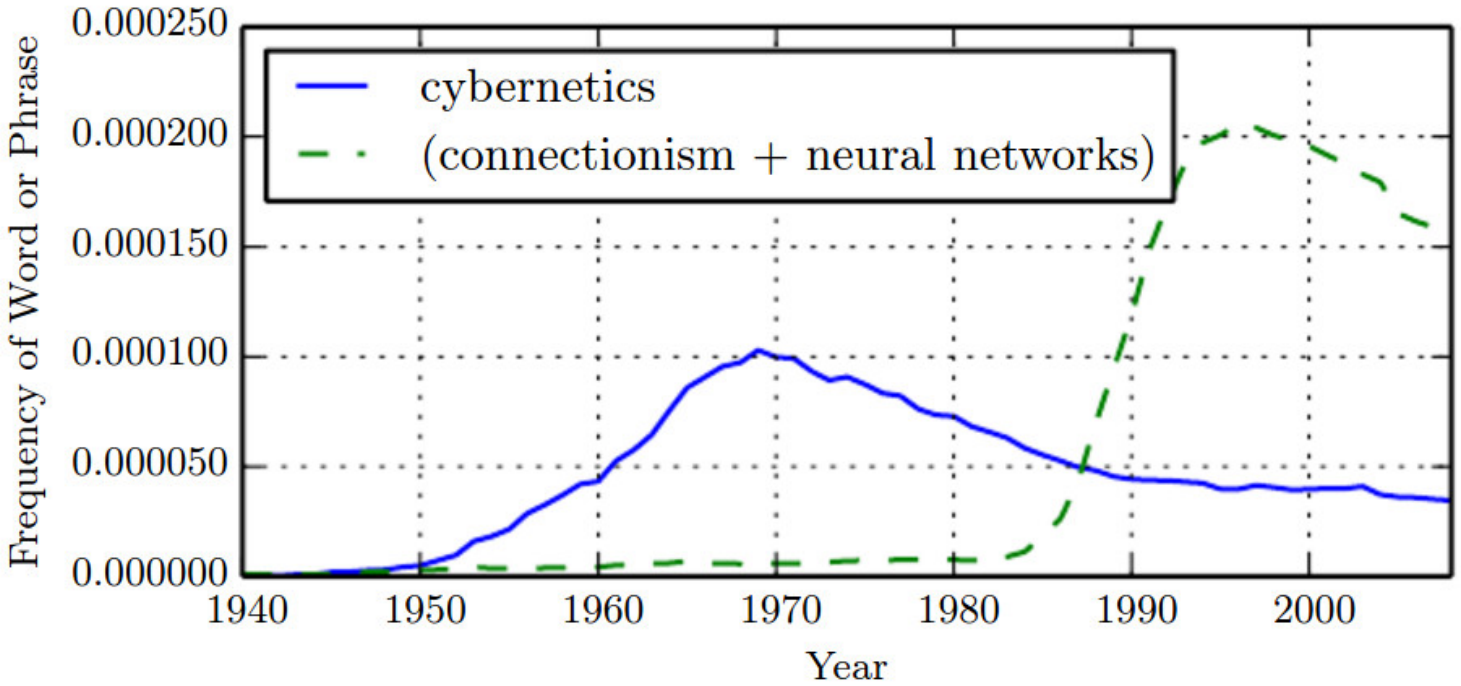
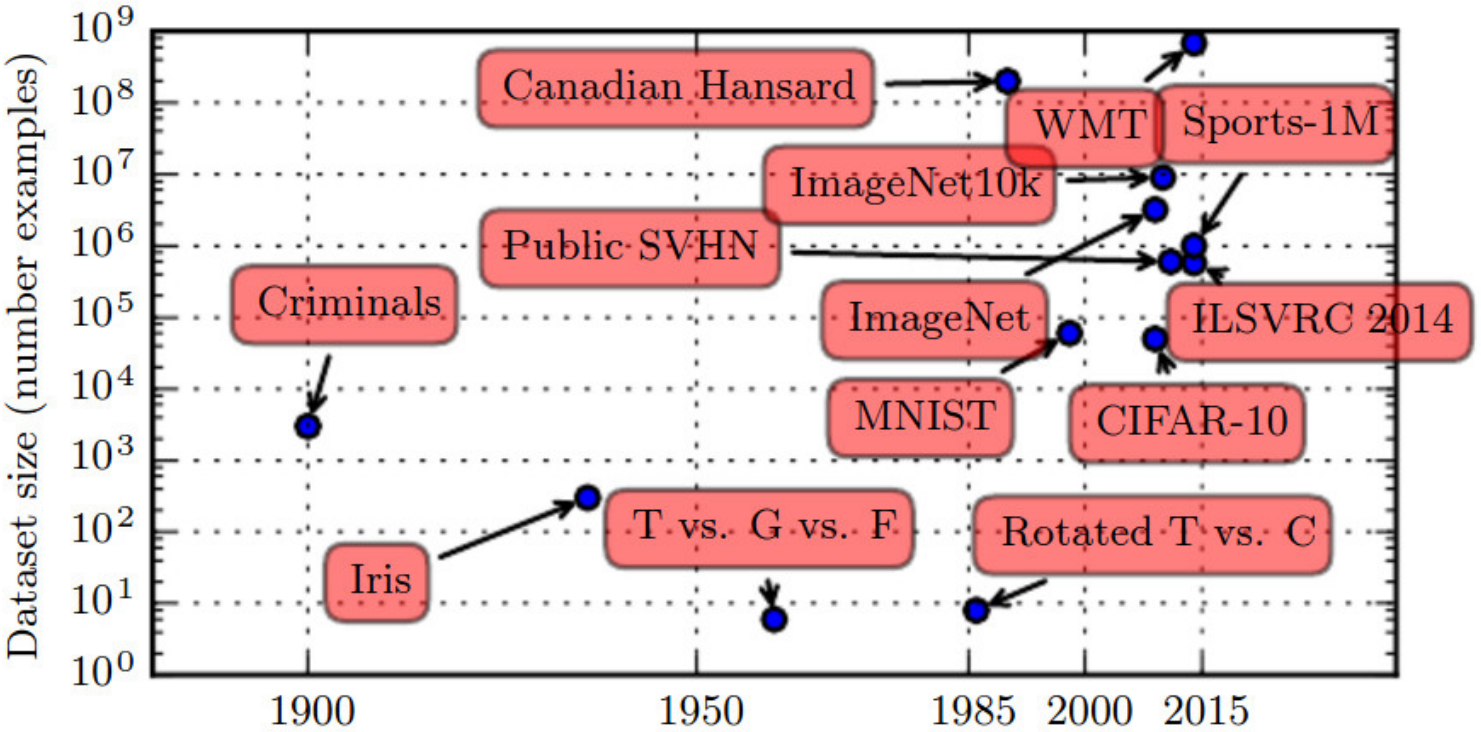


Figure: The figure shows two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks" according to Google Books (the third wave is too recent to appear).

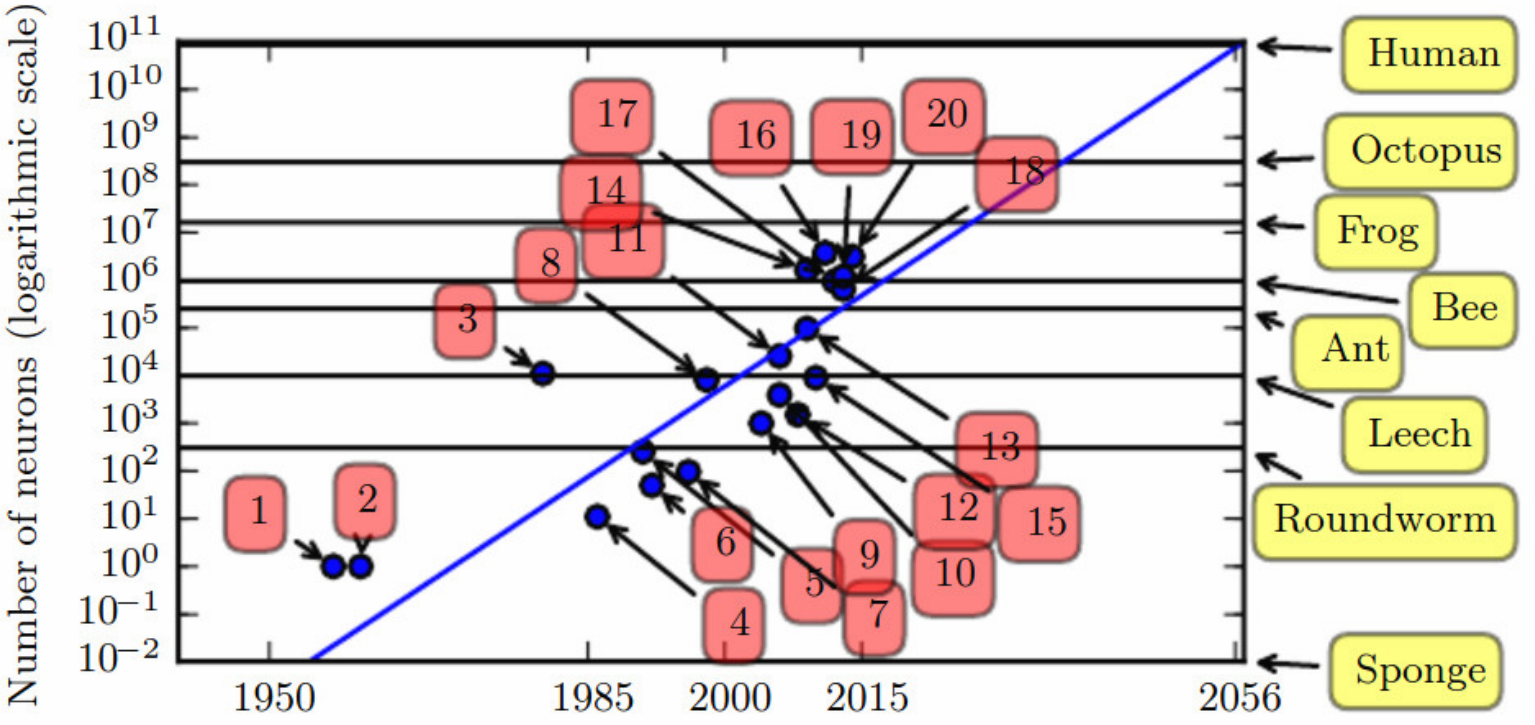
Current hardware – What do we face?

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 ...

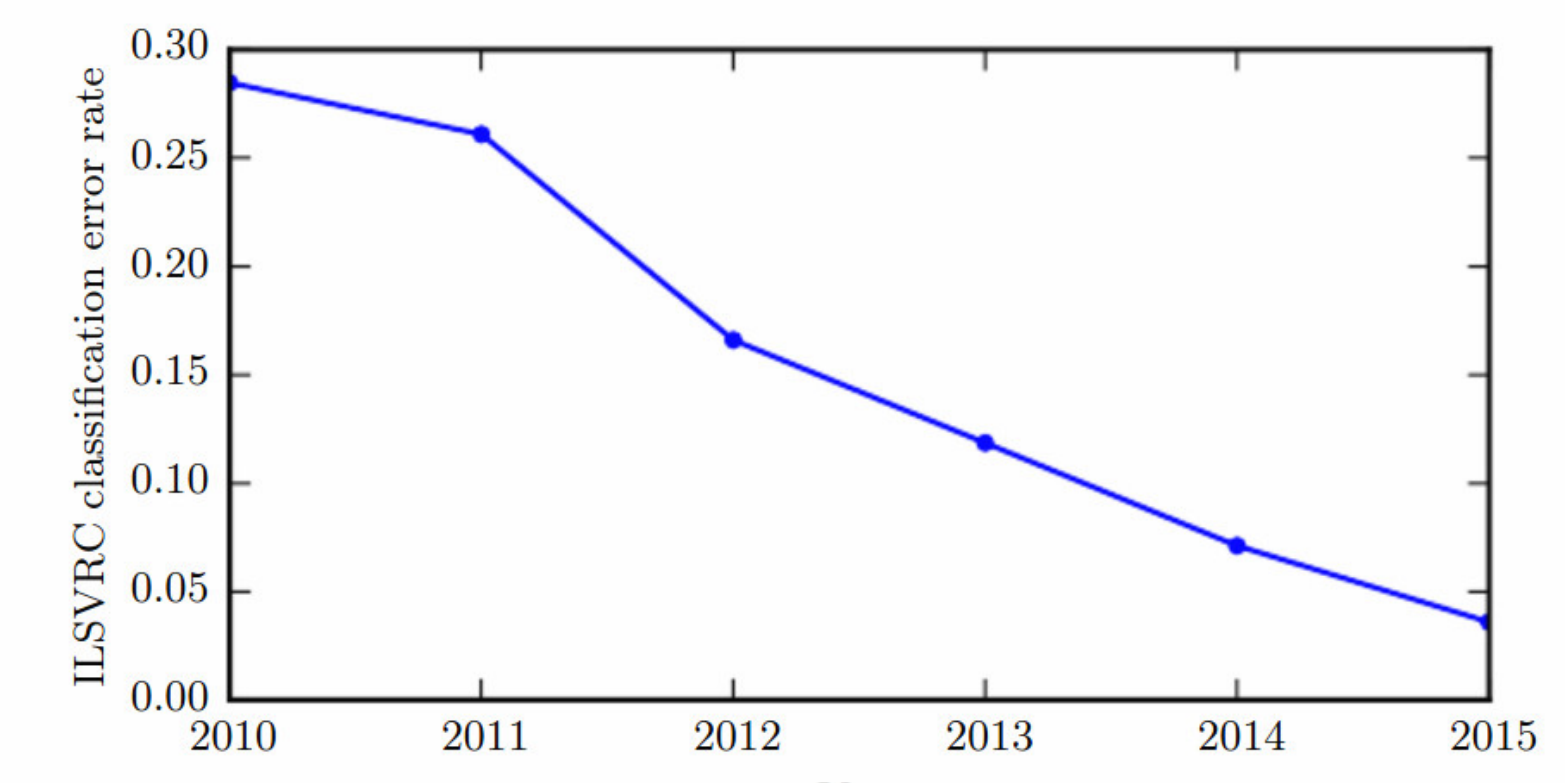


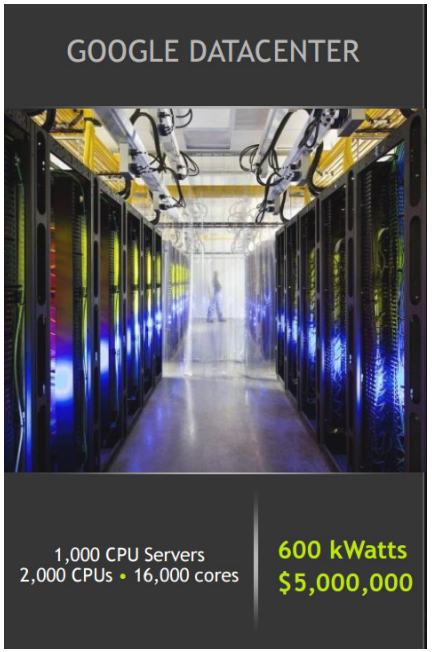
Figure: Since deep networks reached the scale necessary to compete in the ImageNetLarge Scale Visual Recognition Challenge, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky et al. (2014b) and He et al. (2015).

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.



Current hardware

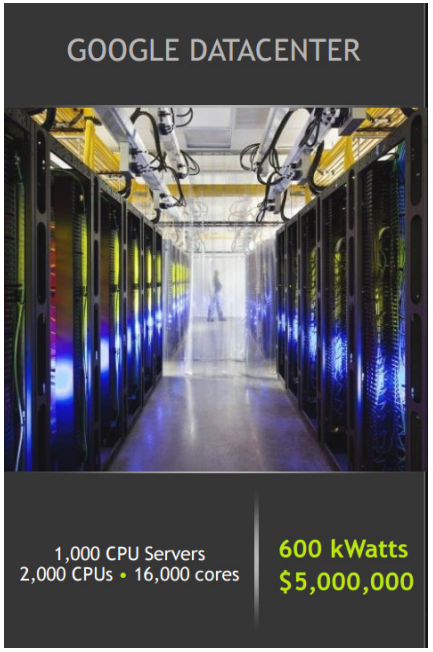
In 2012, Google trained a large network of 1.7 billion weights and 9 layers

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

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

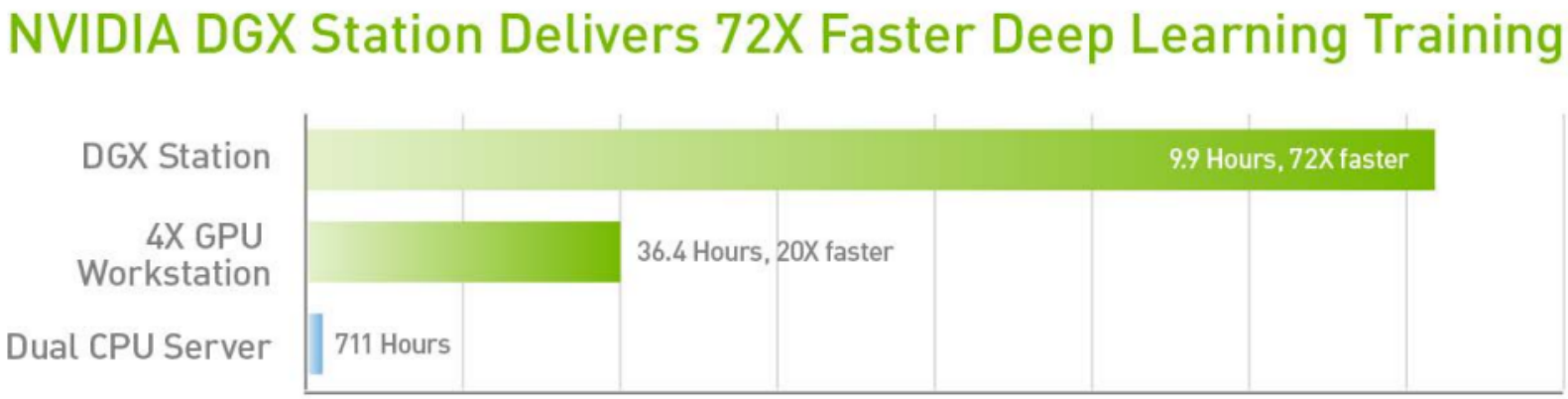
In 2014, similar task performed on Commodity Off-The-Shelf High Performance Computing (COTS HPC) technology: a cluster of GPU servers with Infiniband interconnects and MPI.

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 (dead)**:
  - 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:

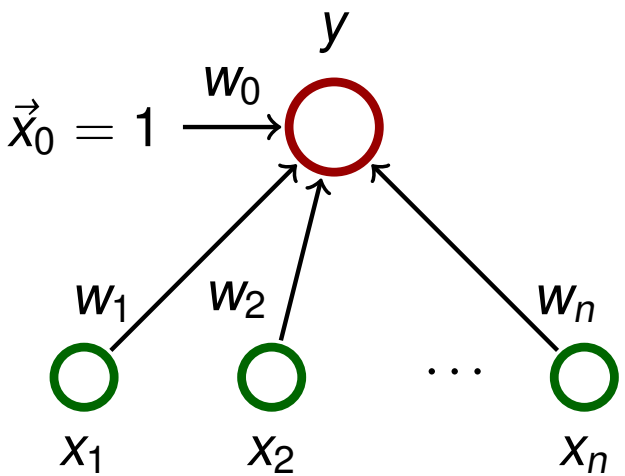
- MATLAB
- R
- STATISTICA
- Weka
- ...

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

Training linear models

Linear regression (ADALINE)

Architecture:



$\vec{w} = (w_0, w_1, \ldots, w_n)$ and $\vec{x} = (x_0, x_1, \ldots, x_n)$ 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: $y[\vec{w}](\vec{x}) = \sigma(\xi) = \vec{w} \cdot \vec{x}$

Linear regression (ADALINE)

Learning:

- Given a **training dataset**

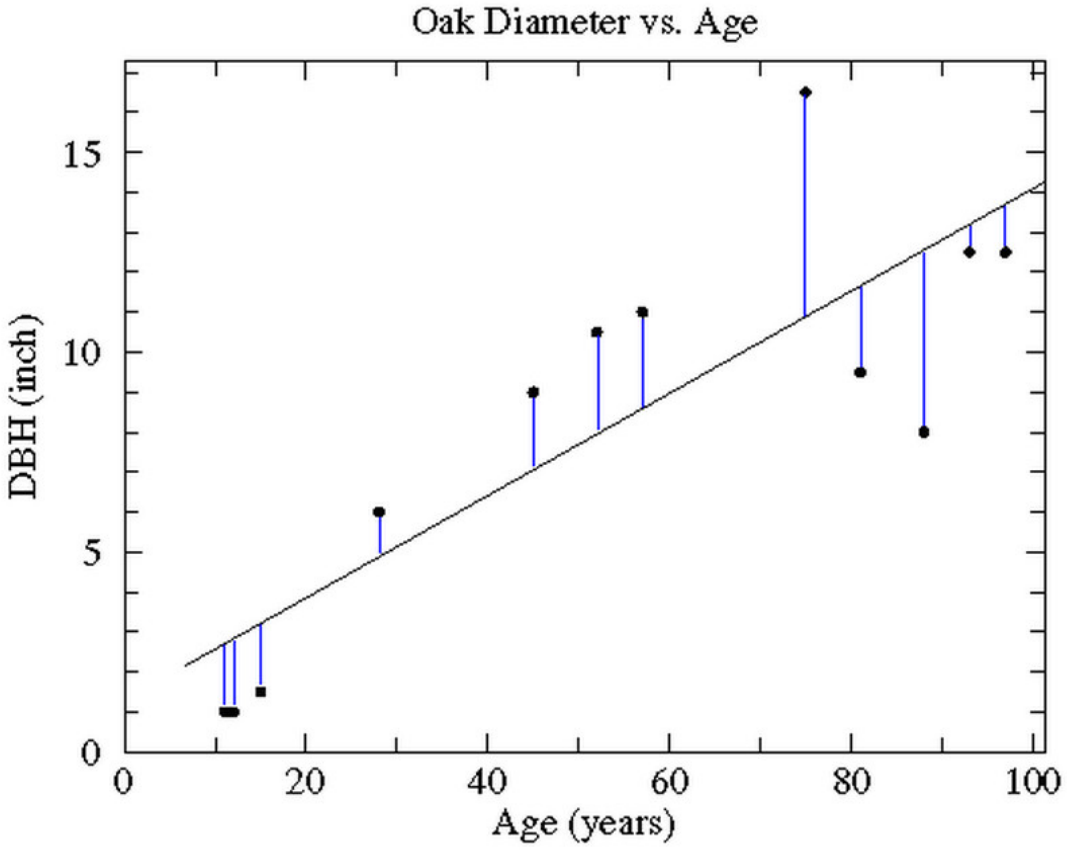
$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

  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

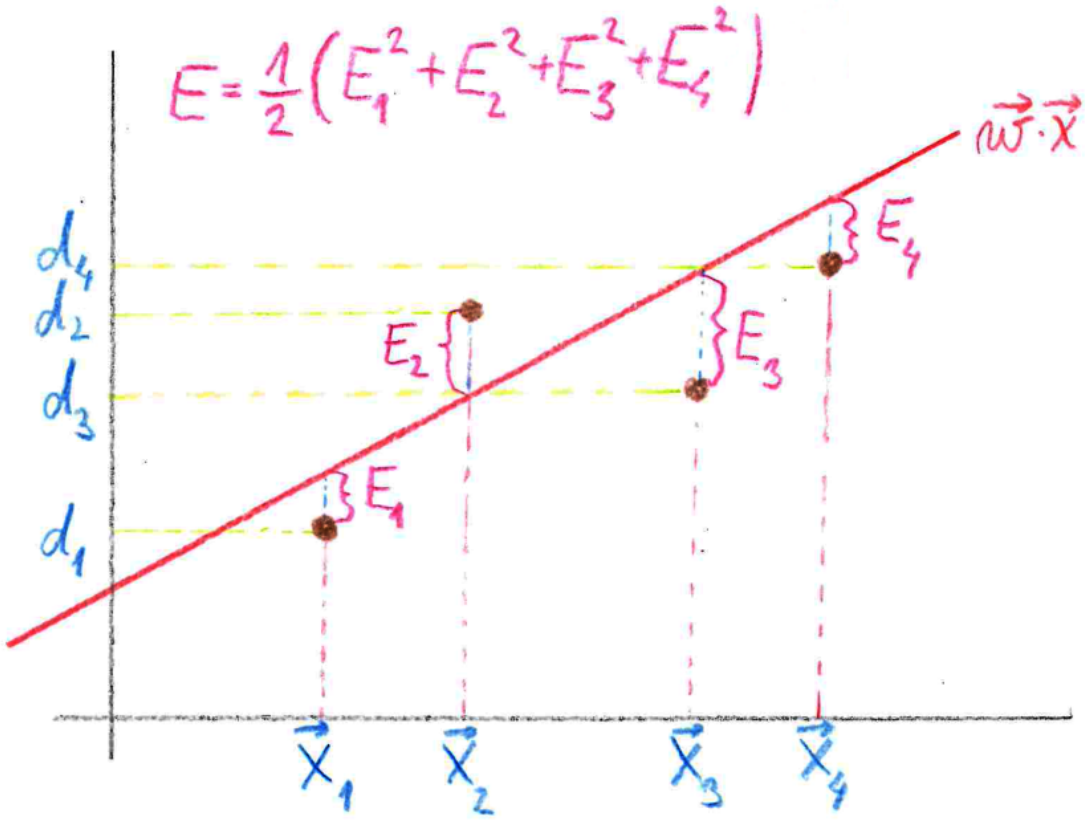
Age (years)	*DBH* (inch)
97	12.5
93	12.5
88	8.0
81	9.5
75	16.5
57	11.0
52	10.5
45	9.0
28	6.0
15	1.5
12	1.0
11	1.0



Linear regression (ADALINE)

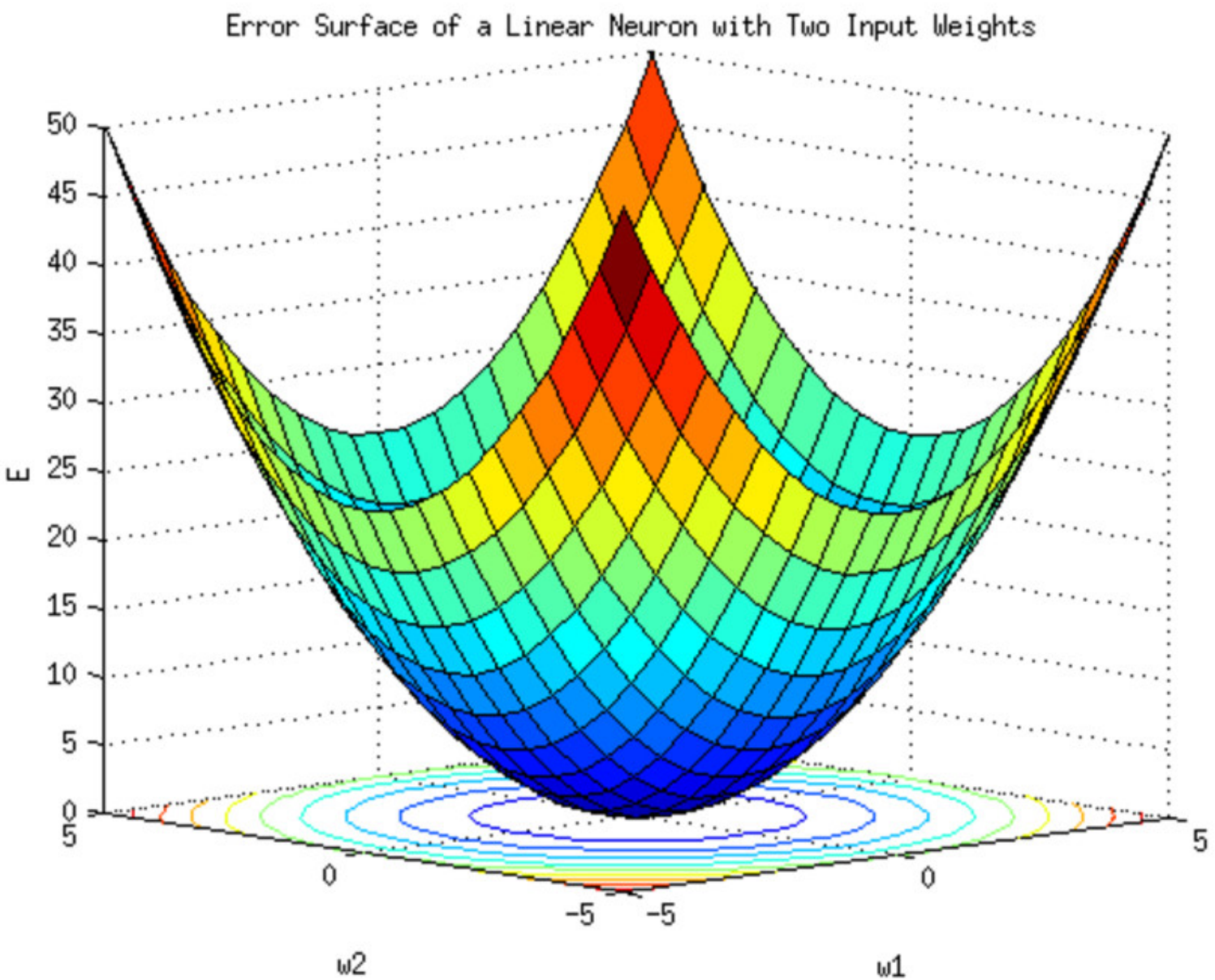
- **Error function:**

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



- 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!

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.

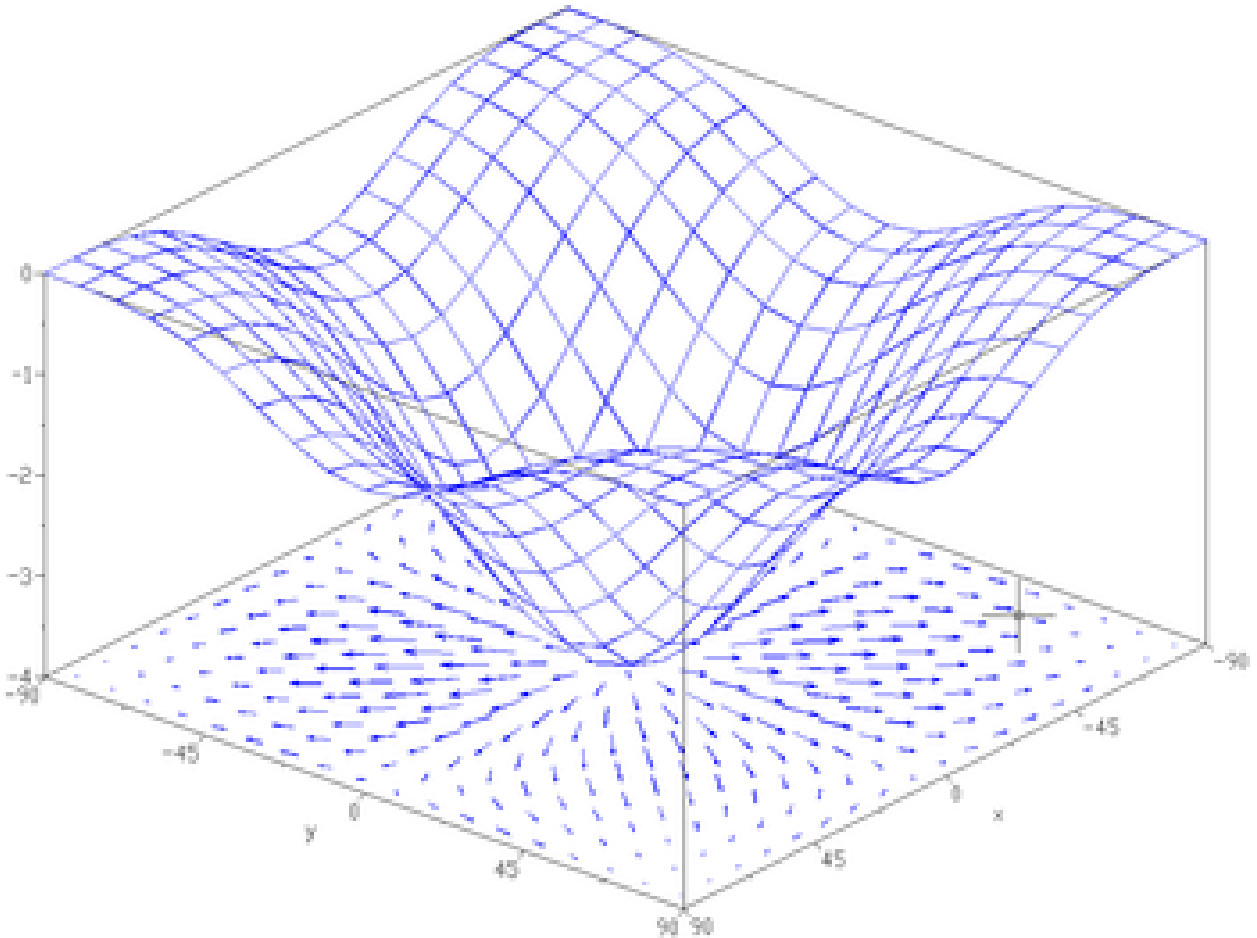
Note that the vectors $\vec{x}_k$ are just parameters of the function E, and are thus fixed!

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

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

$$E(w_0,w_1) = \tfrac{1}{2}[(w_0+w_1\cdot 2-1)^2+(w_0+w_1\cdot 3-2)^2+(w_0+w_1\cdot 4-5)^2]$$

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

$$E(w_0,w_1) = \tfrac{1}{2}[(w_0+w_1\cdot 2-1)^2+(w_0+w_1\cdot 3-2)^2+(w_0+w_1\cdot 4-5)^2]$$

$$\frac{\delta E}{\delta w_0}$$

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

$$E(w_0,w_1) = \tfrac{1}{2}[(w_0+w_1\cdot 2-1)^2+(w_0+w_1\cdot 3-2)^2+(w_0+w_1\cdot 4-5)^2]$$

$$\frac{\delta E}{\delta w_0} = (w_0+w_1\cdot 2-1)\cdot 1+(w_0+w_1\cdot 3-2)\cdot 1+(w_0+w_1\cdot 4-5)\cdot 1$$

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

$$E(w_0,w_1) = \tfrac{1}{2}[(w_0+w_1\cdot 2-1)^2+(w_0+w_1\cdot 3-2)^2+(w_0+w_1\cdot 4-5)^2]$$

$$\frac{\delta E}{\delta w_0} = (w_0+w_1\cdot 2-1)\cdot 1+(w_0+w_1\cdot 3-2)\cdot 1+(w_0+w_1\cdot 4-5)\cdot 1$$

$$\frac{\delta E}{\delta w_1}$$

Gradient of the error function

First, consider $n = 1$.

Then the model is $y = w_0 + w_1 \cdot x$.

Consider a concrete training set:

$$\begin{aligned} \mathcal{T} &= \{((1,2),1),((1,3),2),((1,4),5)\} \\ &= ((x_{10},x_{11}),d_1),((x_{20},x_{21}),d_2),((x_{30},x_{31}),d_3) \end{aligned}$$

$$E(w_0,w_1) = \tfrac{1}{2}[(w_0+w_1\cdot 2-1)^2+(w_0+w_1\cdot 3-2)^2+(w_0+w_1\cdot 4-5)^2]$$

$$\frac{\delta E}{\delta w_0} = (w_0+w_1\cdot 2-1)\cdot 1+(w_0+w_1\cdot 3-2)\cdot 1+(w_0+w_1\cdot 4-5)\cdot 1$$

$$\frac{\delta E}{\delta w_1} = (w_0+w_1\cdot 2-1)\cdot 2+(w_0+w_1\cdot 3-2)\cdot 3+(w_0+w_1\cdot 4-5)\cdot 4$$

Gradient of the error function

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

Gradient of the error function

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

Gradient of the error function

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

Gradient of the error function

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

Gradient of the error function

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

Thus

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

Linear regression - learning

Batch algorithm (gradient descent):

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

Linear regression - 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

Linear regression - 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*.

Linear regression - 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:

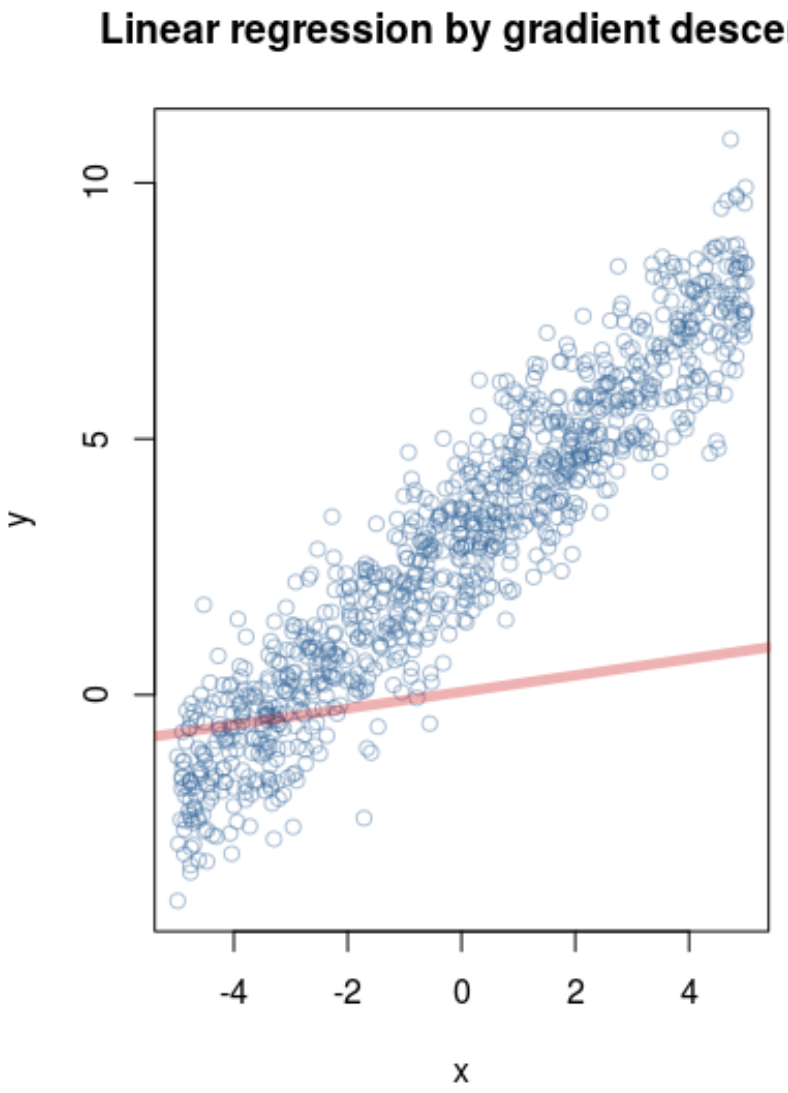
$$\begin{aligned} \vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \vec{x}_k - d_k \right) \cdot \vec{x}_k \end{aligned}$$

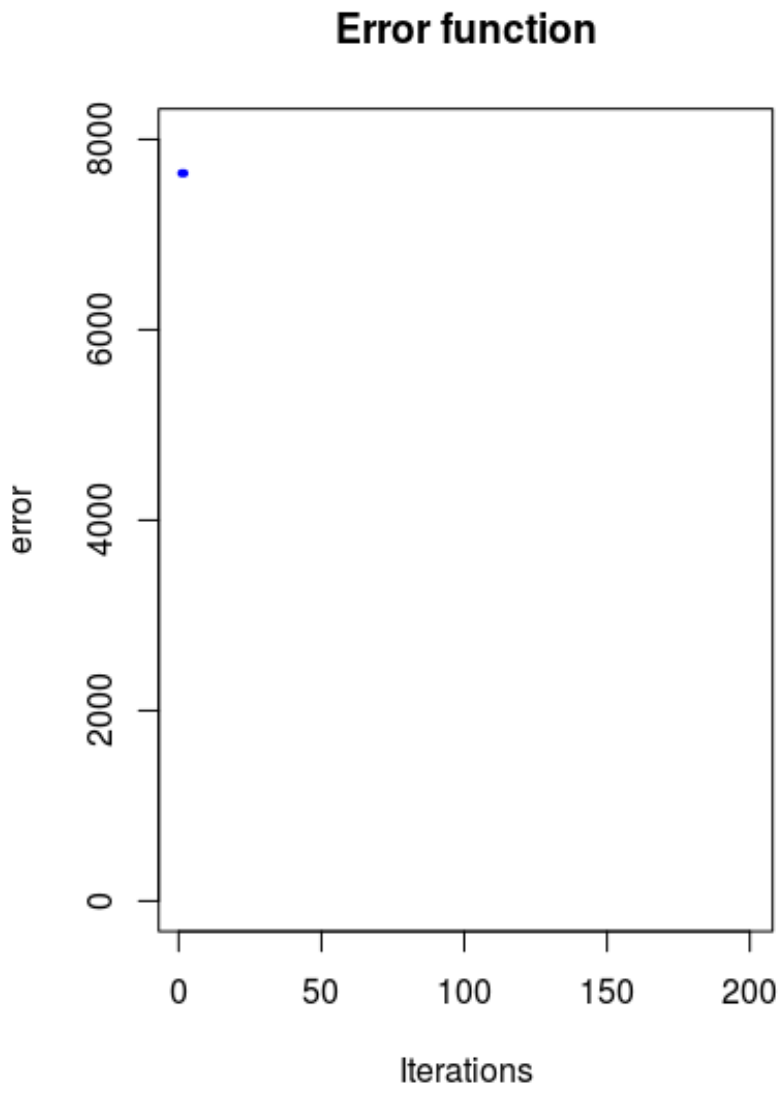
Here $k = (t \mod p) + 1$ and $0 < \varepsilon \leq 1$ is a *learning rate*.

Proposition

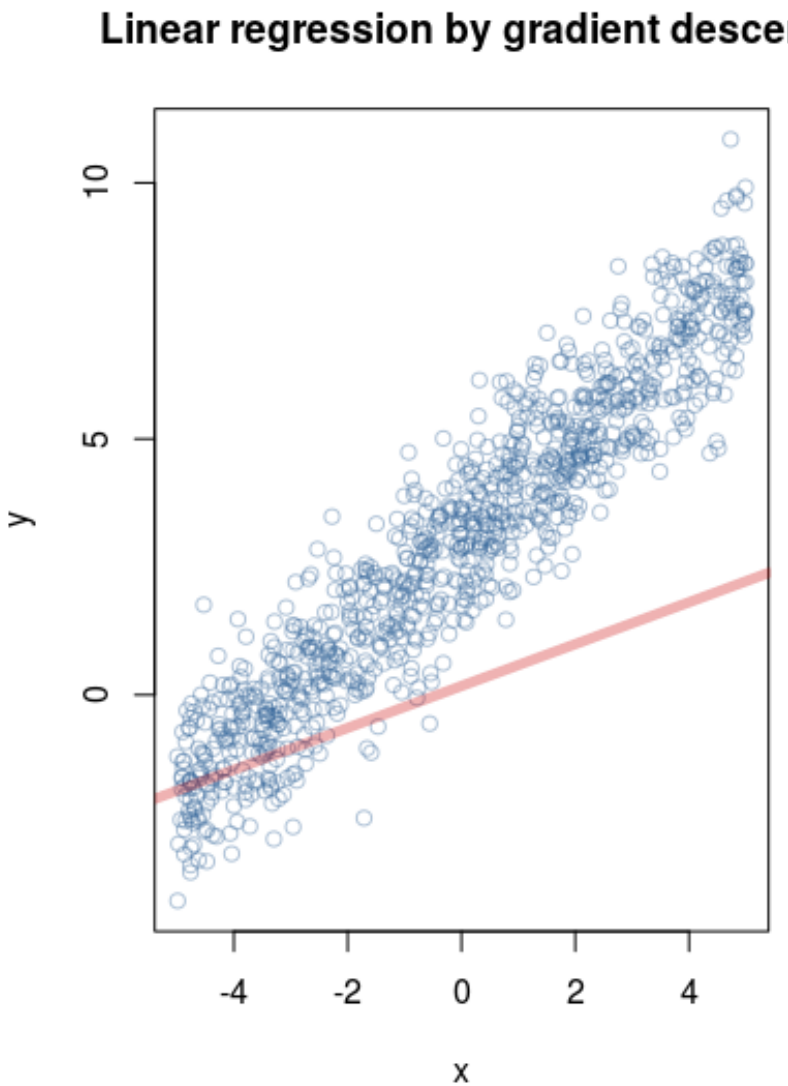
For sufficiently small $\varepsilon > 0$ the sequence $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$ converges (componentwise) to the global minimum of E (i.e. to the vector $\vec{w}$ satisfying $\nabla E(\vec{w}) = \vec{0}$).

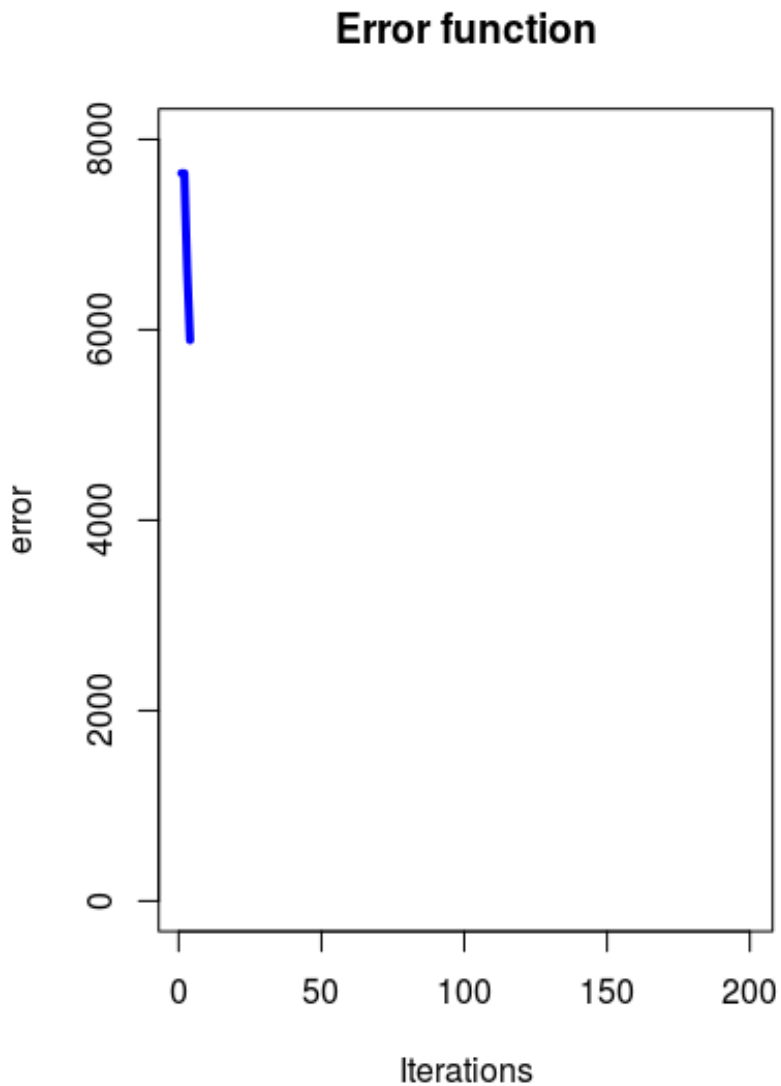
Linear regression - animation



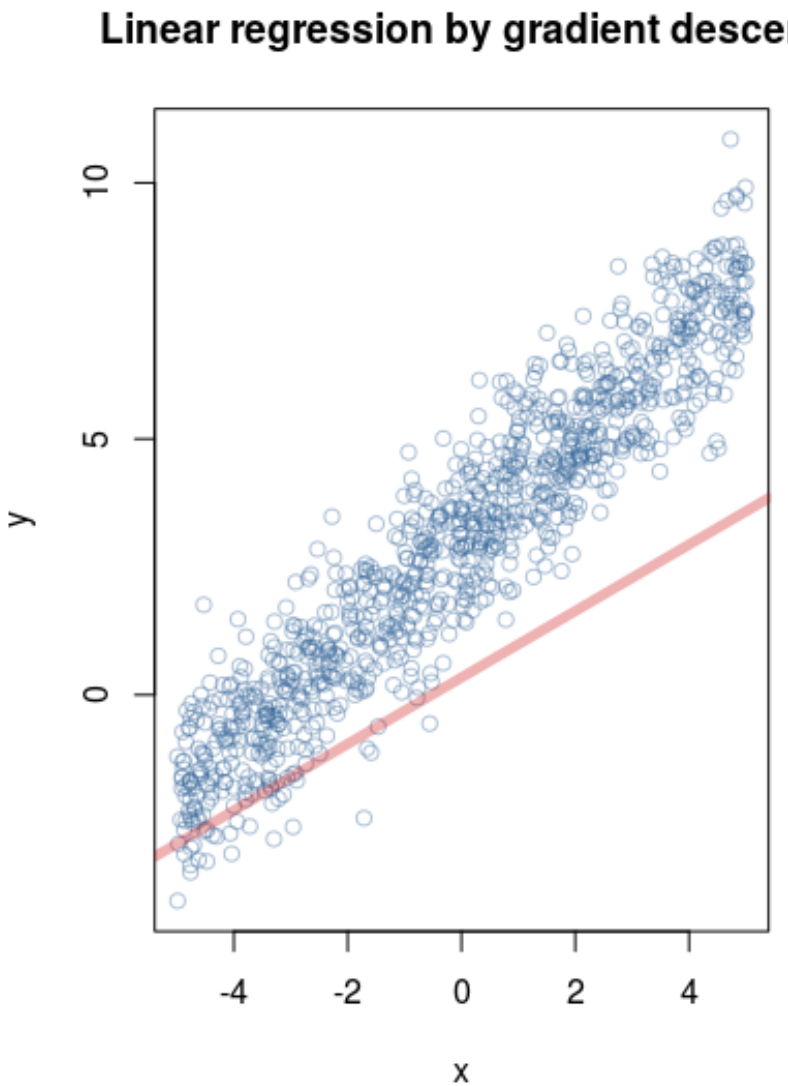


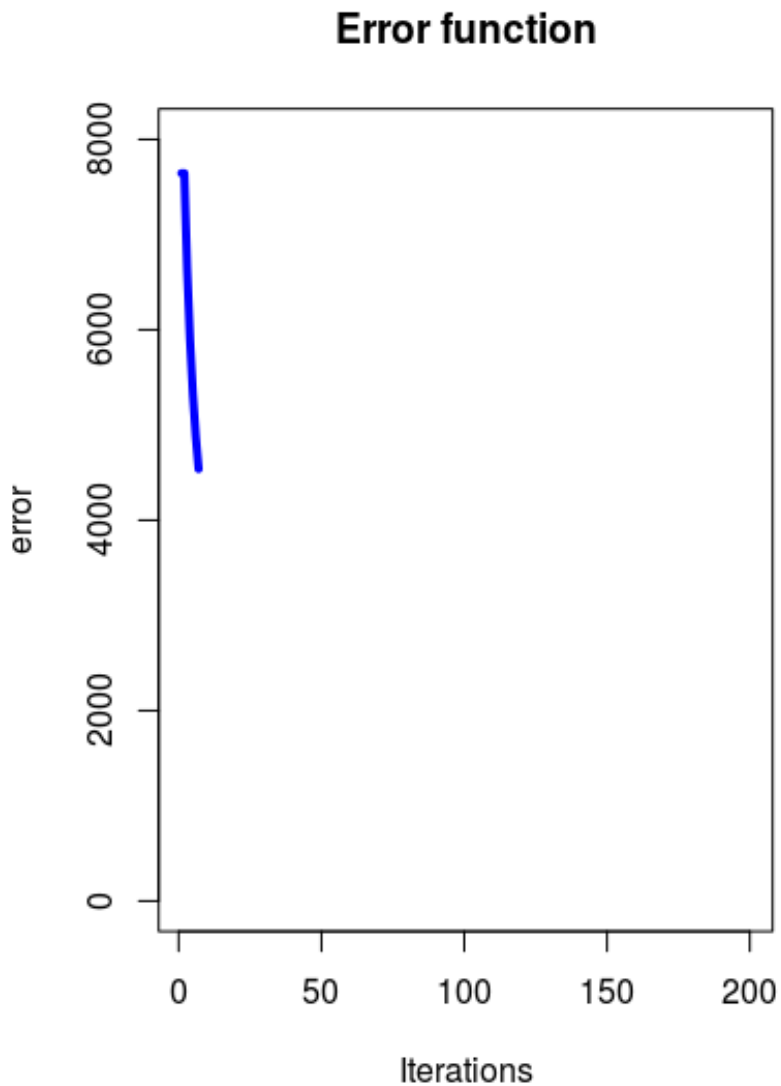
Linear regression - animation



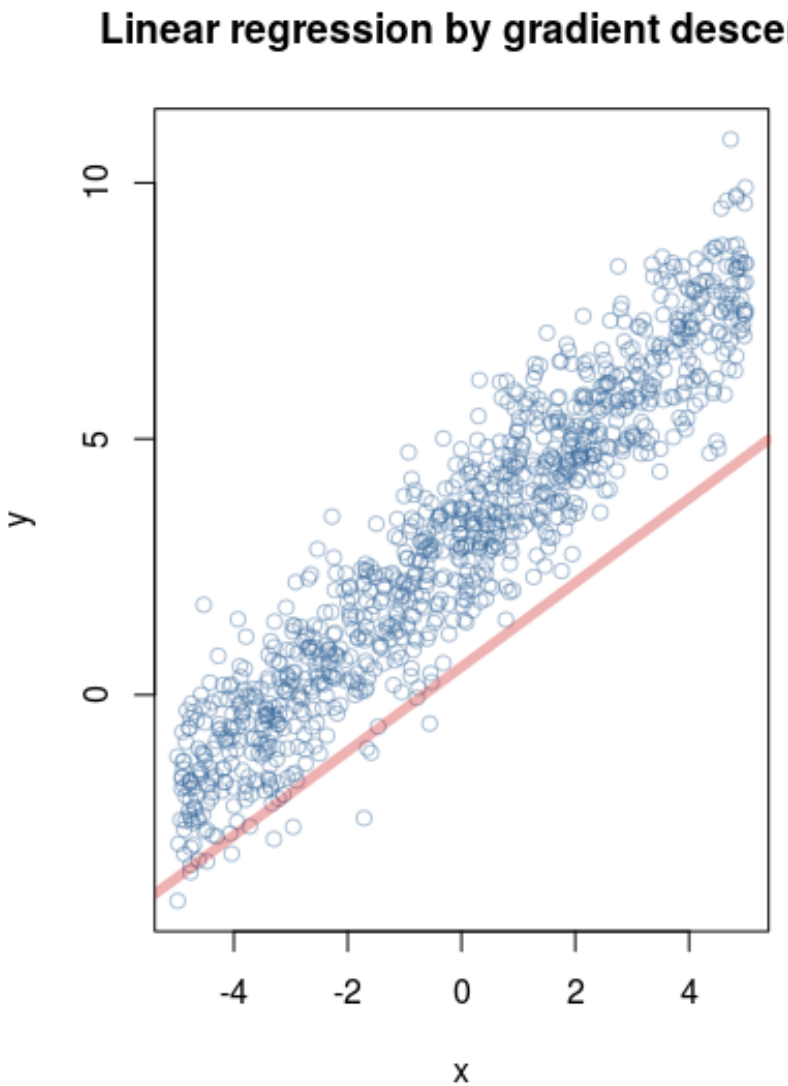


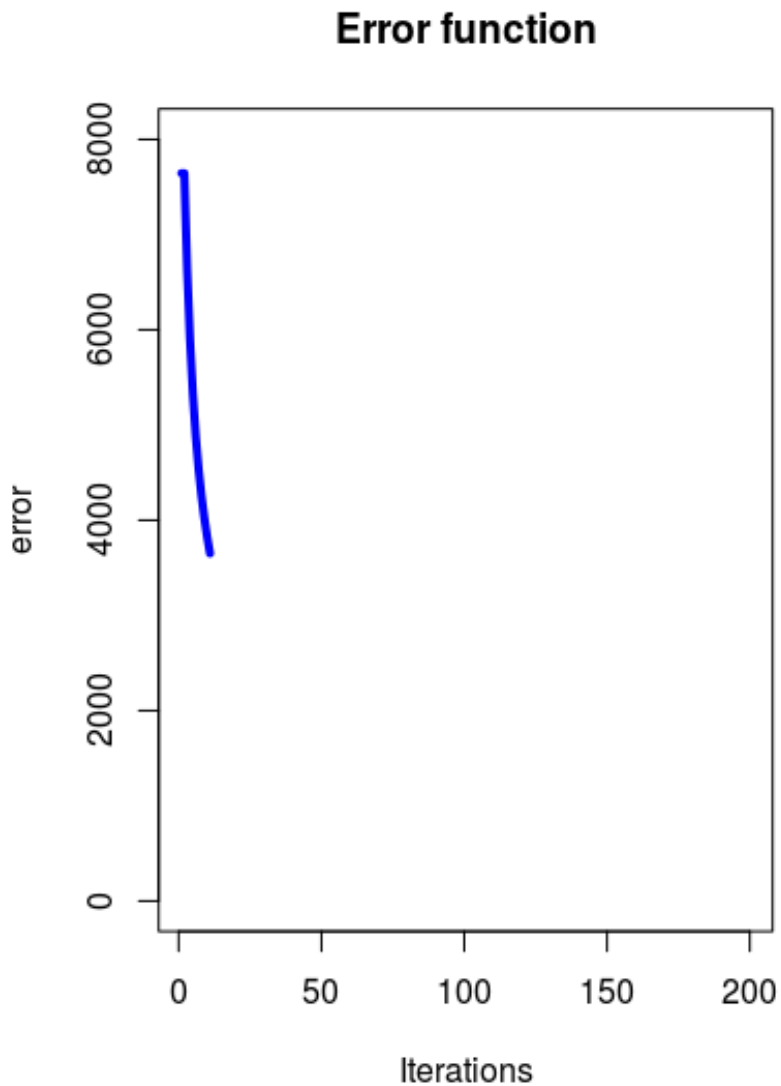
Linear regression - animation



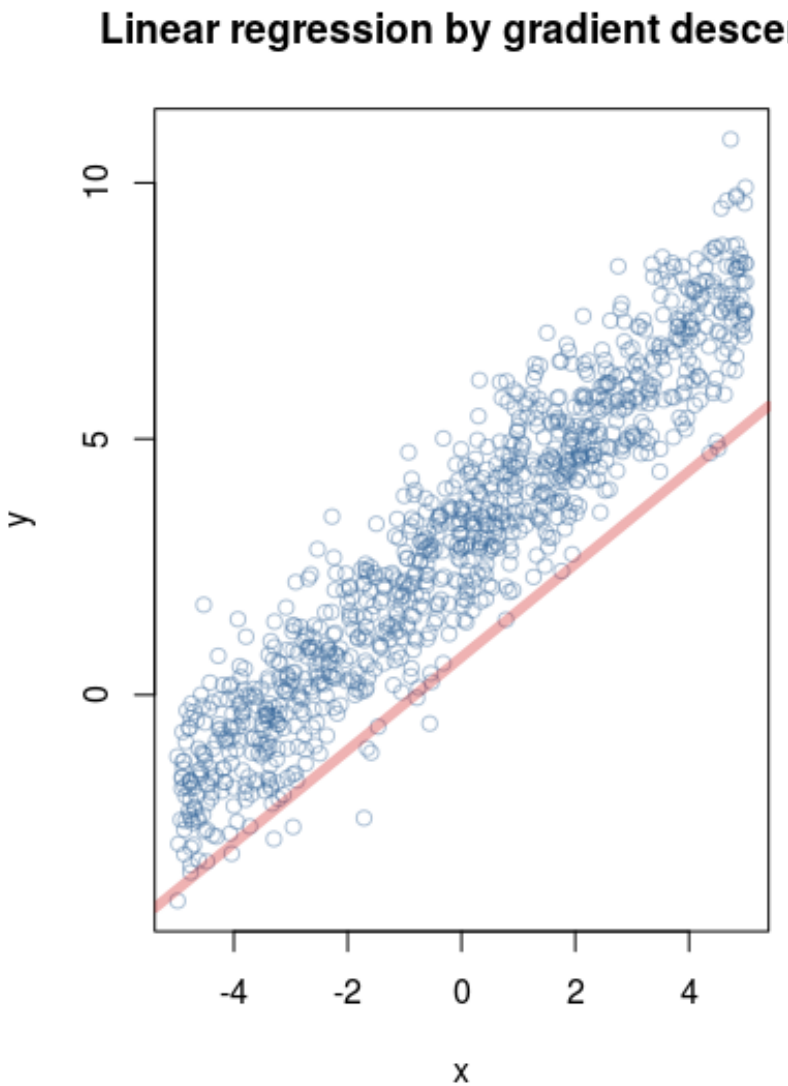


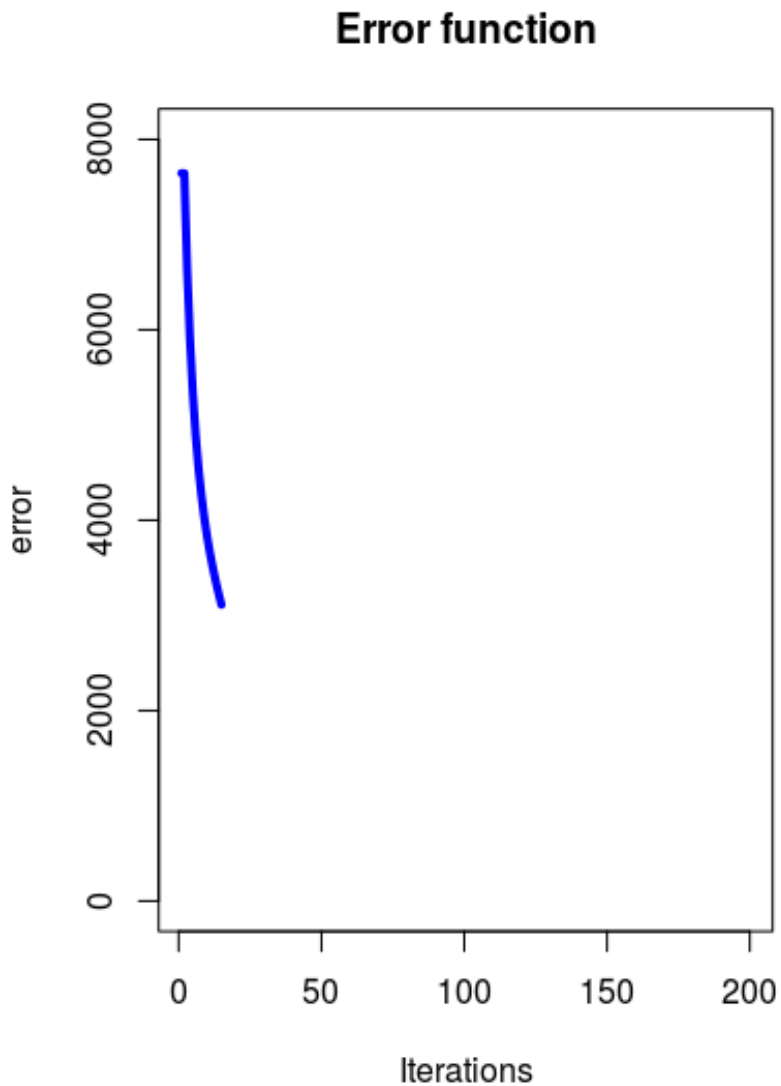
Linear regression - animation



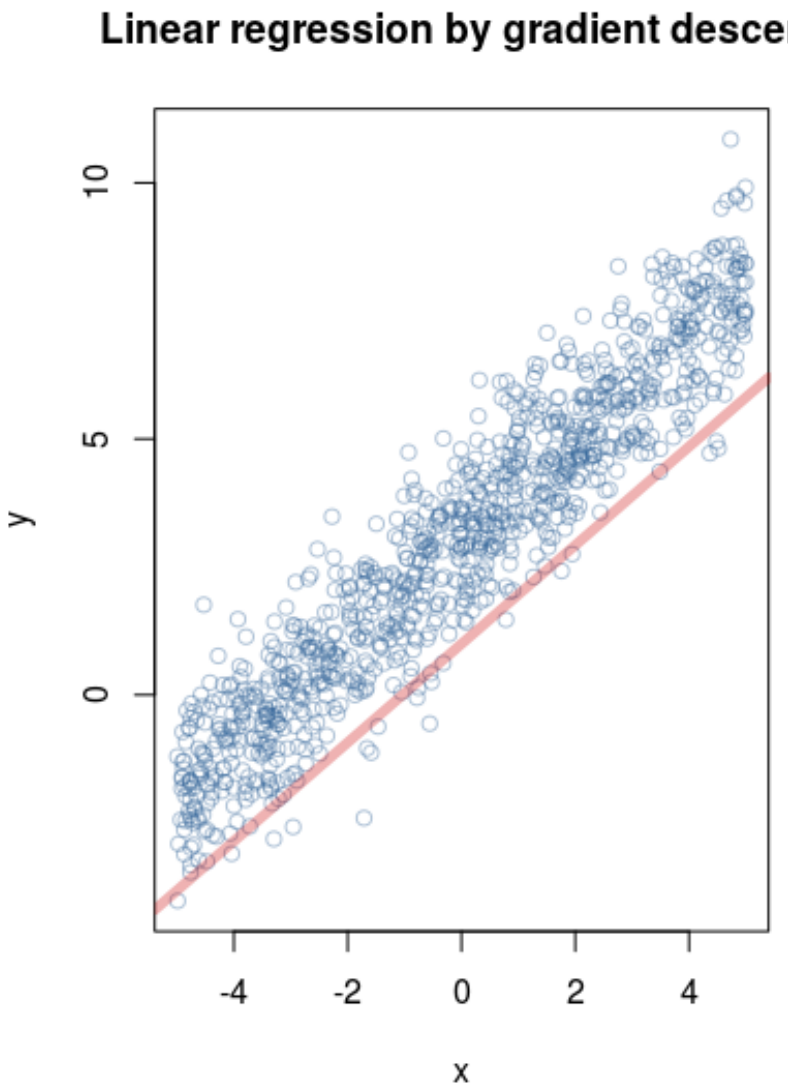


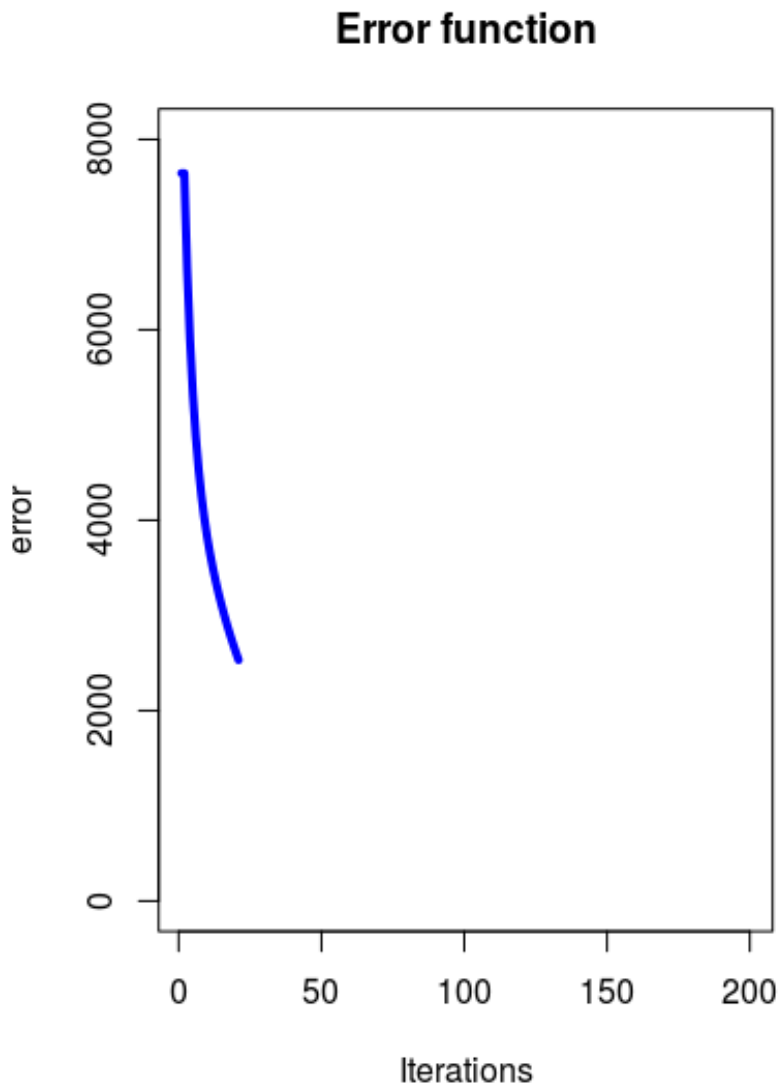
Linear regression - animation



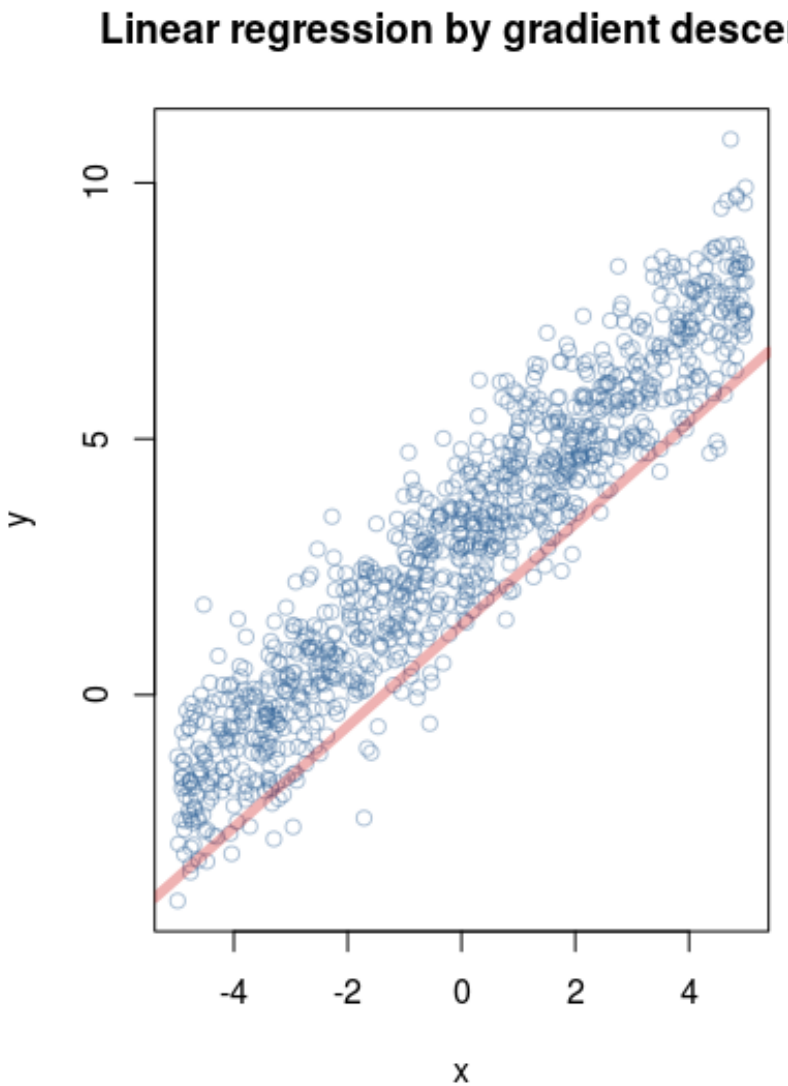


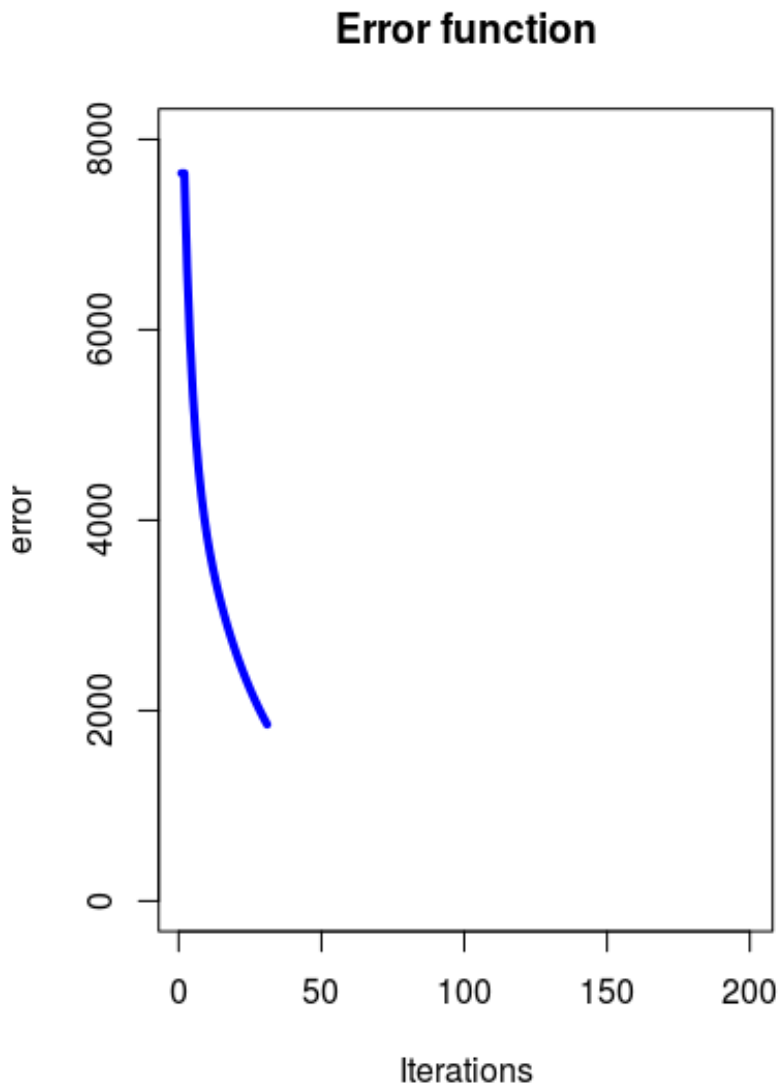
Linear regression - animation



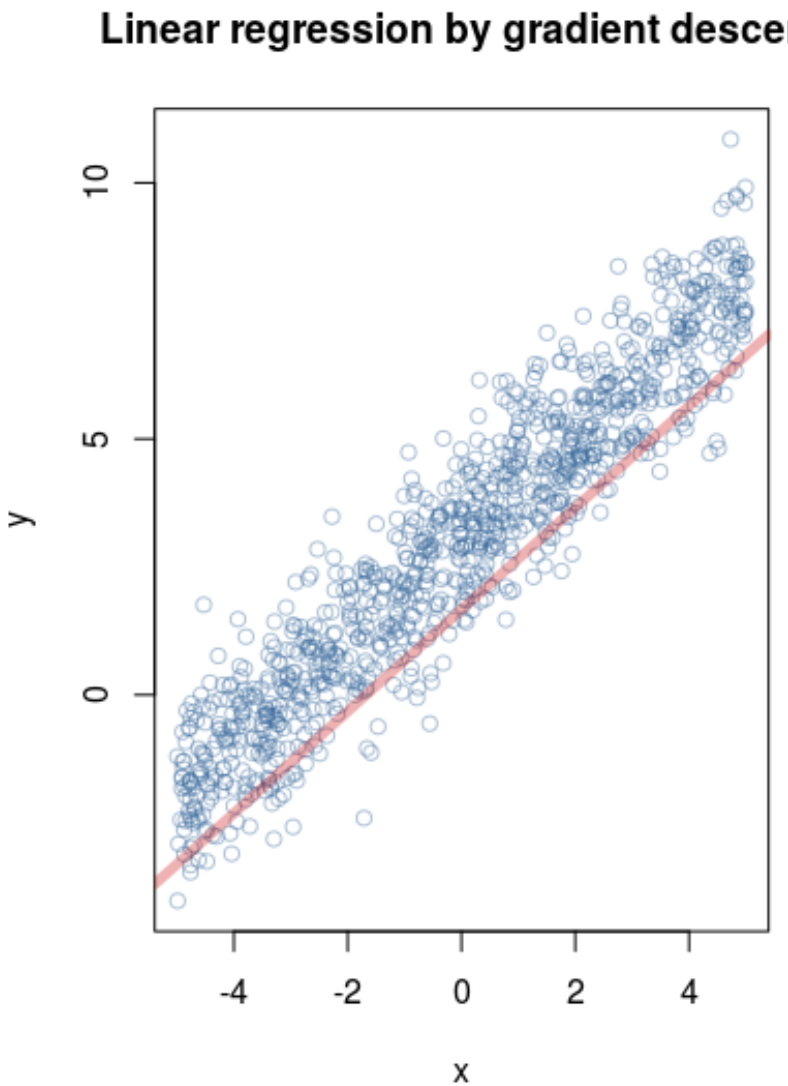


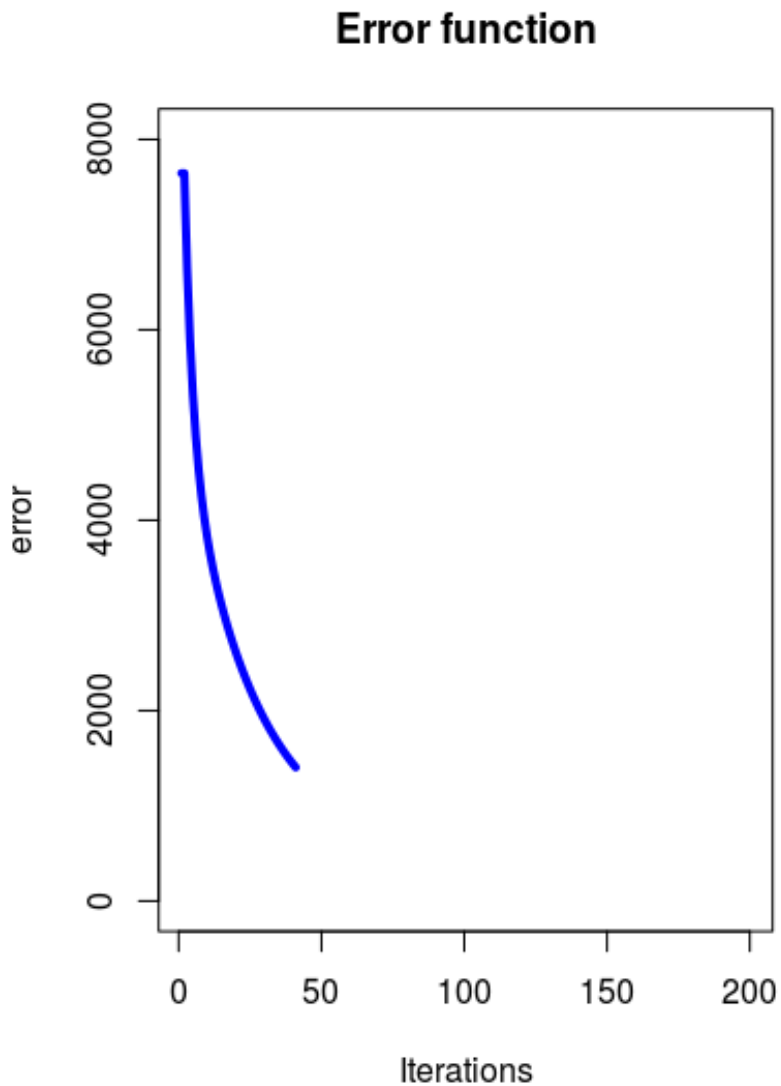
Linear regression - animation



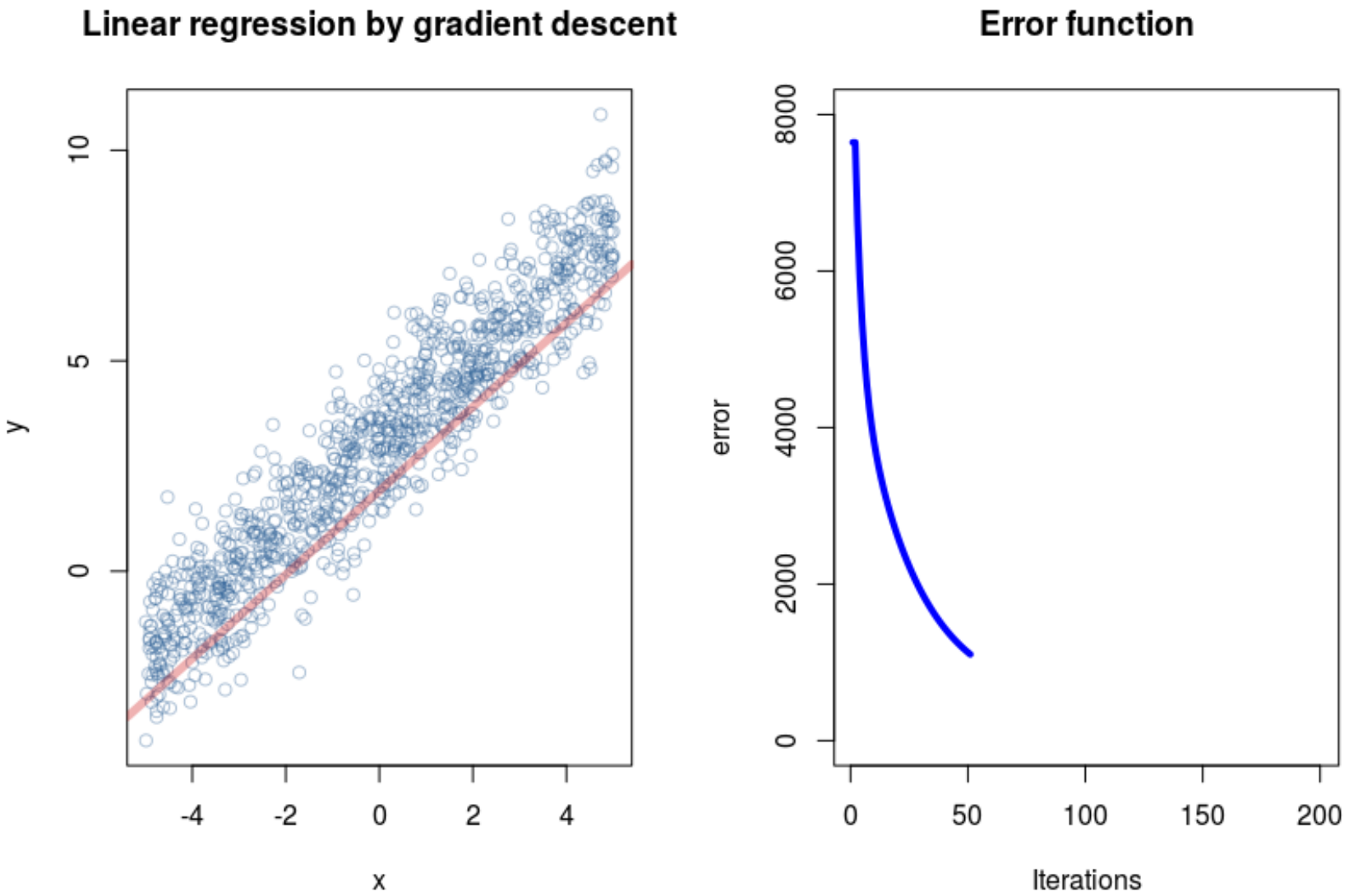


Linear regression - animation

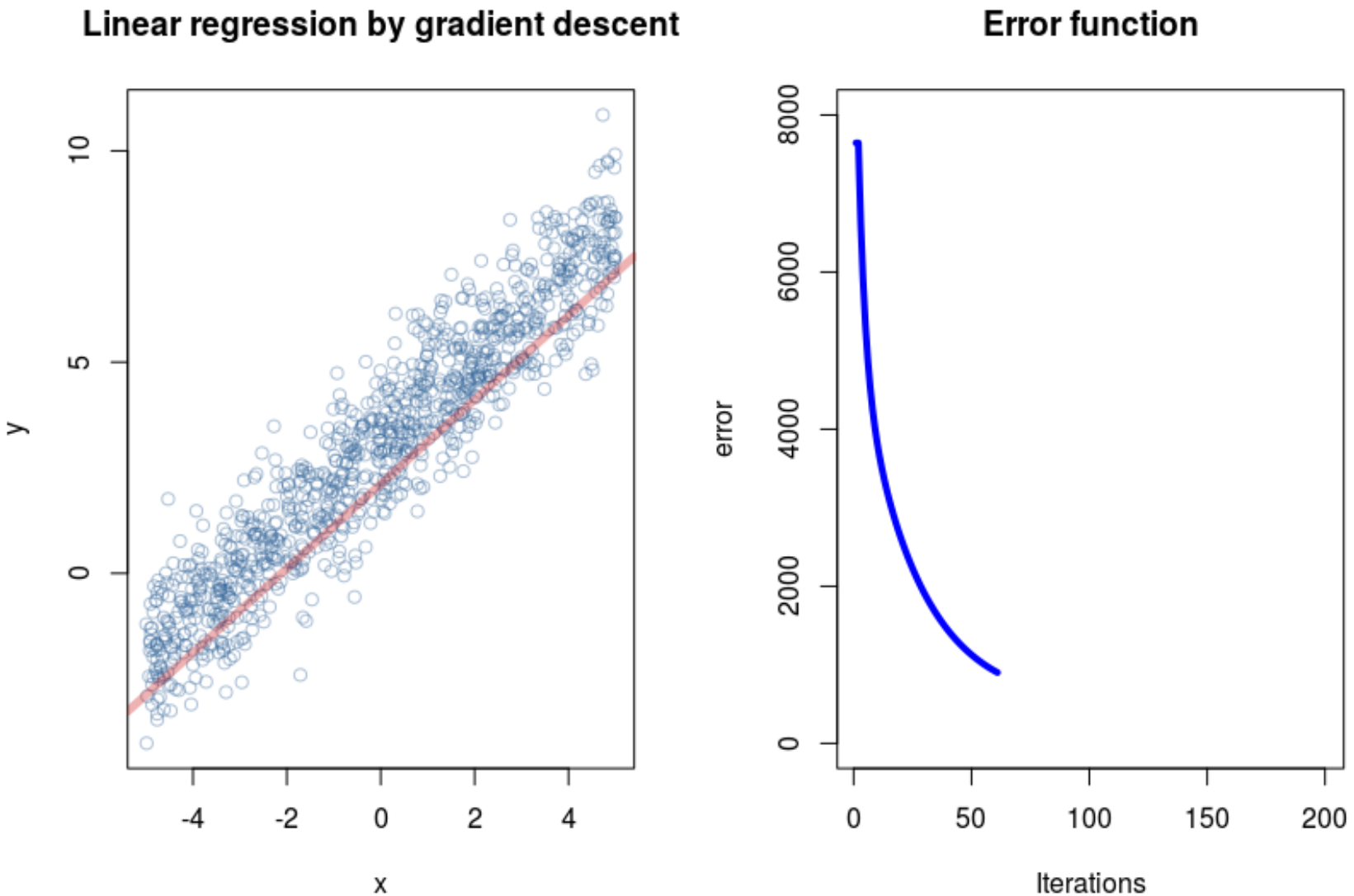




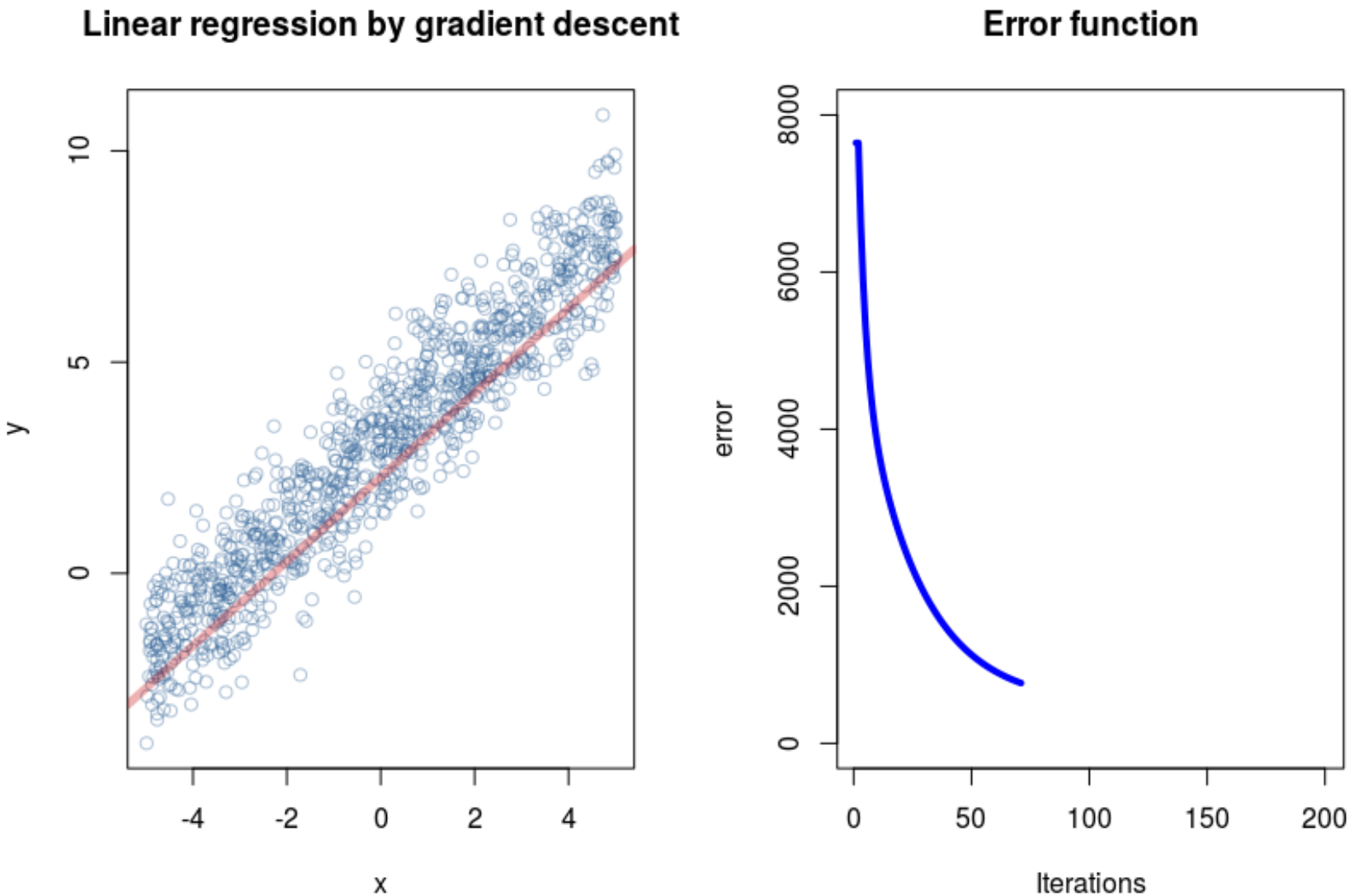
Linear regression - animation



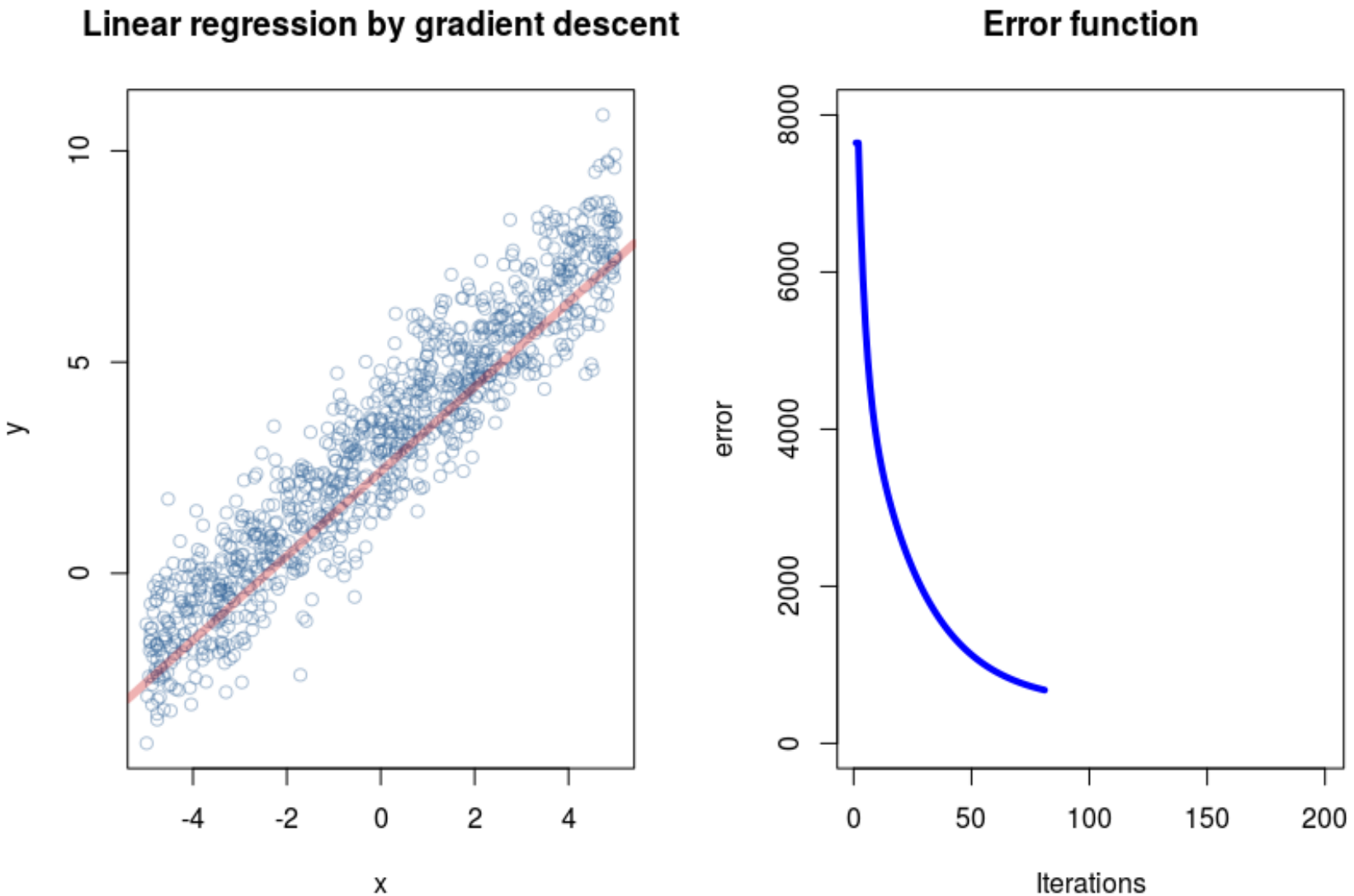
Linear regression - animation



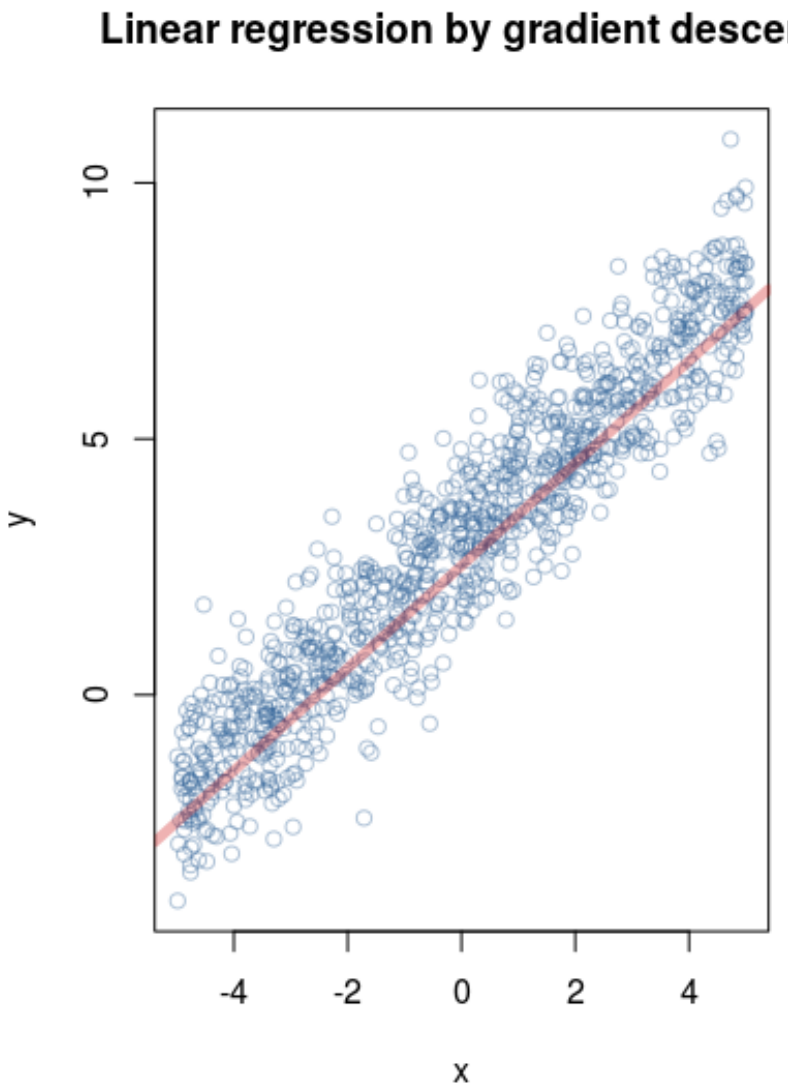
Linear regression - animation

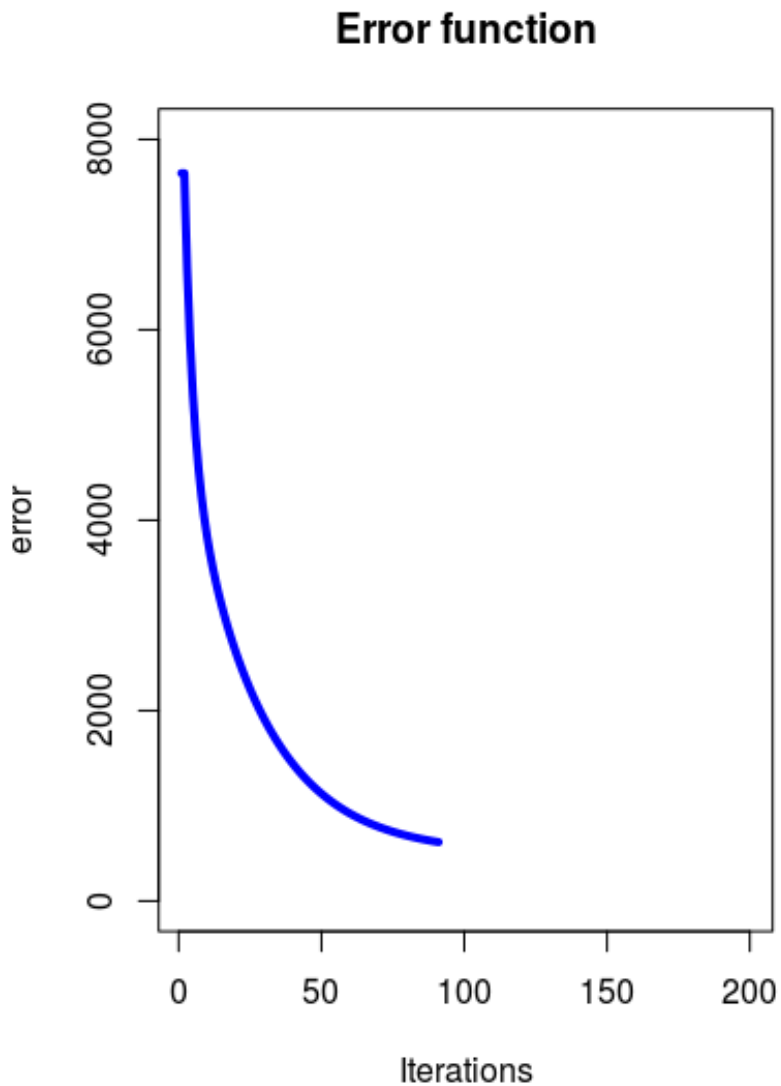


Linear regression - animation

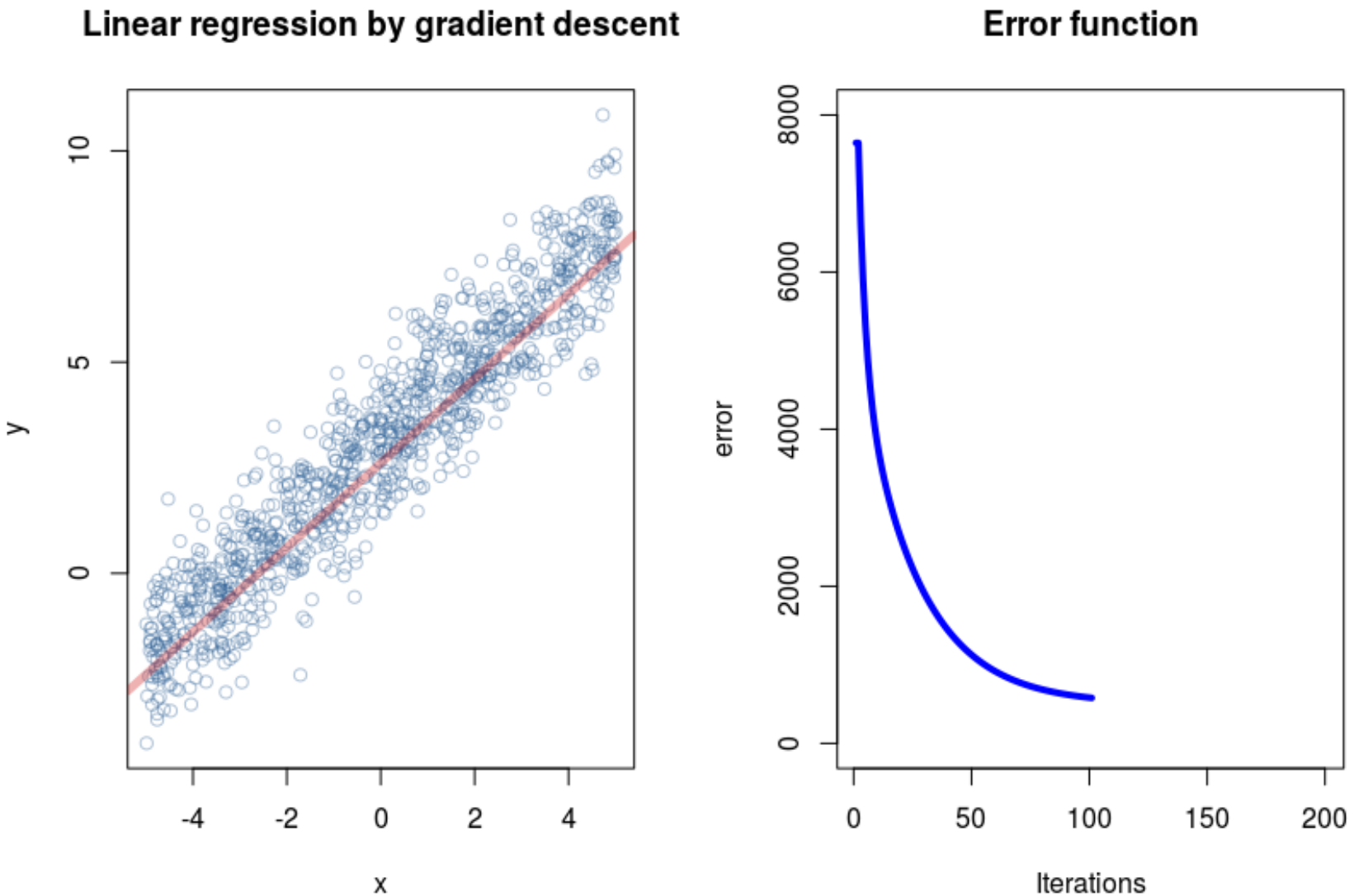


Linear regression - animation

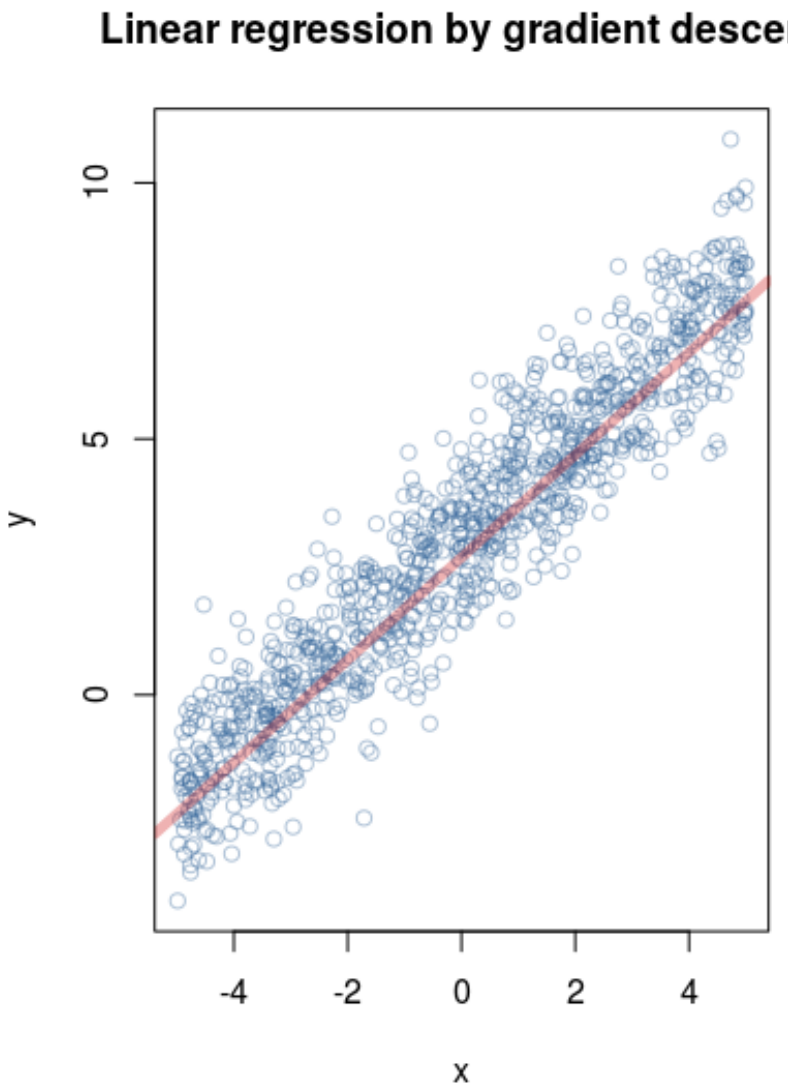


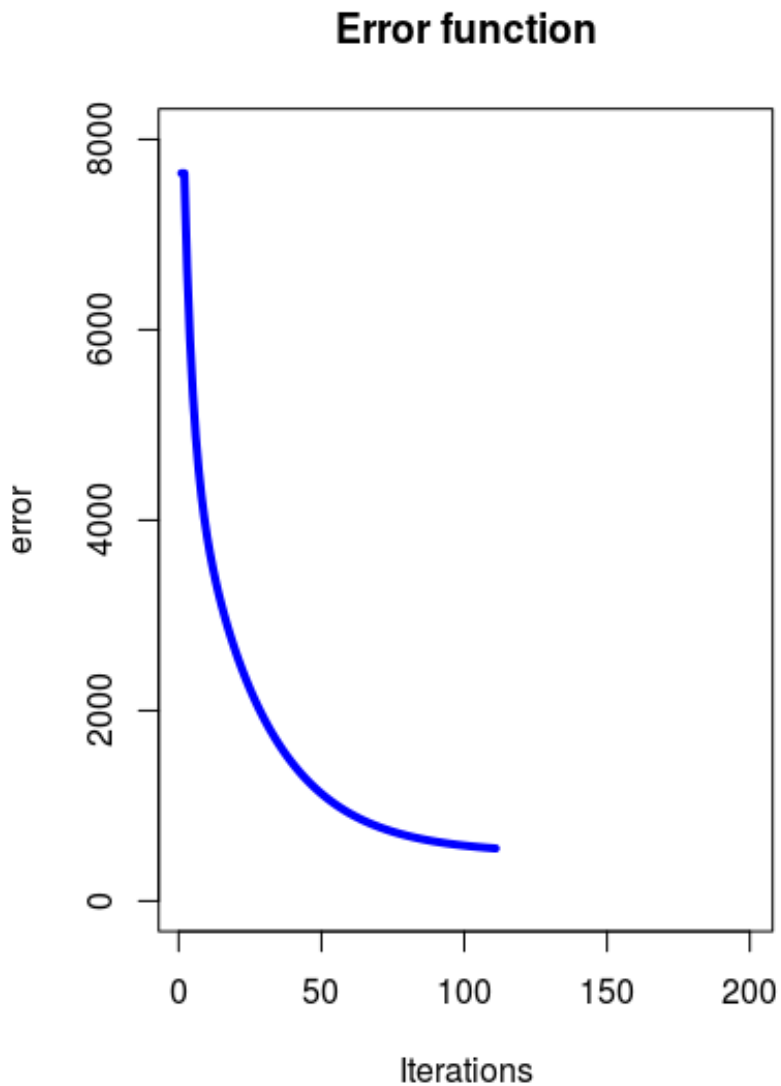


Linear regression - animation

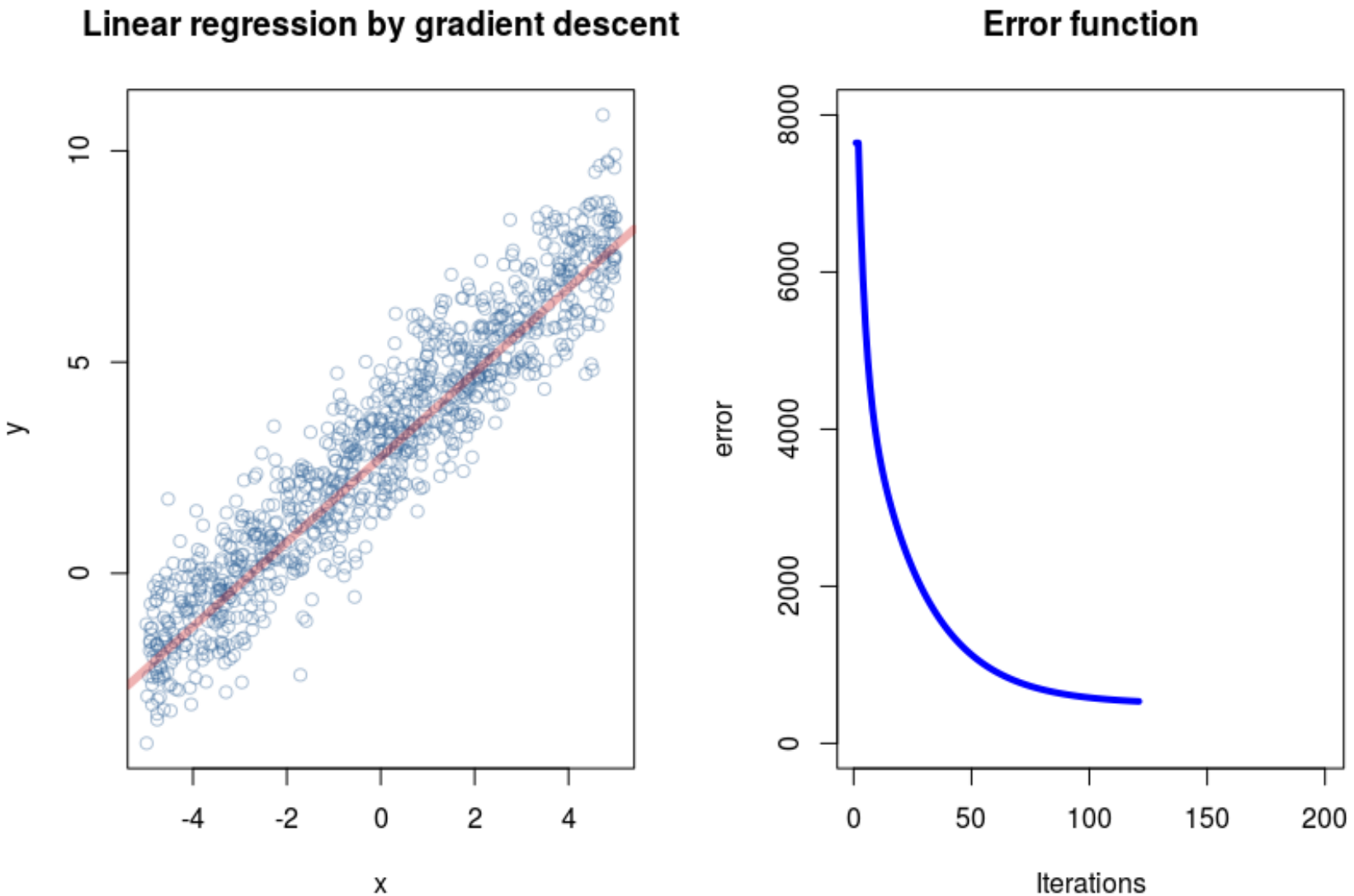


Linear regression - animation

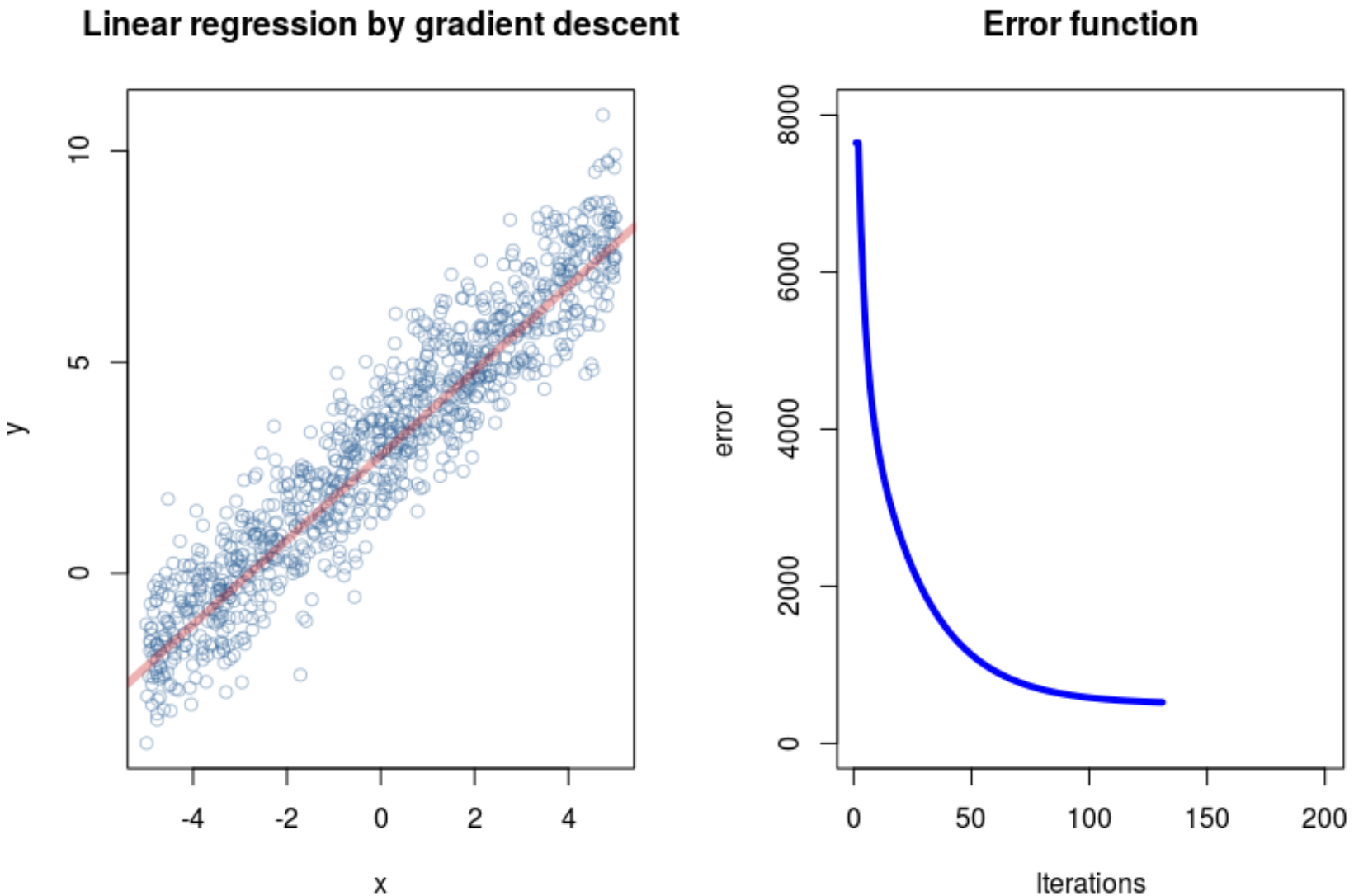




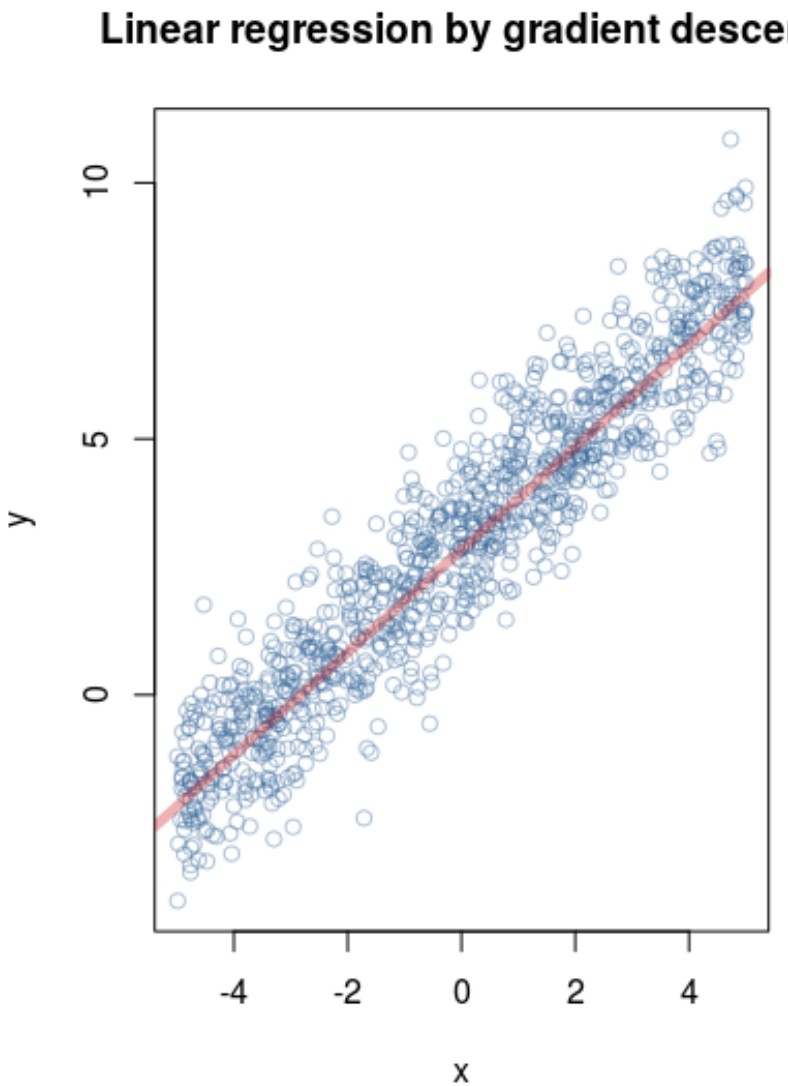
Linear regression - animation

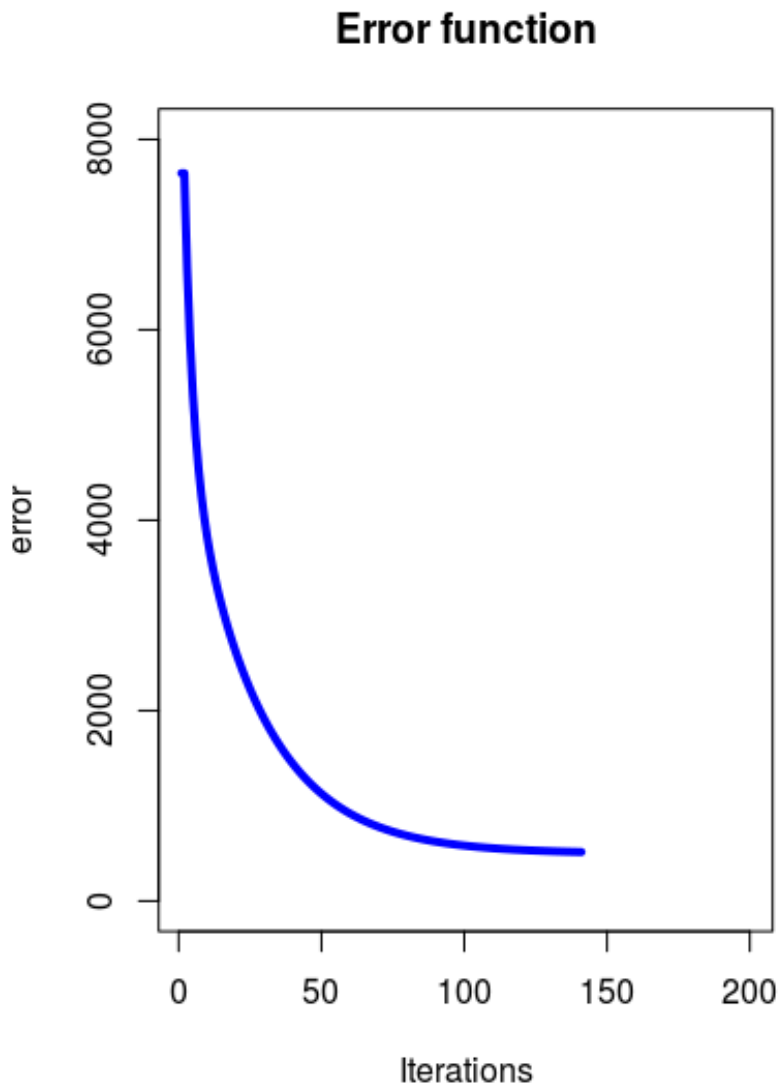


Linear regression - animation

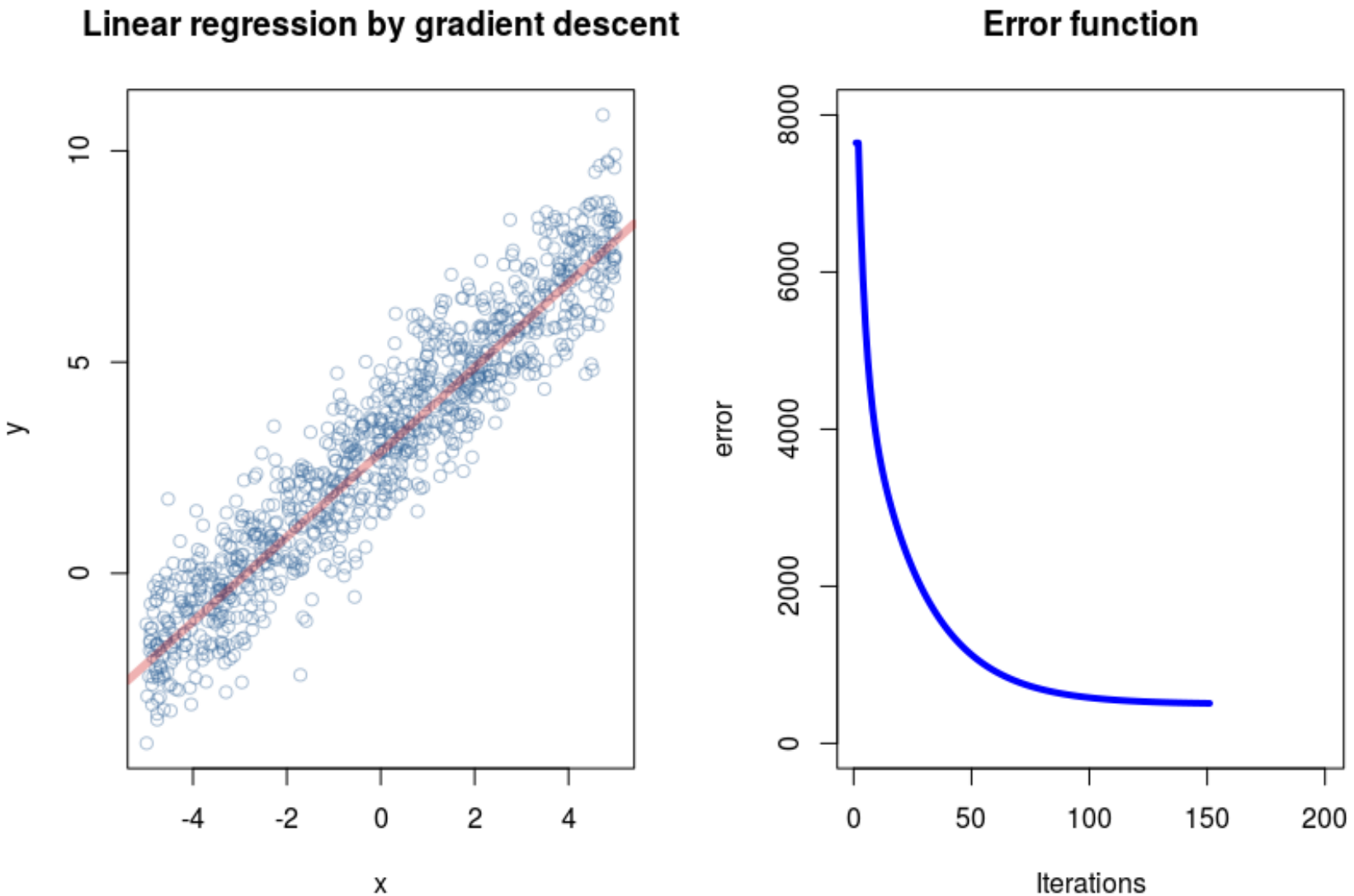


Linear regression - animation

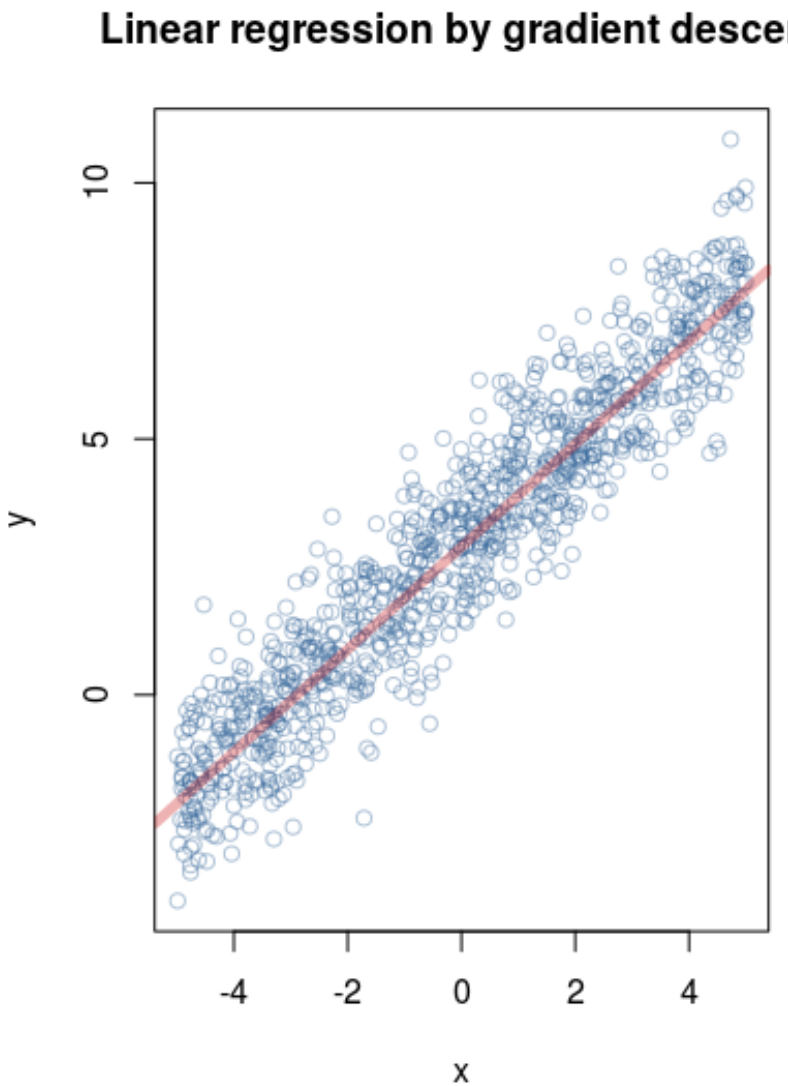


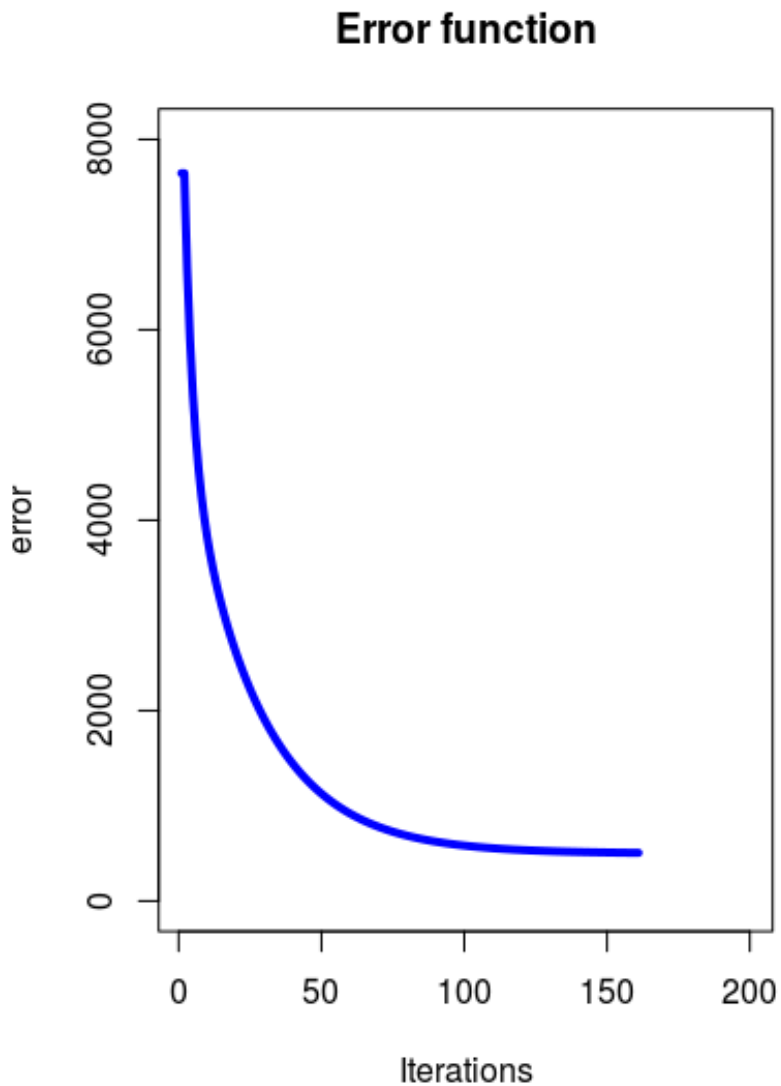


Linear regression - animation

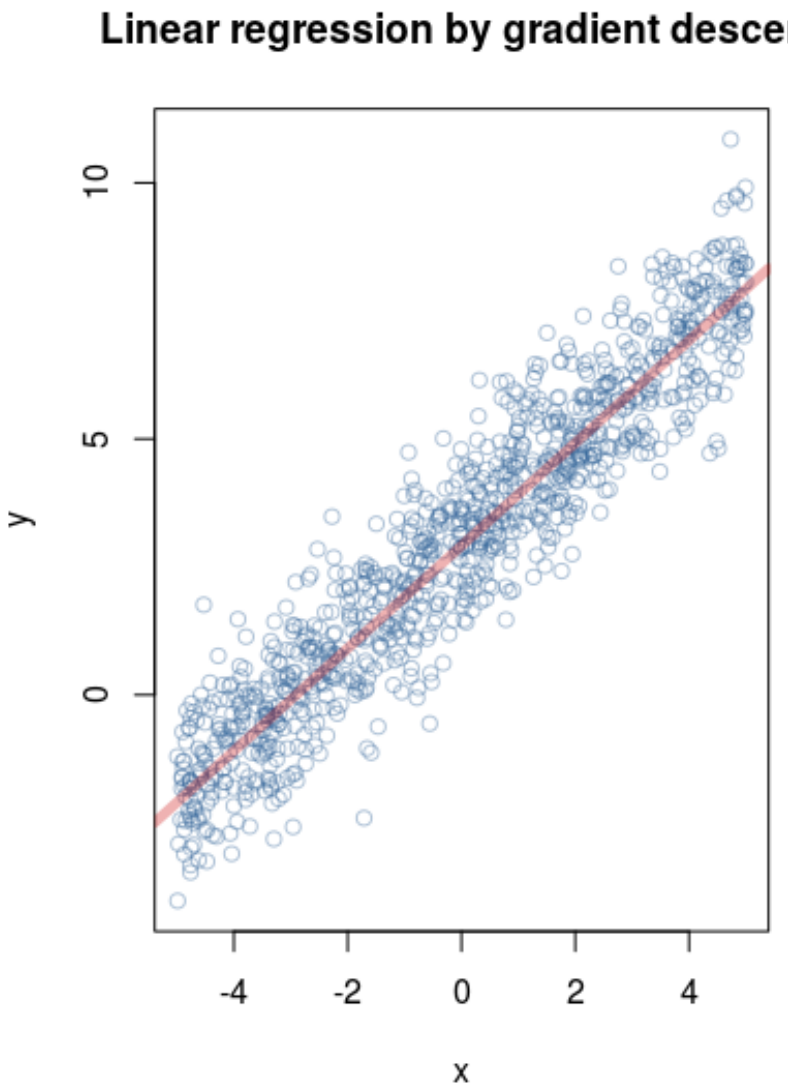


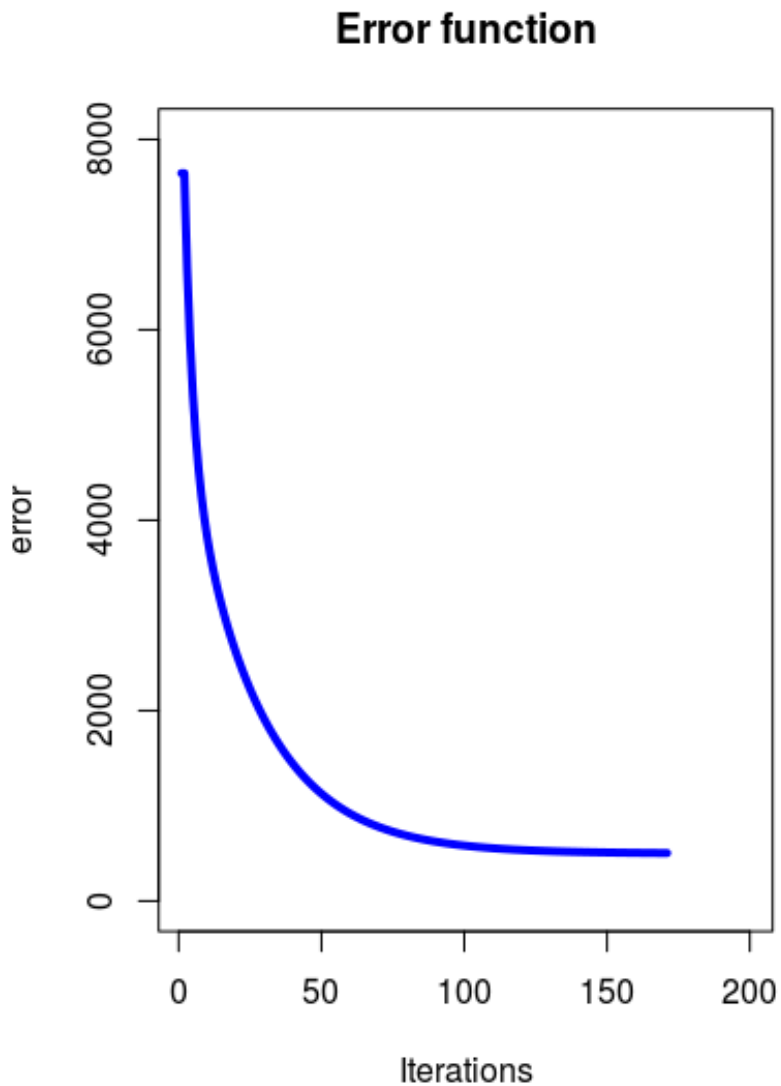
Linear regression - animation



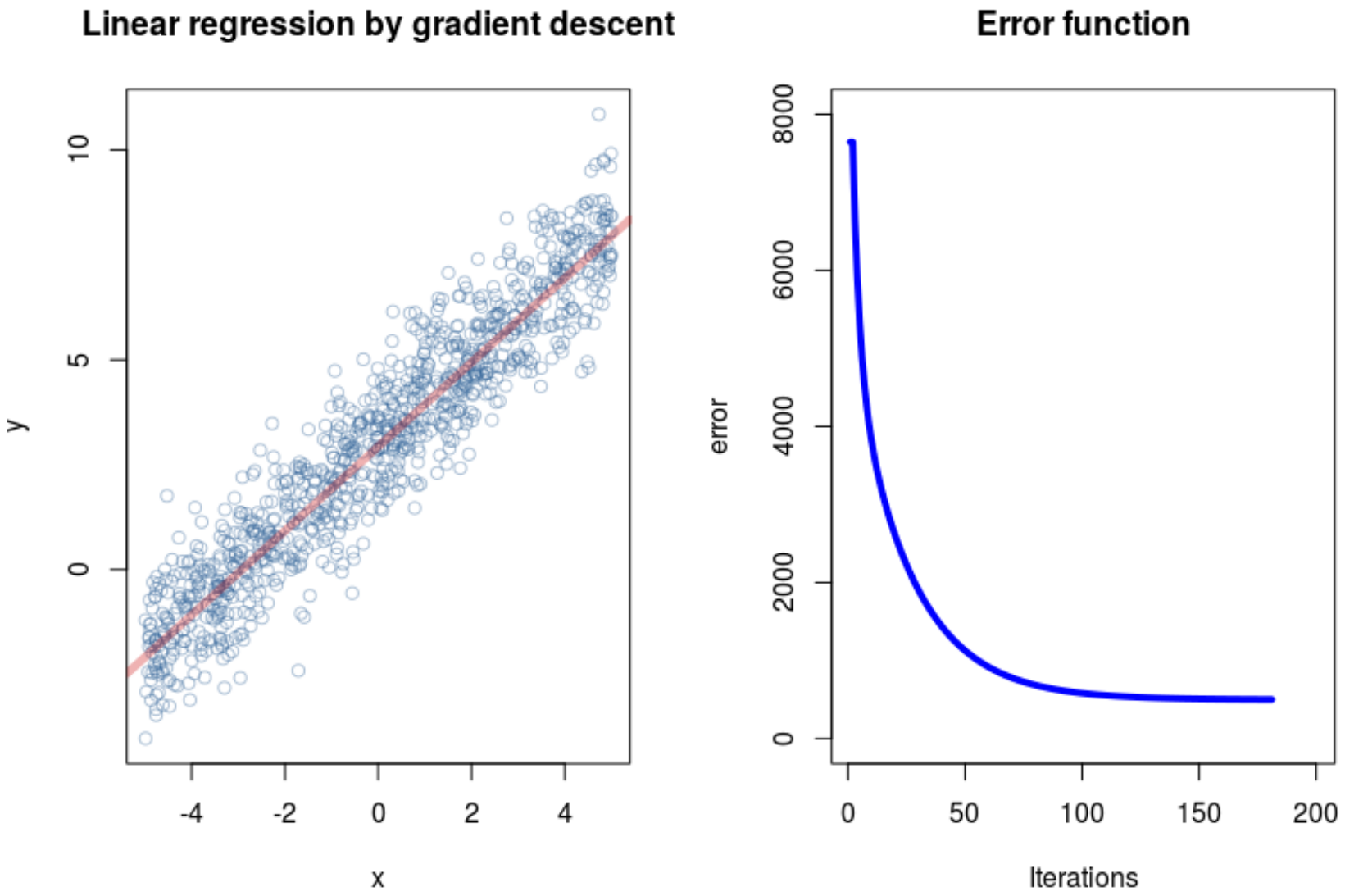


Linear regression - animation

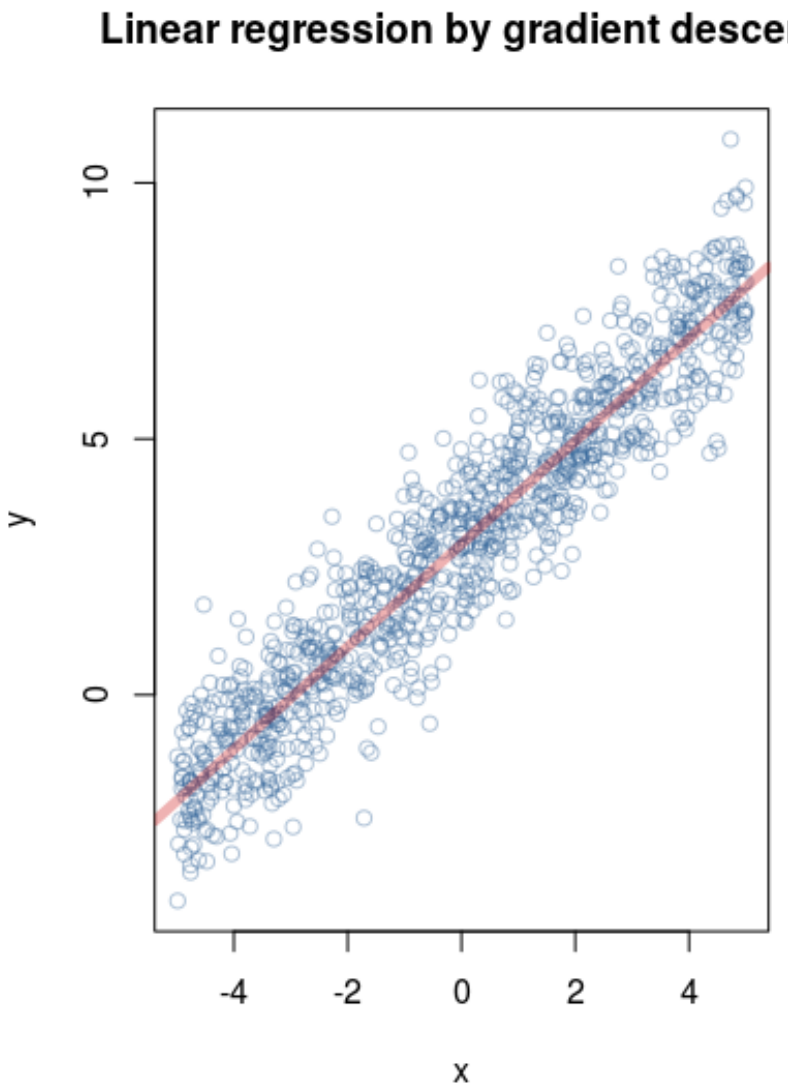


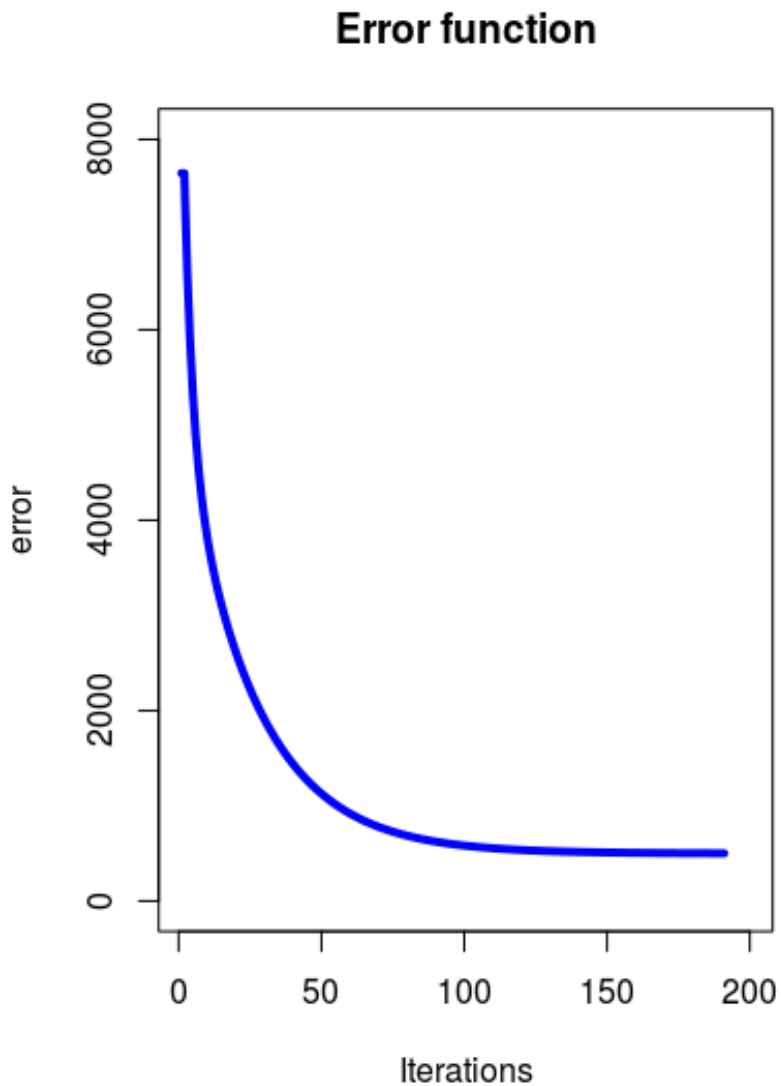


Linear regression - animation

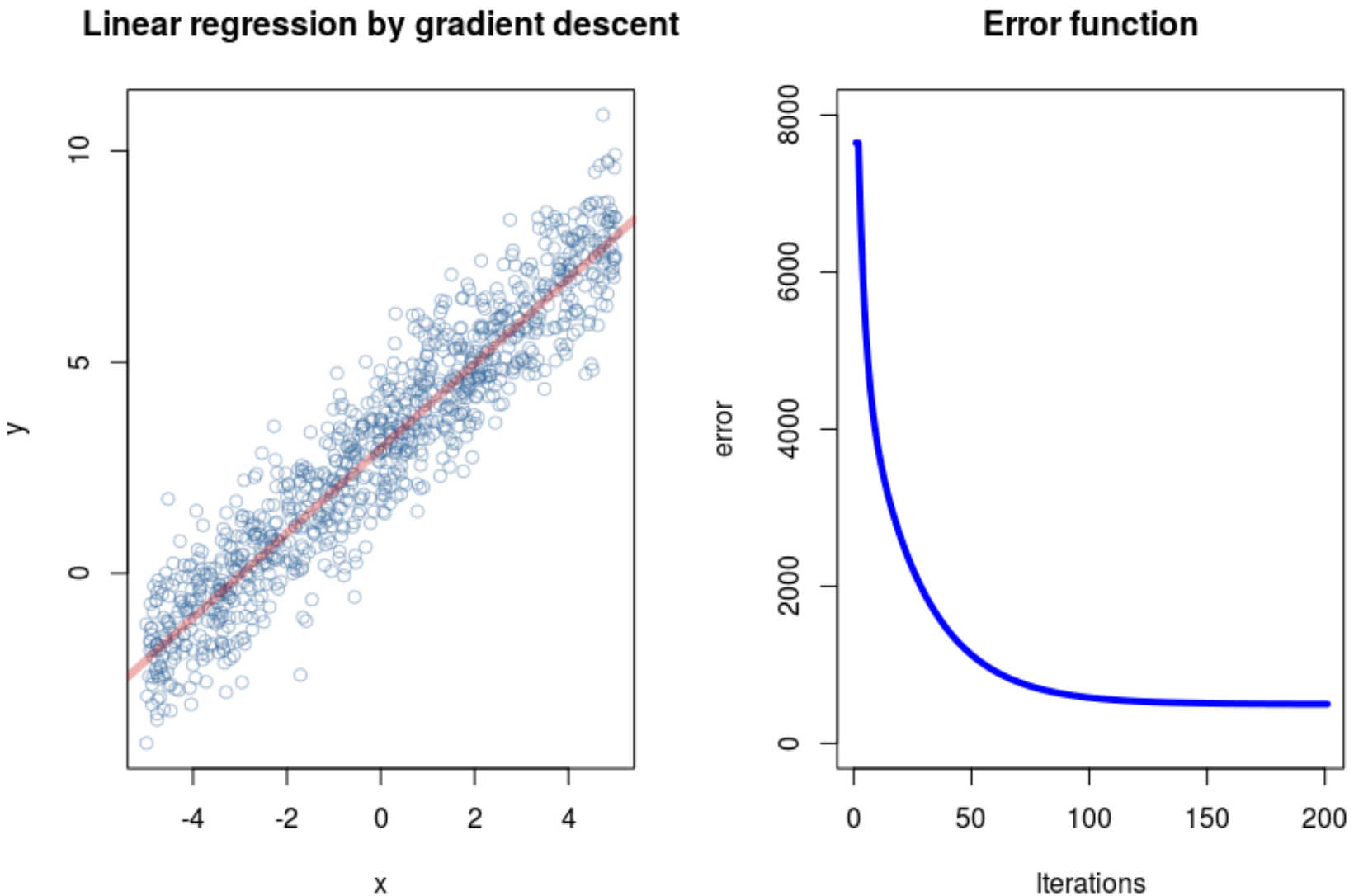


Linear regression - animation





Linear regression - animation



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.

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 $t+1$.

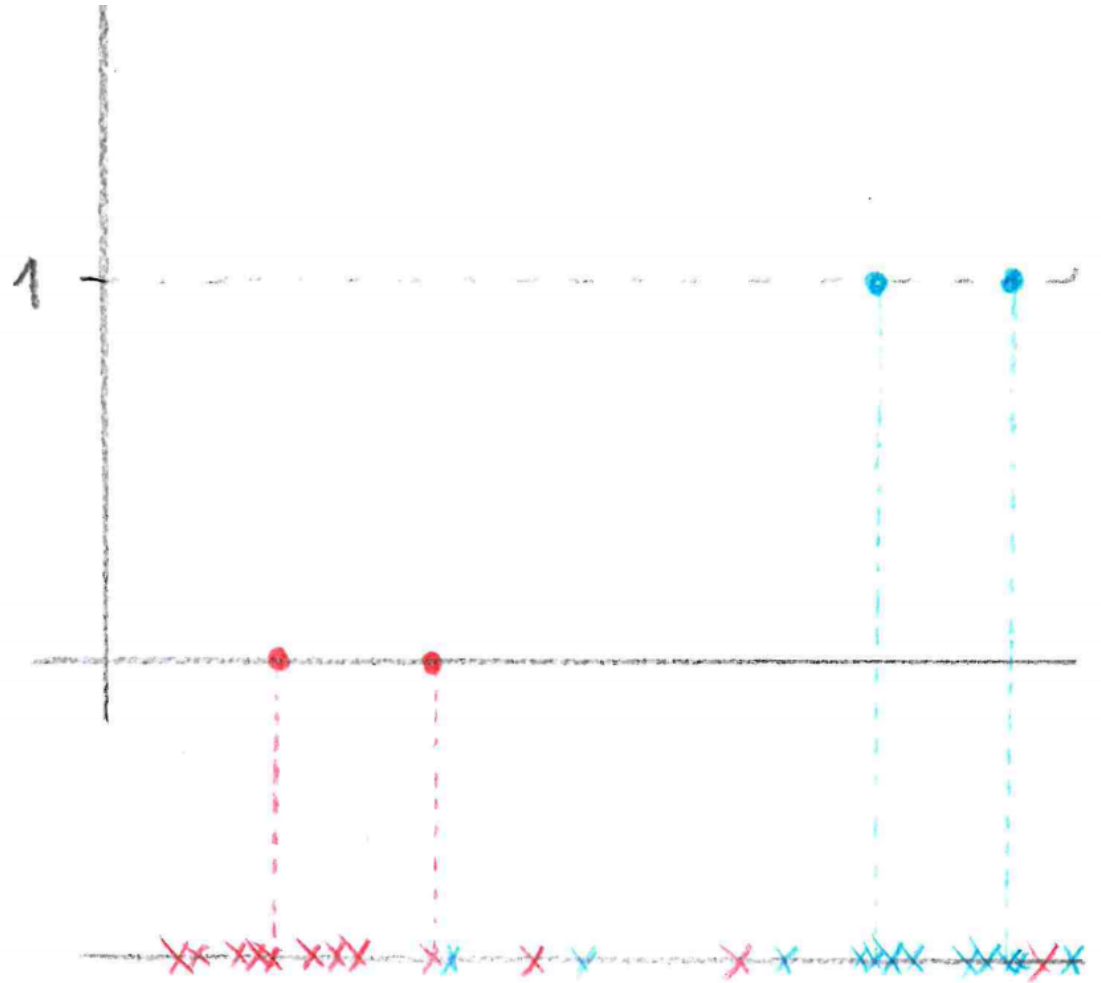
Note that the algorithm does not work with the complete gradient but only with its part determined by the currently considered training example.

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.

What about classification?

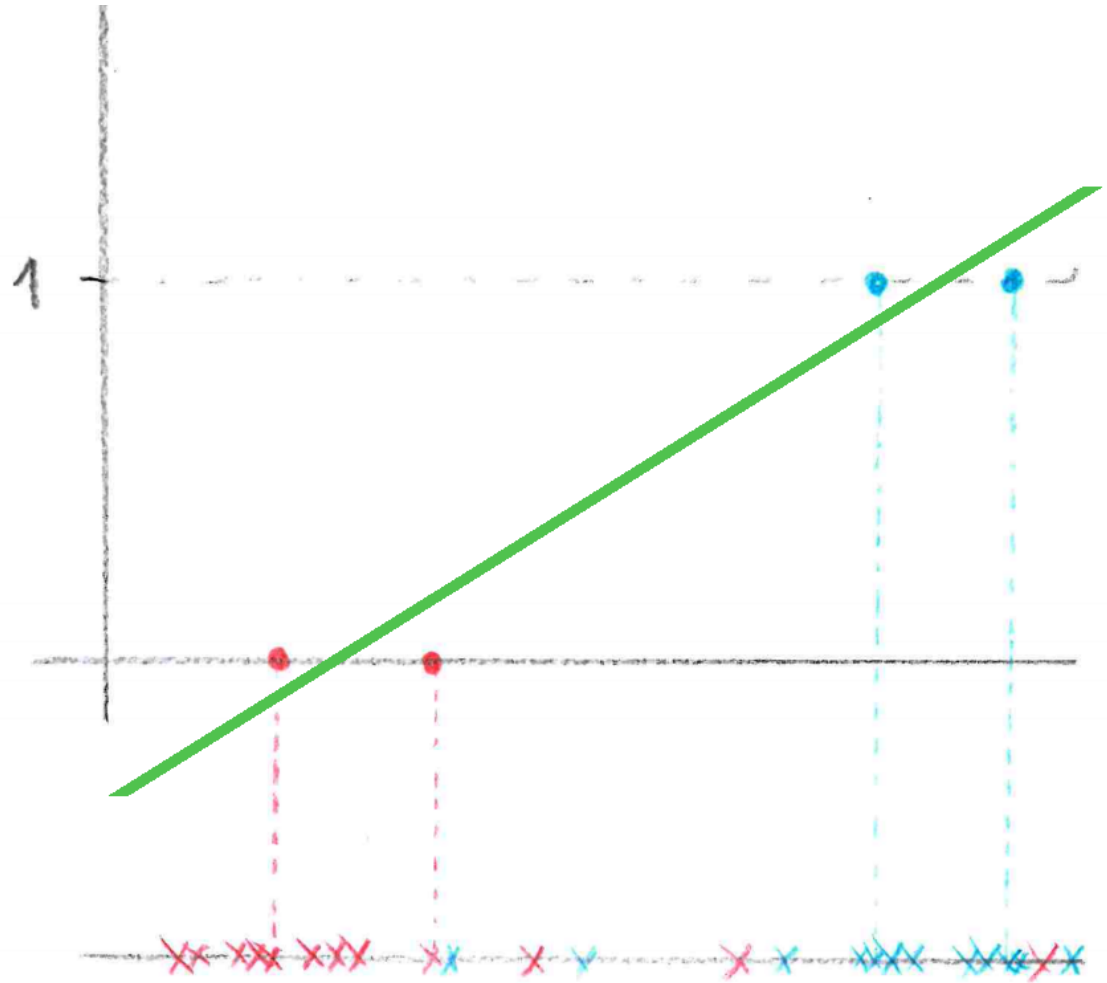
Binary classification: Desired outputs 0 and 1.



Ideally, capture the probability distribution of classes.

What about classification?

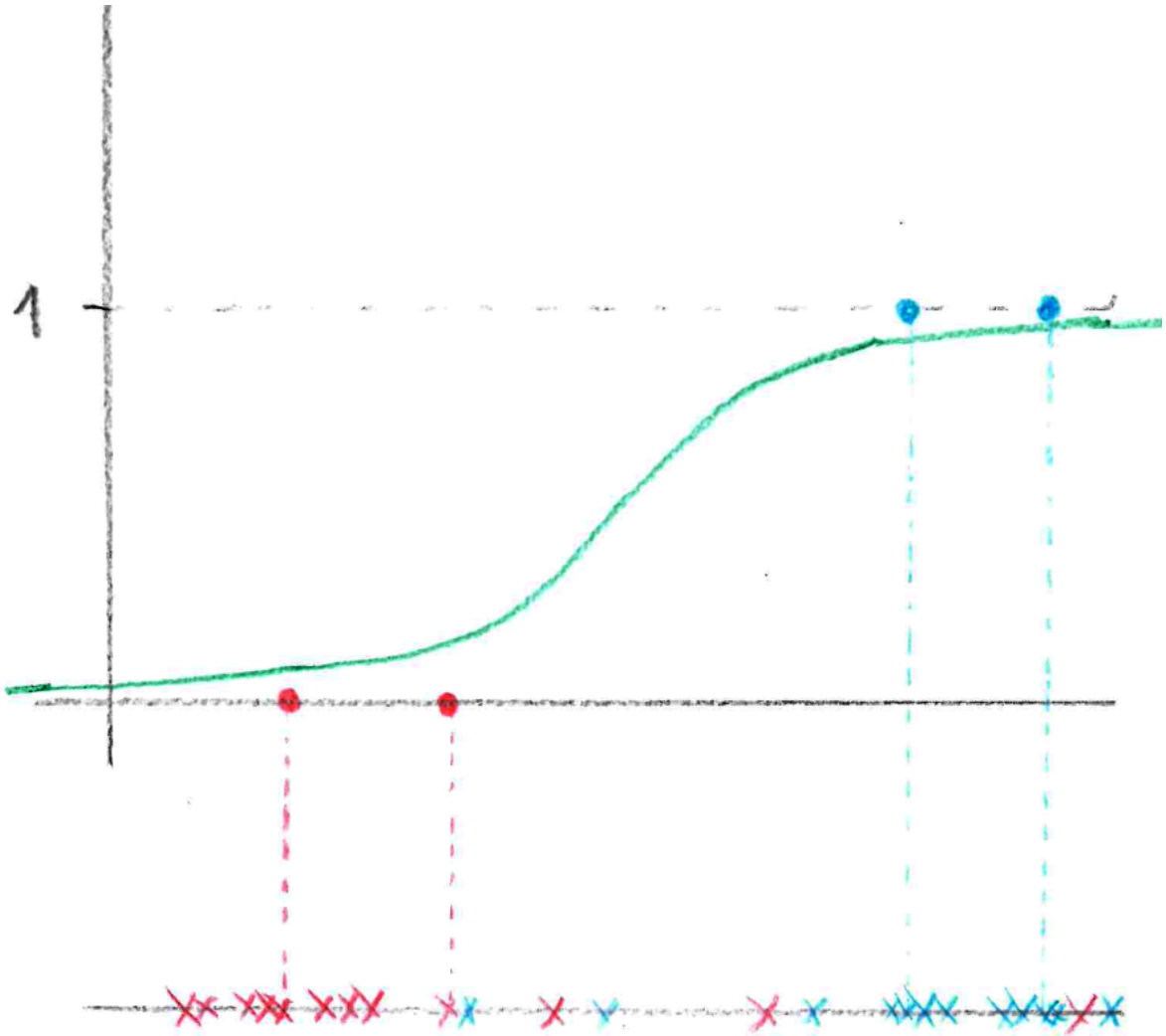
Binary classification: Desired outputs 0 and 1.



... does not capture probability well (it is not a probability at all)

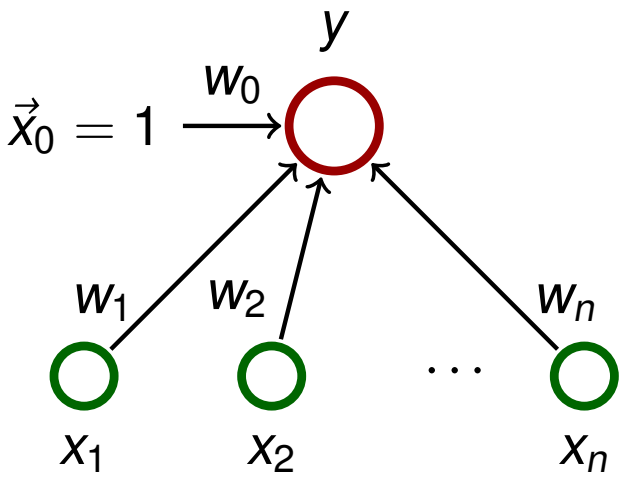
What about classification?

Binary classification: Desired outputs 0 and 1.



... logistic sigmoid $\frac{1}{1+e^{-(\vec{w}\cdot\vec{x})}}$ is much better!

Logistic regression



$\vec{w} = (w_0, w_1, \ldots, w_n)$ and $\vec{x} = (x_0, x_1, \ldots, x_n)$ 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: $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$
- network function: $y[\vec{w}](\vec{x}) = \sigma(\xi) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x})}}$

Intuition: The output y is now the probability of the class 1 given the input $\vec{x}$.

But what is the meaning of the sigmoid?

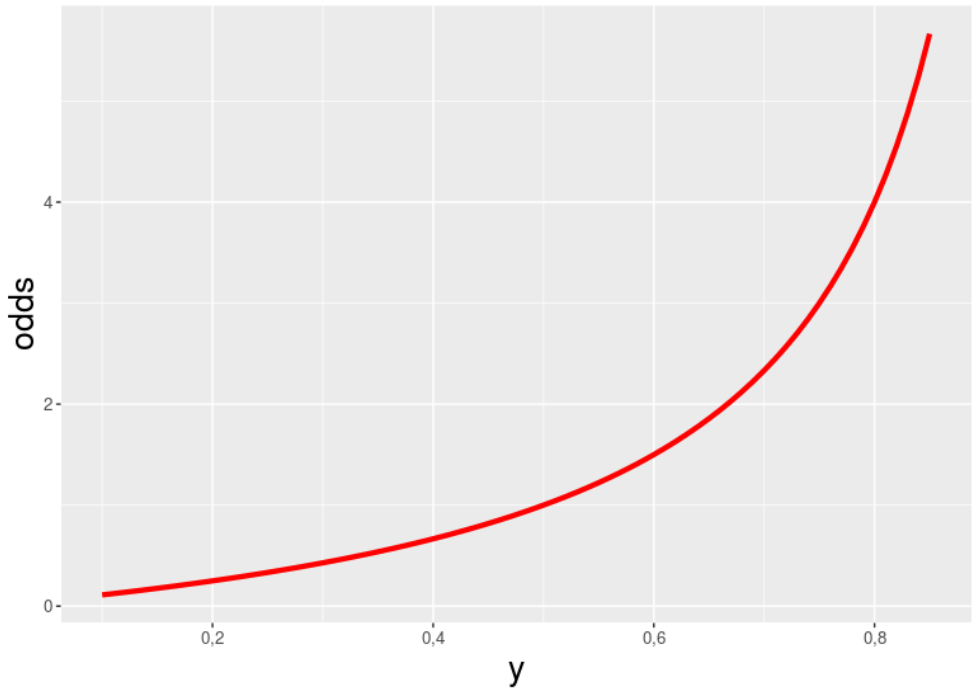
The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1 + e^{-\vec{w}\cdot\vec{x}})$??

But what is the meaning of the sigmoid?

The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1+e^{-\vec{w}\cdot\vec{x}})$??

What about odds of the class 1?

$$odds(y) = \frac{y}{1-y}$$



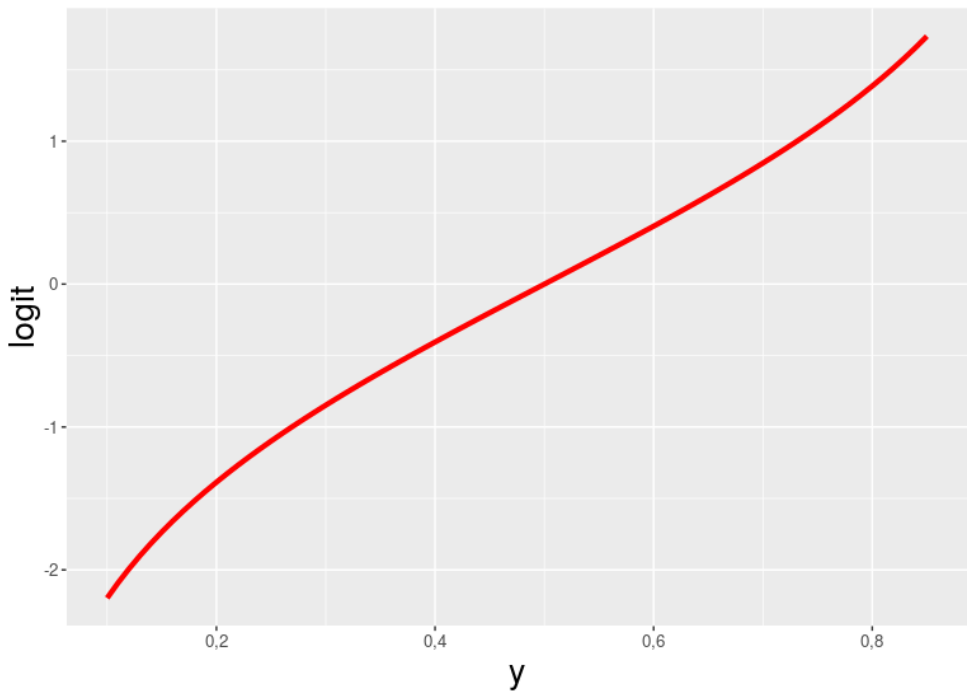
Resembles an exponential function ...

But what is the meaning of the sigmoid?

The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1+e^{-\vec{w}\cdot\vec{x}})$??

What about log odds (aka logit) of the class 1?

$$logit(y) = \log(y/(1-y))$$



Looks almost linear ...

But what is the meaning of the sigmoid?

Put

$$\log(y/(1-y)) = \vec{w} \cdot \vec{x}$$

But what is the meaning of the sigmoid?

Put

$$\log(y/(1-y)) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-y)/y) = -\vec{w} \cdot \vec{x}$$

But what is the meaning of the sigmoid?

Put

$$\log(y/(1-y)) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-y)/y) = -\vec{w} \cdot \vec{x}$$

and

$$(1-y)/y = e^{-\vec{w} \cdot \vec{x}}$$

But what is the meaning of the sigmoid?

Put

$$\log(y/(1-y)) = \vec{w} \cdot \vec{x}$$

Then

$$\log((1-y)/y) = -\vec{w} \cdot \vec{x}$$

and

$$(1-y)/y = e^{-\vec{w} \cdot \vec{x}}$$

and

$$y = \frac{1}{1 + e^{-\vec{w} \cdot \vec{x}}}$$

That is, if we model log odds using a linear function, the probability is obtained by applying the logistic sigmoid on the result of the linear function.

Logistic regression

Learning:

- Given a **training dataset**

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

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 $d_k \in \{0, 1\}$ is the expected output.

Logistic regression

Learning:

- Given a **training dataset**

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

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 $d_k \in \{0, 1\}$ is the expected output.

What error function?

(Binary) cross-entropy:

$$E(\vec{w}) = \sum_{k=1}^{p} -(d_k \log(y_k) + (1 - d_k) \log(1 - y_k))$$

What?!?

Log likelihood is your friend!

- Let’s have a "coin" (sides 0 and 1).

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is p and is unknown!

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is p and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is p and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

- What is the best model y of p based on the data?

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is p and is unknown!
- You have tossed the coin 5 times and got a training dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Consider this to be a very special case where the input dimension is 0

- What is the best model y of p based on the data?

Answer: The one that generates the data with maximum probability!

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Maximize

$$LL = log(L) = \log(y) + \log(y) + \log(1-y) + \log(1-y) + \log(y)$$

Log likelihood is your friend!

Keep in mind our dataset:

$$\mathcal{T} = \{1, 1, 0, 0, 1\} = \{d_1, \ldots, d_5\}$$

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ is

$$L = y \cdot y \cdot (1 - y) \cdot (1 - y) \cdot y$$

How to maximize this w.r.t. y?

Maximize

$$LL = log(L) = \log(y) + \log(y) + \log(1-y) + \log(1-y) + \log(y)$$

But then

$$-LL = -1 \cdot log(y) - 1 \cdot log(y) - (1-0) \cdot log(1-y) - (1-0) \cdot \log(1-y) - 1 \cdot \log(y)$$

and thus $-LL$ is the cross-entropy.

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}}$$

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}}$$

The training dataset is now standard:

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

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 $d_k \in \{0, 1\}$ is the expected output.

Let the coin depend on the input

Consider our model:

$$y = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}}$$

The training dataset is now standard:

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

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 $d_k \in \{0, 1\}$ is the expected output.

The likelihood:

$$L = \prod_{k=1}^{p} y_k^{d_k} \cdot (1 - y_k)^{(1-d_k)}$$

and $LL = \log(L) = \sum_{k=1}^{p} (d_k \log(y_k) + (1 - d_k) \log(1 - y_k))$
and thus $-LL$ = the cross-entropy.

Minimizing the cross-netropy maximizes the log-likelihood (and vice versa).

Normal Distribution

Distribution of continuous random variables.

Density (one dimensional, that is over $\mathbb{R}$):

$$p(x) \quad = \quad \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \quad =: \quad N[\mu, \sigma^2](x)$$

μ is the expected value (the mean), σ^2 is the variance.

Maximum Likelihood vs Least Squares (Dim 1)

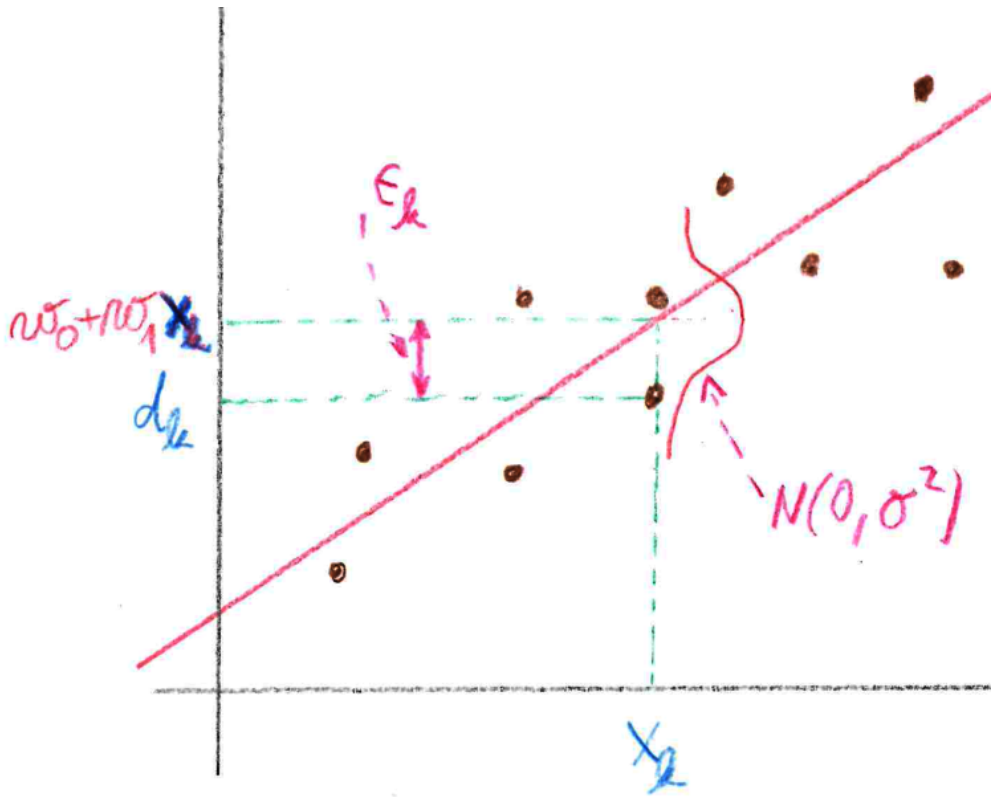
Fix a training set $D = \left\{ (x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right) \right\}$

Maximum Likelihood vs Least Squares (Dim 1)

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$
Assume that each d_k has been generated randomly by

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

- w_0, w_1 are **unknown numbers**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2



Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Denote by $p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$ the probability density according to which the values $d_1, \ldots, d_n$ were generated assuming fixed $w_0, w_1, \sigma^2, x_1, \ldots, x_p$.

Maximum Likelihood vs Least Squares (Dim 1)

Keep in mind:

$$d_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

Assume that $\epsilon_1, \ldots, \epsilon_p$ were generated **independently**.

Denote by $p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$ the probability density according to which the values $d_1, \ldots, d_n$ were generated assuming fixed $w_0, w_1, \sigma^2, x_1, \ldots, x_p$.

The independence and normality imply

$$\begin{aligned} p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2) &= \prod_{k=1}^{p} N[w_0 + w_1 x_k, \sigma^2](d_k) \\ &= \prod_{k=1}^{p} \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(d_k - w_0 - w_1 x_k)^2}{2\sigma^2}\right\} \end{aligned}$$

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes $E(w_0, w_1)$, i.e. the least squares error function.*

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes $E(w_0, w_1)$, i.e. the least squares error function.*

Note that the maximizing/minimizing (w_0, w_1) does not depend on σ^2.

Maximum Likelihood vs Least Squares

Our goal is to find (w_0, w_1) that maximizes the likelihood that the training set D with **fixed** values $d_1, \ldots, d_n$ has been generated:

$$L(w_0, w_1, \sigma^2) := p(d_1, \ldots, d_p \mid w_0, w_1, \sigma^2)$$

Theorem

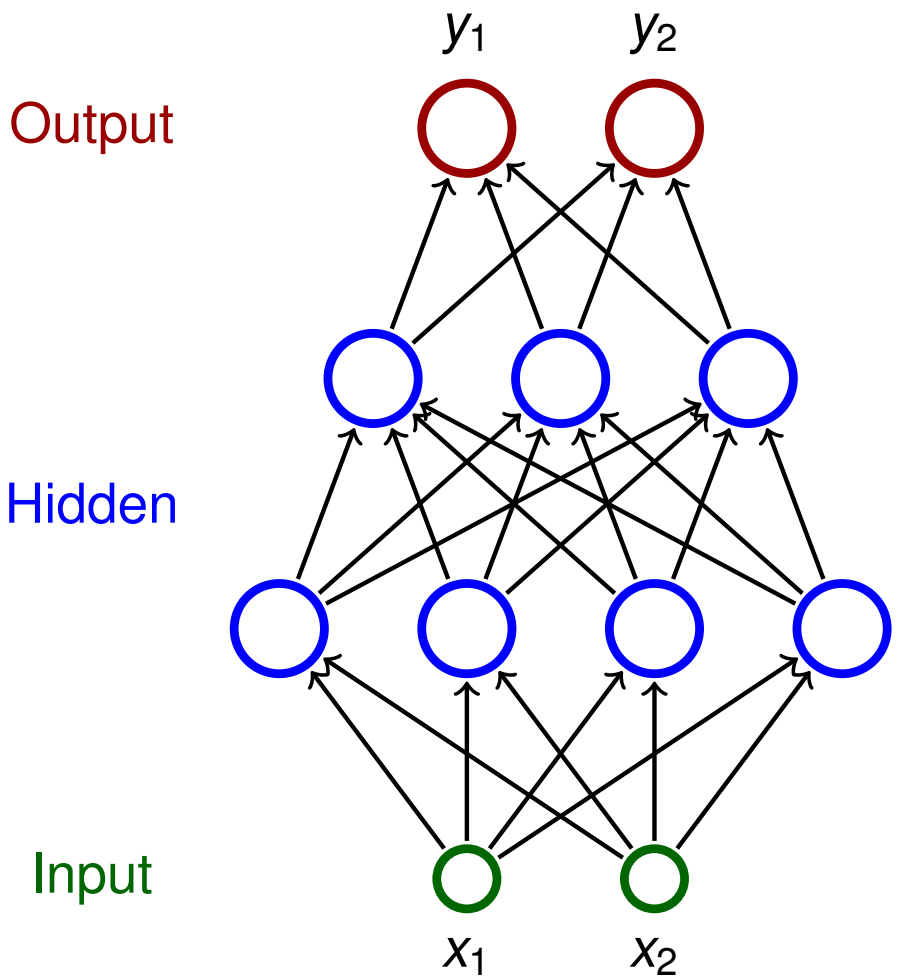
(w_0, w_1) maximizes $L(w_0, w_1, \sigma^2)$ for arbitrary σ^2 **iff** *(w_0, w_1) minimizes $E(w_0, w_1)$, i.e. the least squares error function.*

Note that the maximizing/minimizing (w_0, w_1) does not depend on σ^2.

Maximizing σ^2 satisfies $\sigma^2 = \frac{1}{p} \sum_{k=1}^{p} (d_k - w_0 - w_1 \cdot x_k)^2$.

MLP training – theory

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 ($X, Y \subseteq Z$)

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*

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)

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)

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)

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:

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

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 $\sigma_j(\xi) = \frac{1}{1+e^{-\lambda_j \xi}}$]

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 $y_j(\vec{w}, \vec{x})$)

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 $\left(d_{kj}\right)_{j \in Y}$).

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:

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

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 $t+1$.

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

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

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

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

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 $j \in Z \setminus X$ we get

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

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 $j \in Z \setminus X$ 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 $y_j(\vec{w}, \vec{x}_k)$).

MLP – error function gradient

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

$$\sigma_j'(\xi_j) = \lambda_j y_j (1 - y_j)$$

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 $j \in Z \setminus X$:

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

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

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 $j \in Z \setminus X$:

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

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

- 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:

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: $\mathcal{E}_{ji} = \frac{\partial E}{\partial w_{ji}}$)

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:

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 $j \in Z$

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!)

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

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

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:

MLP – backpropagation

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

- if $j \in Y$, then $\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$

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)

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:

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 $y_j(\vec{w}, \vec{x}_k)$

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 $\frac{\partial E_k}{\partial y_j}$

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)

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.

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)

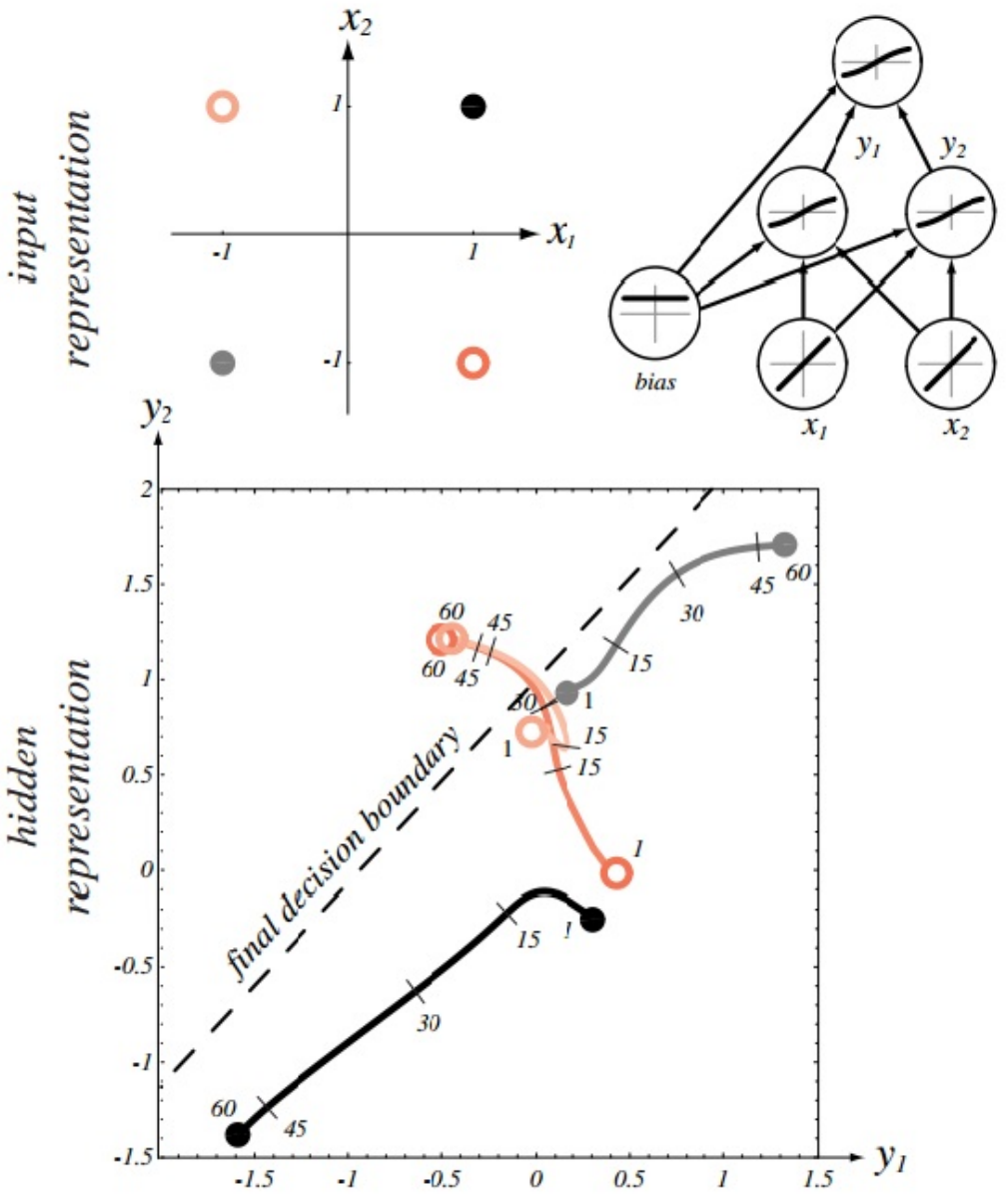
Proof sketch: The algorithm does the following p times:

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 ...

Illustration of the gradient descent – XOR



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

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).

SGD

- 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

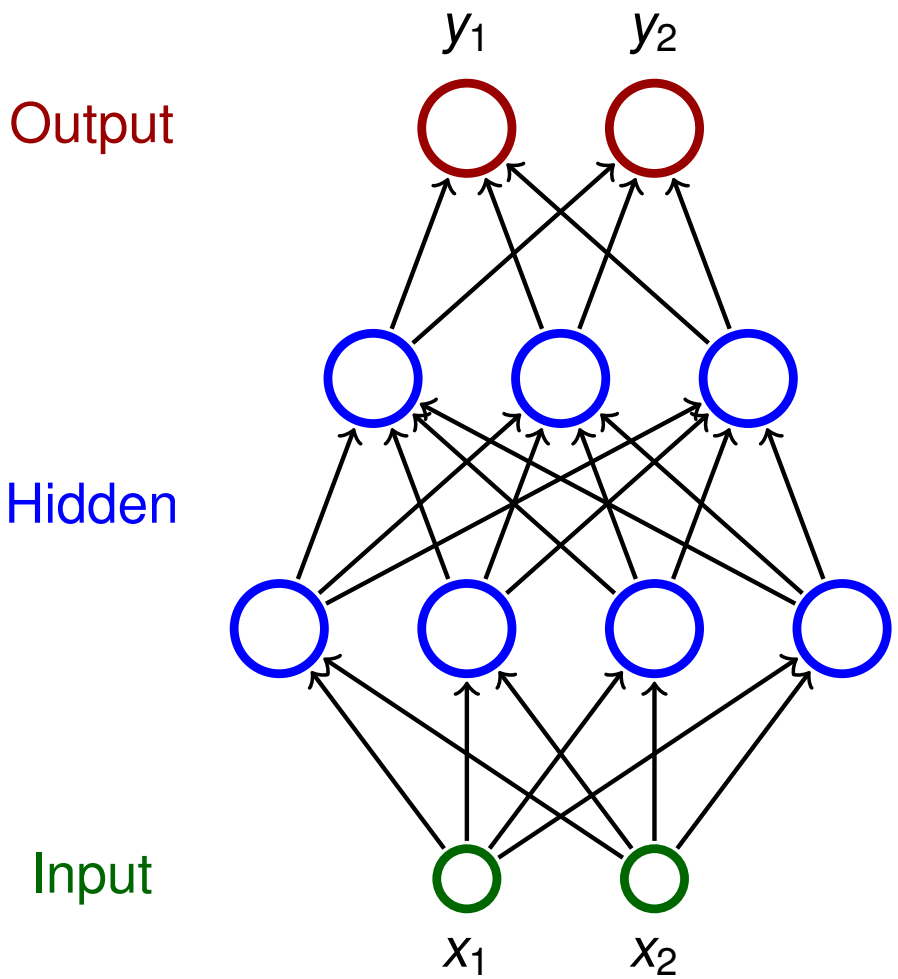
$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

- $0 < \varepsilon(t) \leq 1$ 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.

MLP training – practical issues

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:** (for example)

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

where

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

SGD

- 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

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

- $0 < \varepsilon(t) \leq 1$ 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.

MLP – mse gradient

For every w_{ji} we have

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

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 (for squared error)

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

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

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

(Some) error functions

- **squared error:**

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

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

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

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

- **(categorical) cross entropy:**

$$E(\vec{w}) = -\frac{1}{p} \sum_{k=1}^{p} \sum_{j \in Y} d_{kj} \ln(y_j)$$

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?

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.

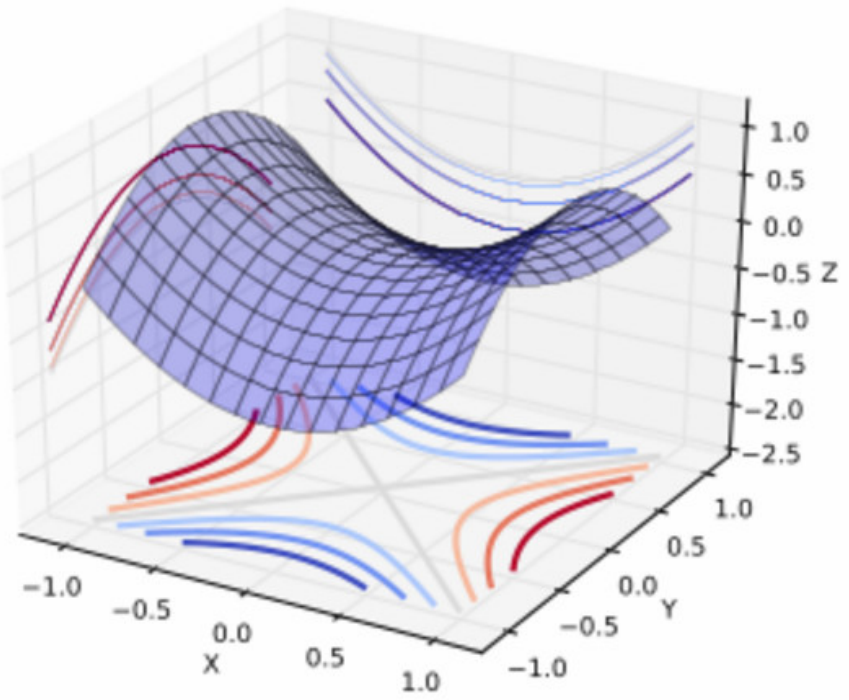
Issues in gradient descent

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

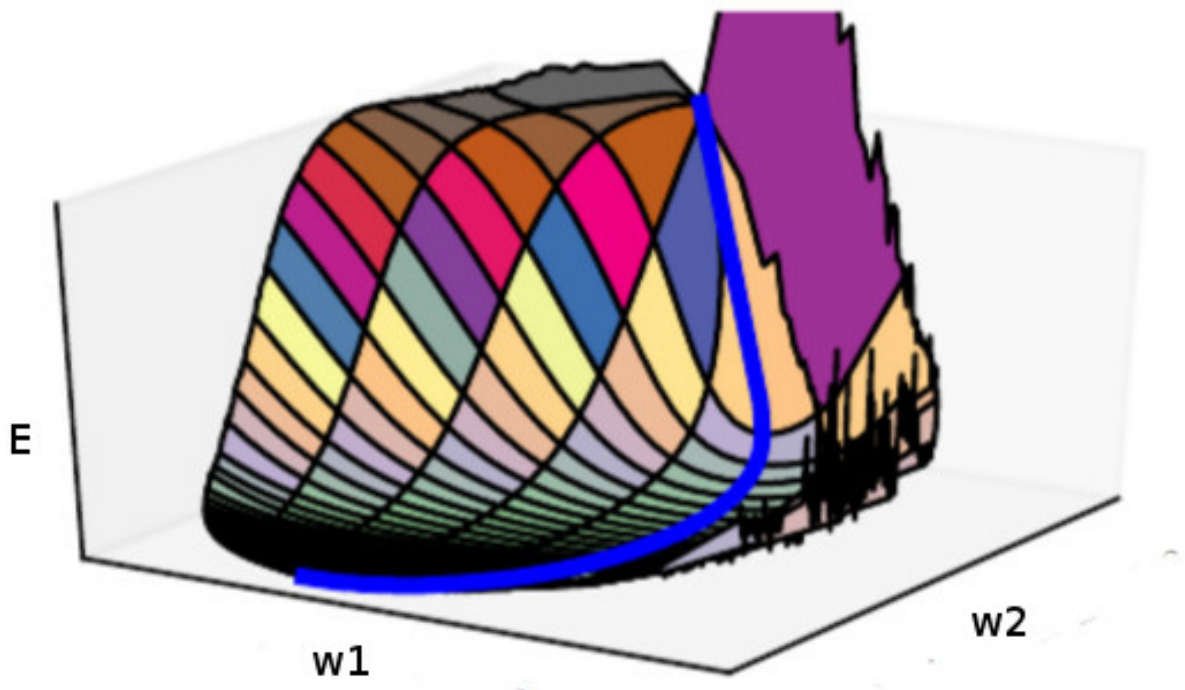
One can show (by a combinatorial argument) that larger networks have exponentially more saddle points than local minima.



Issues in gradient descent – too slow descent

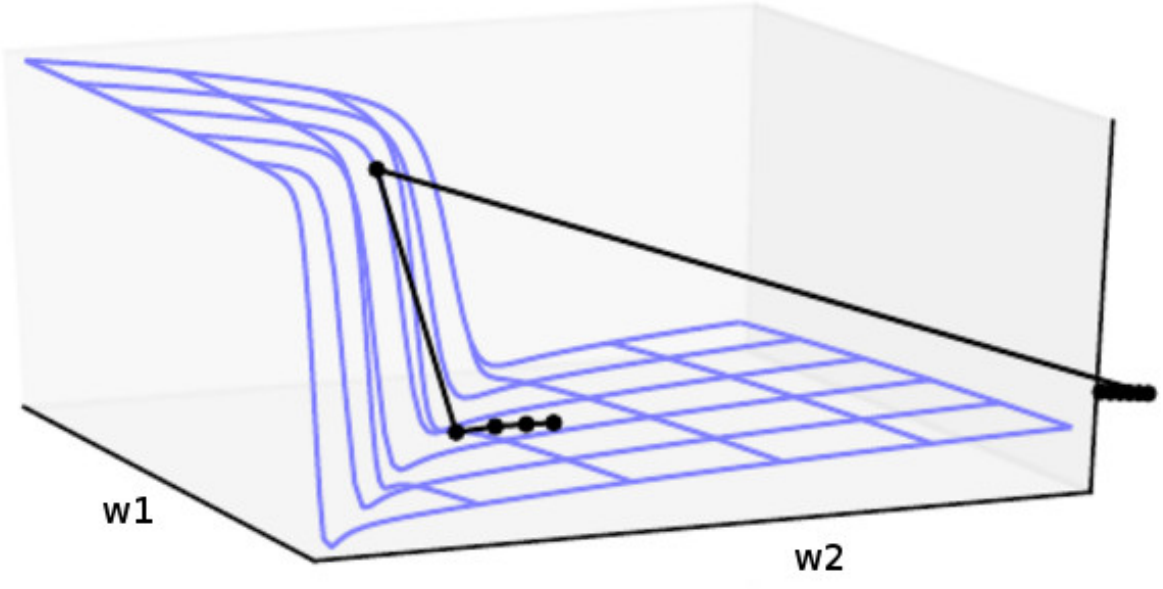
- flat regions

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

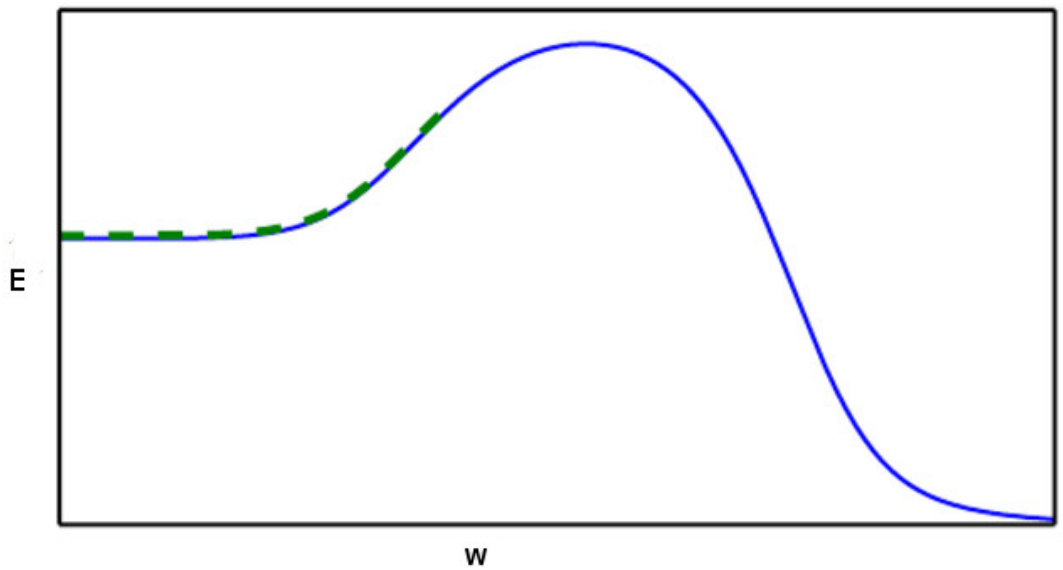


Issues in gradient descent – too fast descent

- 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

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

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

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.

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.

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.

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.
- It is common (especially when using GPUs) 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.

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.
- It is common (especially when using GPUs) 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.

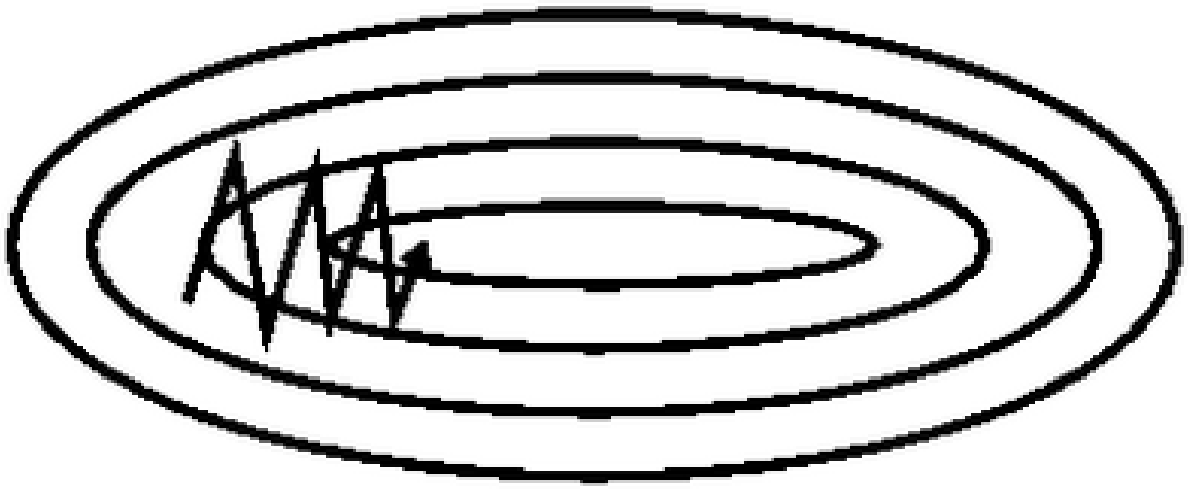
  It has been observed in practice that when using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize.

("On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". Keskar et al, ICLR'17)

Moment

Issue in the gradient descent:

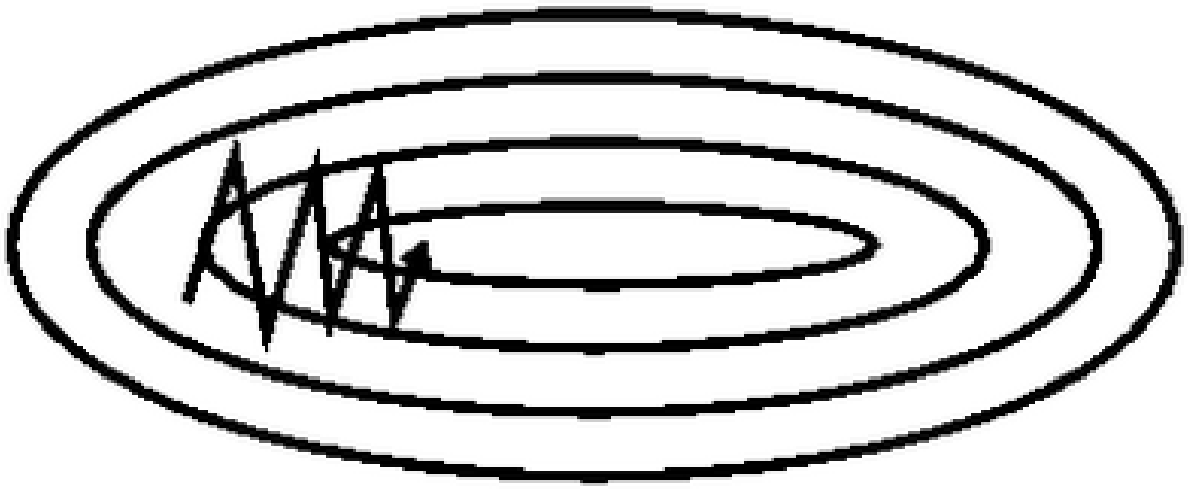
- $\nabla E(\vec{w}^{(t)})$ constantly changes direction (but the error steadily decreases).



Moment

Issue in the gradient descent:

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

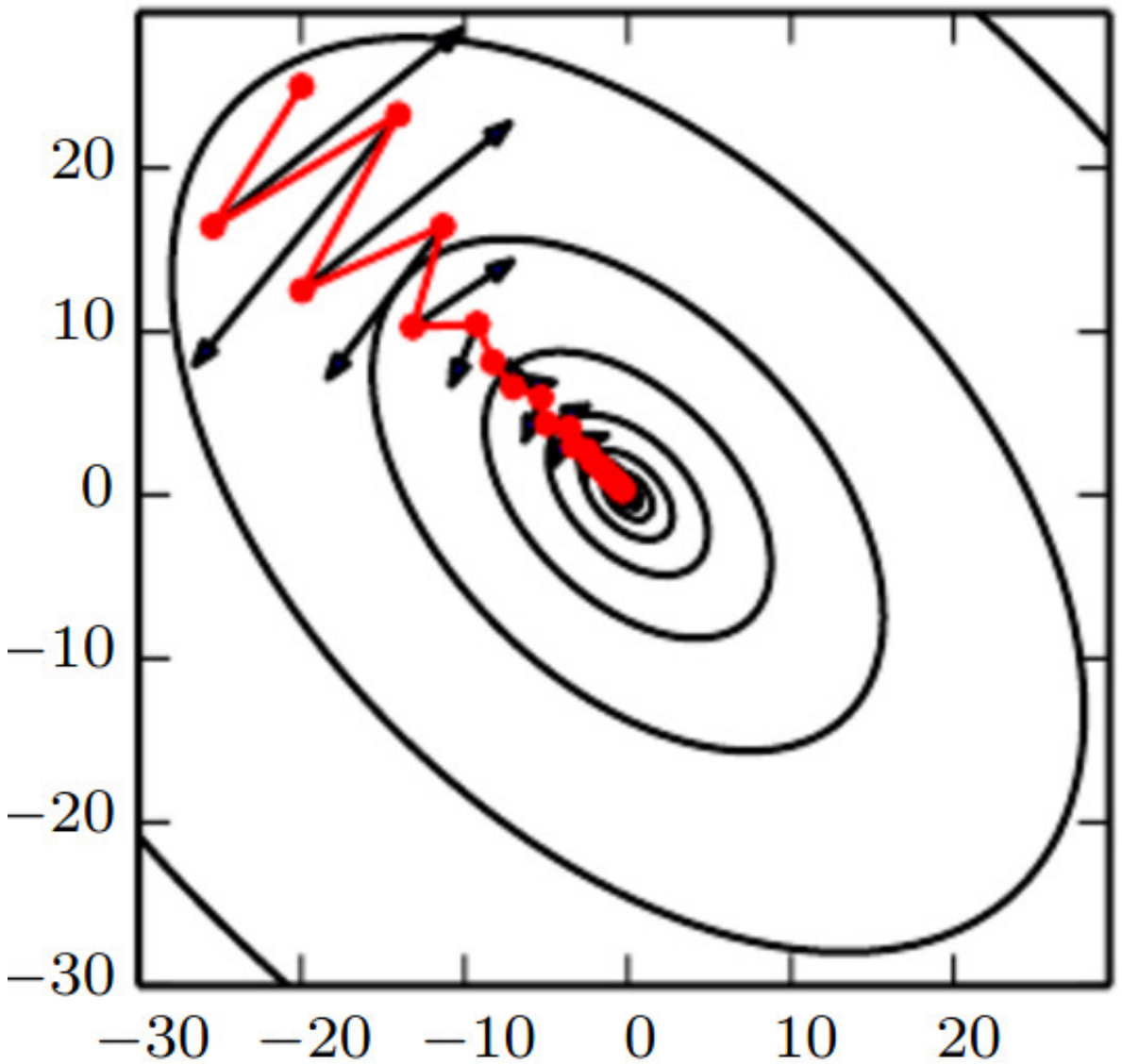


Solution: In every step add the change made in the previous step (weighted by a factor α):

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

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

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

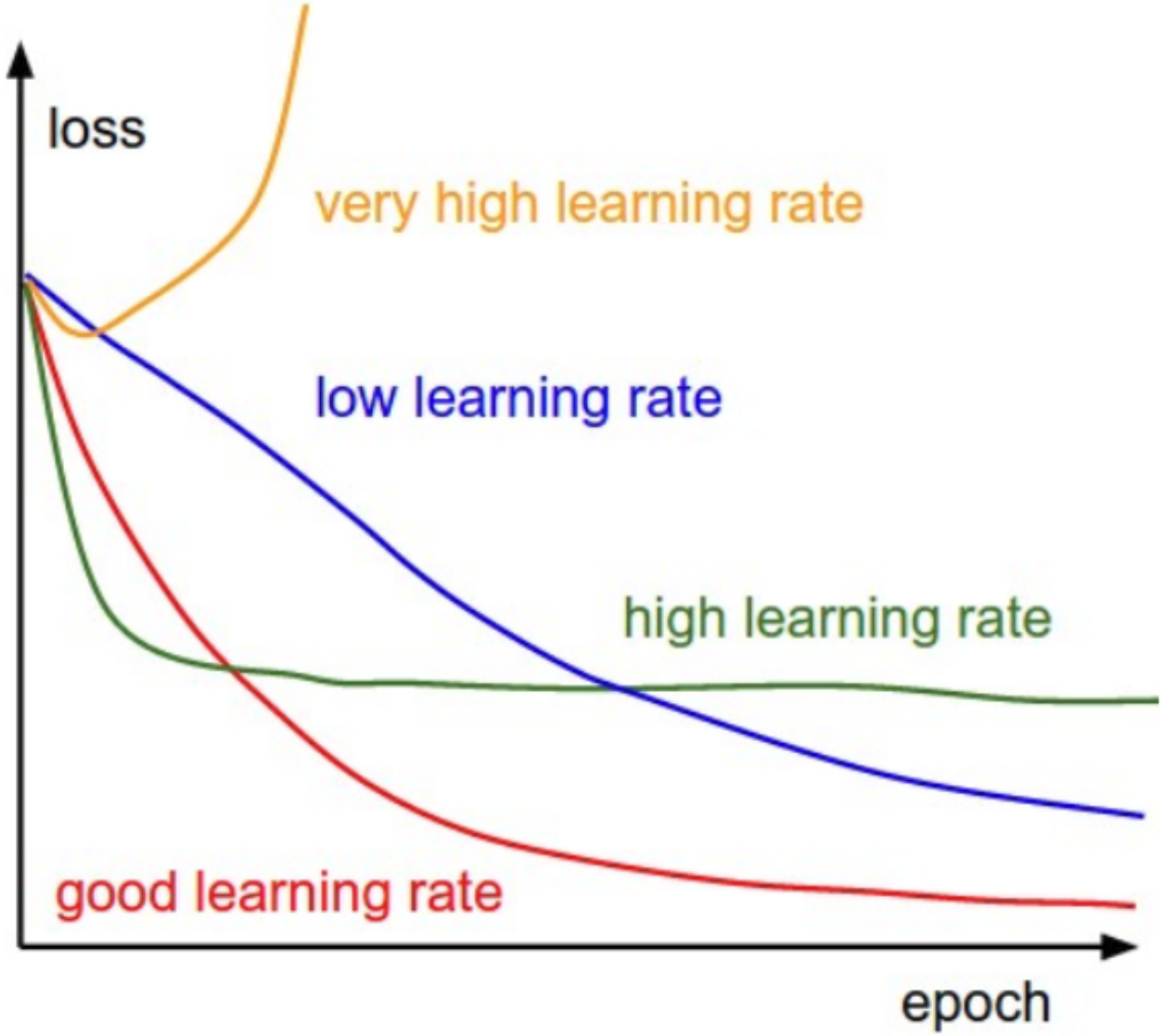
where

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

- $0 < \varepsilon(t) \leq 1$ 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



Learning rate

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

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.

Learning rate

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

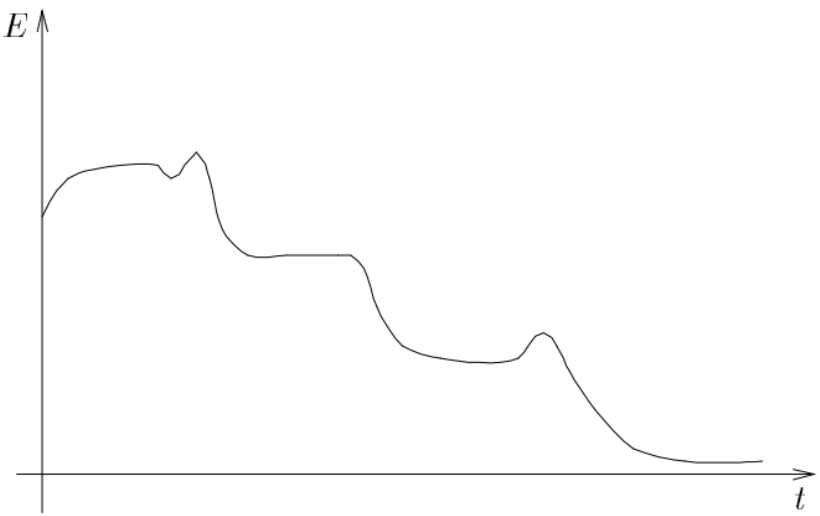
- Start with a larger learning rate (e.g. $\varepsilon = 0.1$).

  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

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

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

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

and

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

- 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

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

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

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

and

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

- 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?

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.

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

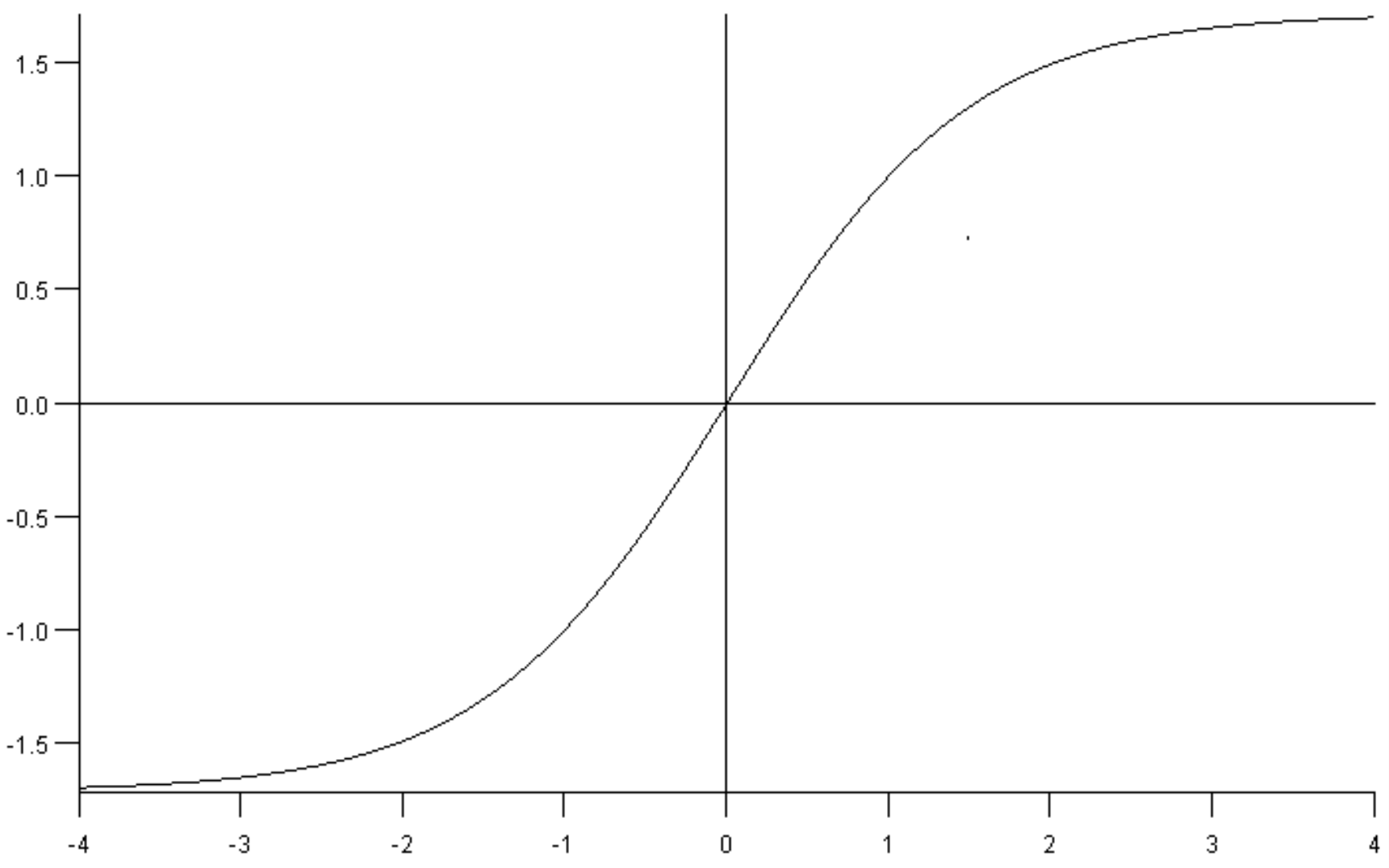
   (local extrema of activation functions induce local extrema of the error function)

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)

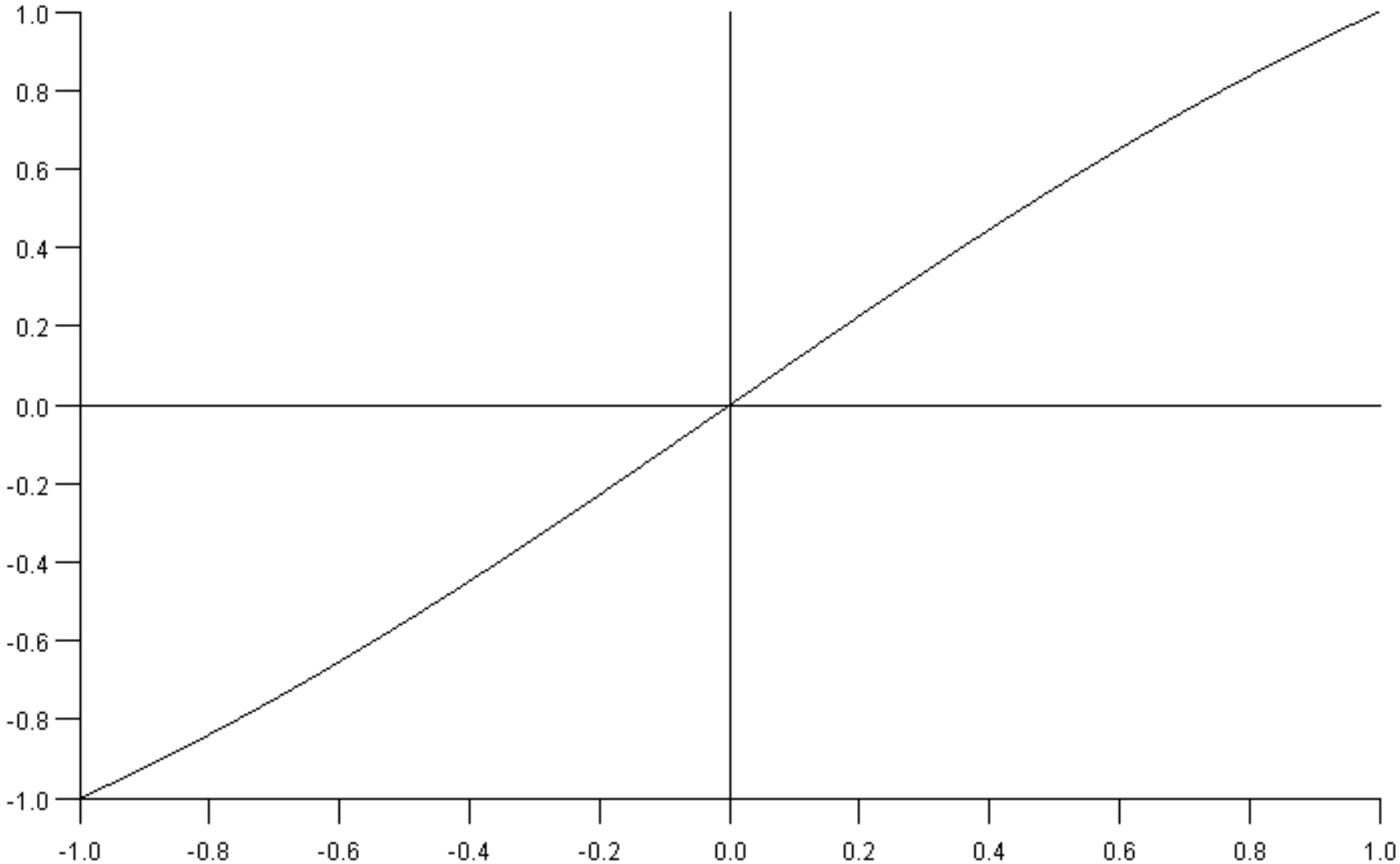
The choice of activation functions is closely related to input preprocessing and the initial choice of weights. I will illustrate the reasoning on sigmoidal functions; say few words about other activation functions later.

Activation functions – tanh



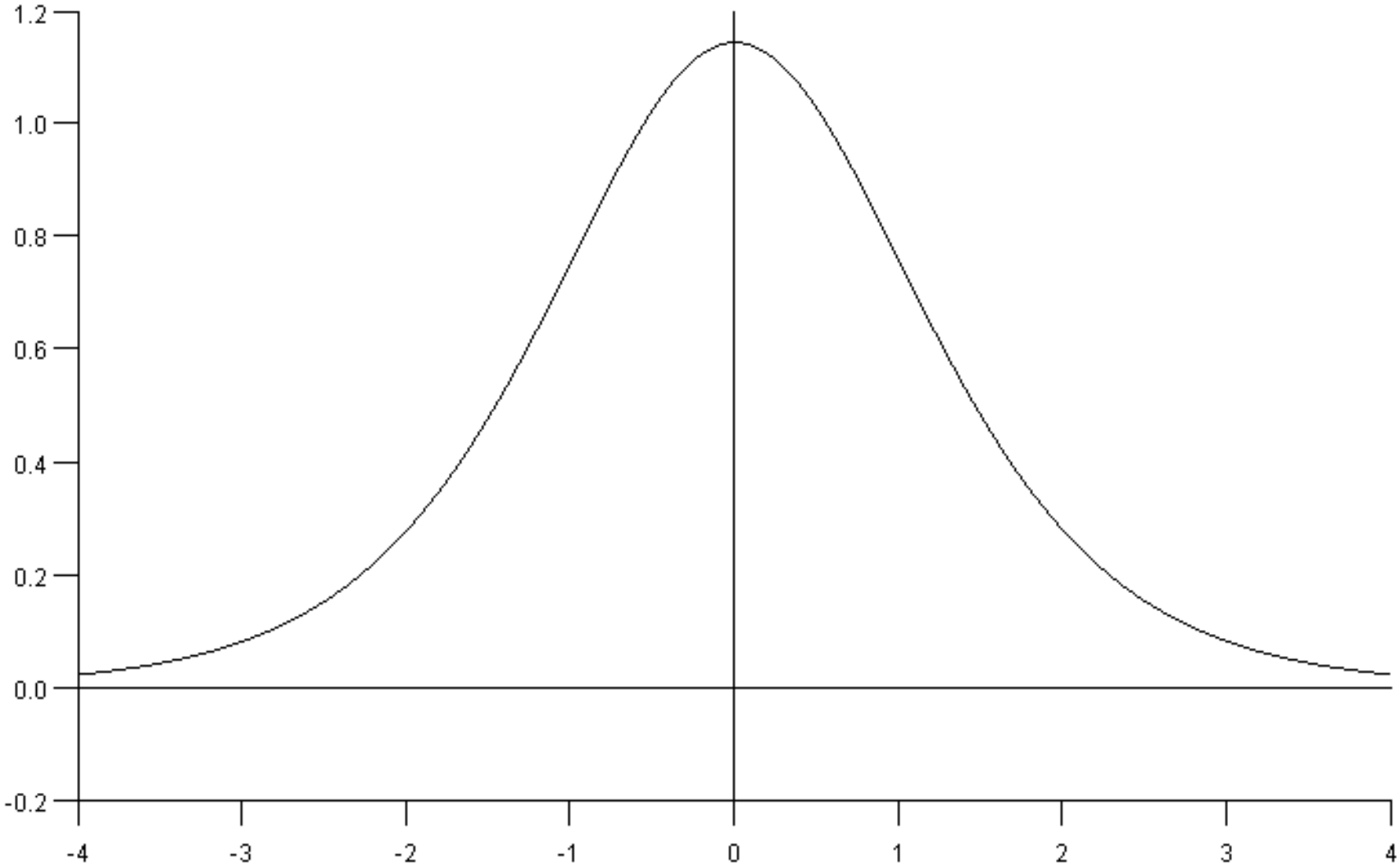
$\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$, we have $\lim_{\xi \to \infty} \sigma(\xi) = 1.7159$ and $\lim_{\xi \to -\infty} \sigma(\xi) = -1.7159$

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.

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).

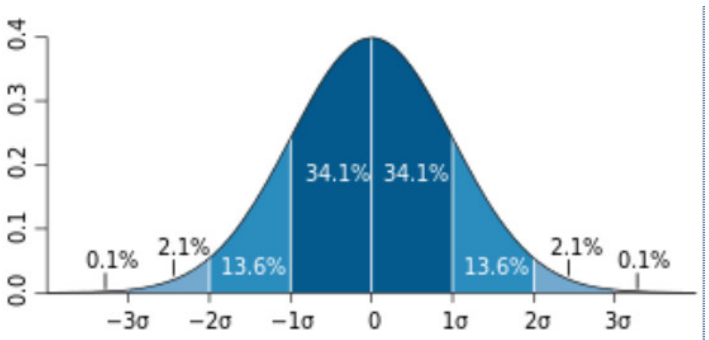
Input preprocessing

- 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)

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.

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 $[-1, 1]$,
  - σ saturates out of the interval $[-4, 4]$ (i.e. it is close to its limit values and its derivative is close to 0.

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 $[-1, 1]$,
  - 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.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data.
- Consider a neuron j from the first layer with d inputs. Assume that its weights are chosen uniformly from $[-w, w]$.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data.
- 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: $o_j \approx 1$.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data.
- 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 $w = \frac{\sqrt{3}}{\sqrt{d}}$.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data.
- 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 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 n is the number of inputs to the layer, m is the number of outputs of the layer (i.e. the number of neurons in the layer).

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:

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

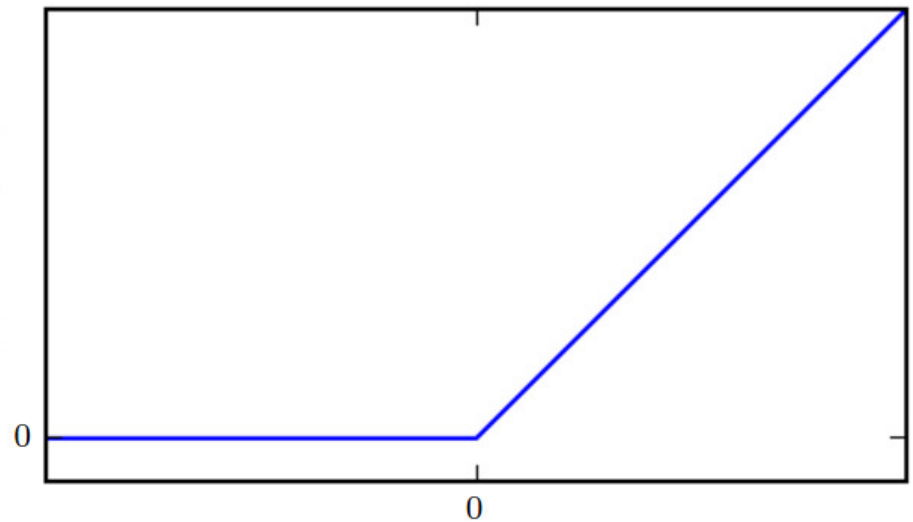
Here n is the number of inputs to the layer, m is the number of outputs of the layer (i.e. the number of neurons in the layer).

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.

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).

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: Given an output neuron $j \in Y$

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

for multi-class classification.

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: Given an output neuron $j \in Y$

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

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**:

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

... 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 $y = \sigma_j(\xi_j)$.

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.

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.

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?

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?

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 :-)

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.

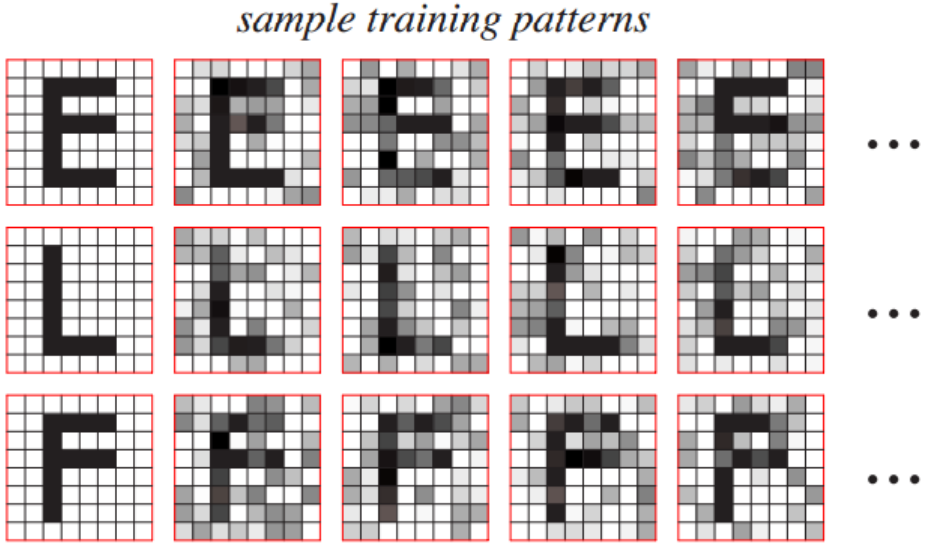
Solution: Optimal number of neurons :-)

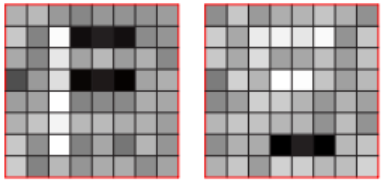
- there are some (useless) theoretical bounds
- there are algorithms dynamically adding/removing neurons (not much use nowadays)
- In practice:
  - start using a rule of thumb: the number of neurons $\approx$ ten times less than the number of training instances.
  - experiment, experiment, experiment.

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:





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.

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 $T_1, ..., T_k$ by *sampling from* $\mathcal{T}$ *uniformly with replacement*.

  If the number of samples is $|\mathcal{T}|$, 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).

- do forward and backward propagations only using the selected neurons

  (i.e. leave weights of the other neurons unchanged)

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).

- do forward and backward propagations only using the selected neurons

  (i.e. 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 and L2 regularization

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

Weight decay and L2 regularization

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 and L2 regularization

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 $\frac{2\zeta}{\varepsilon}(\vec{w} \cdot \vec{w})$ is the L2 regularization that penalizes large weights.

More optimization, regularization ...

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

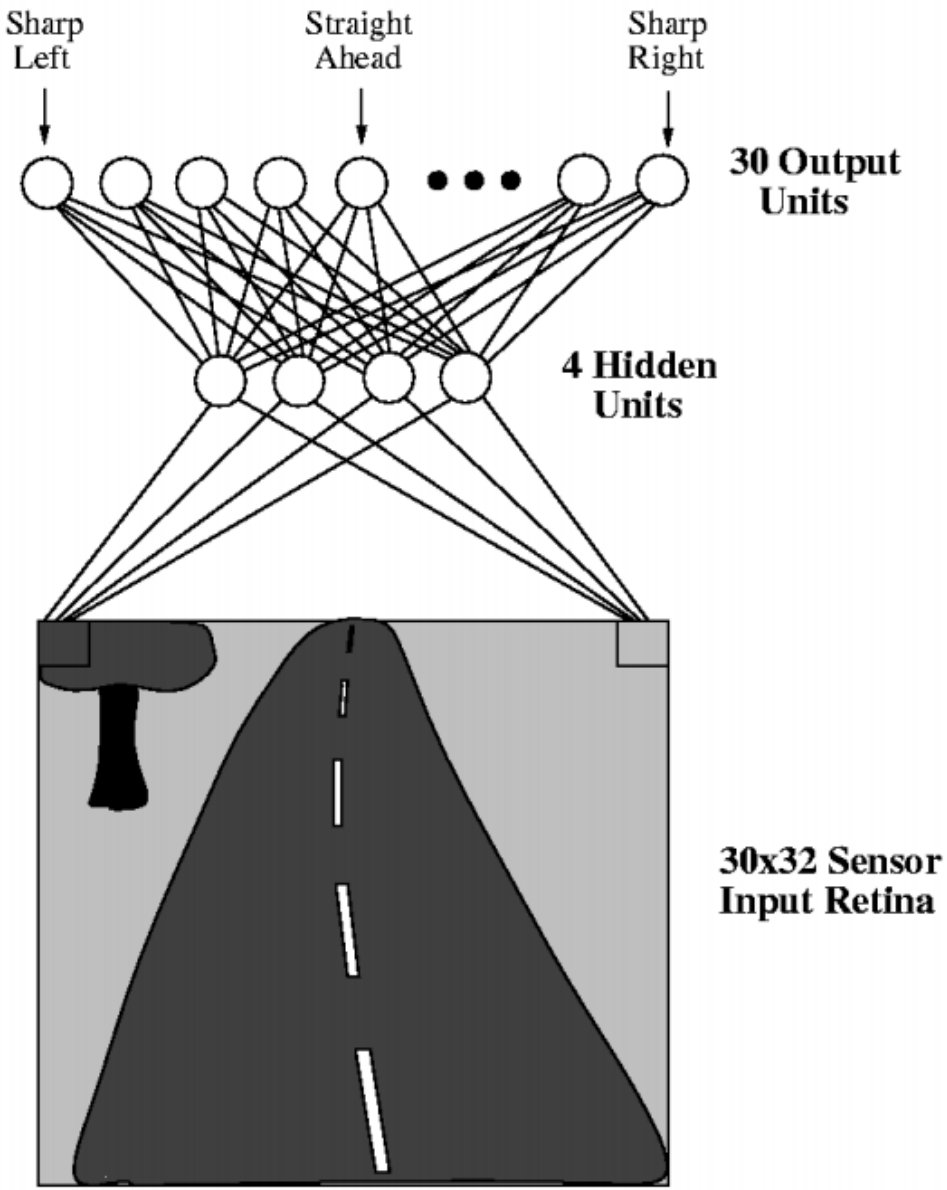
For a very nice survey see

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

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

Some applications

ALVINN (history)



ALVINN

Architecture:

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

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.

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.

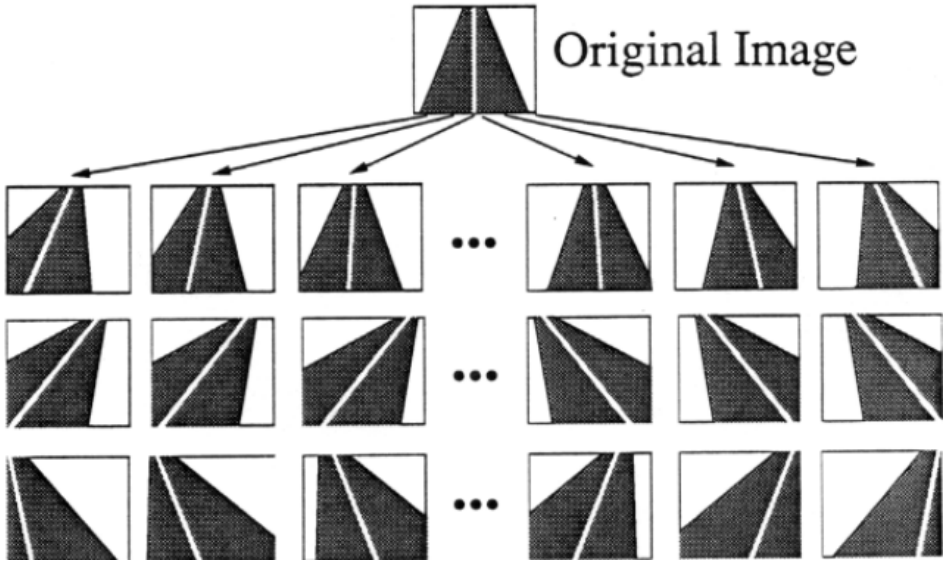
ALVINN – Selection of training examples

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

ALVINN – Selection of training examples

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

- 15 distorted copies of each image:

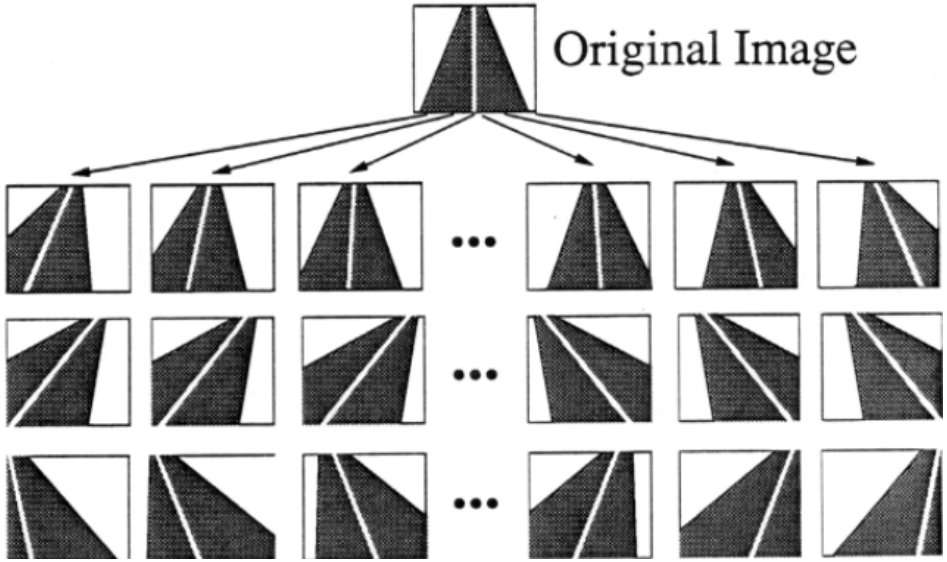


- desired output generated for each copy

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 rate
- 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).

ALVINN - learning

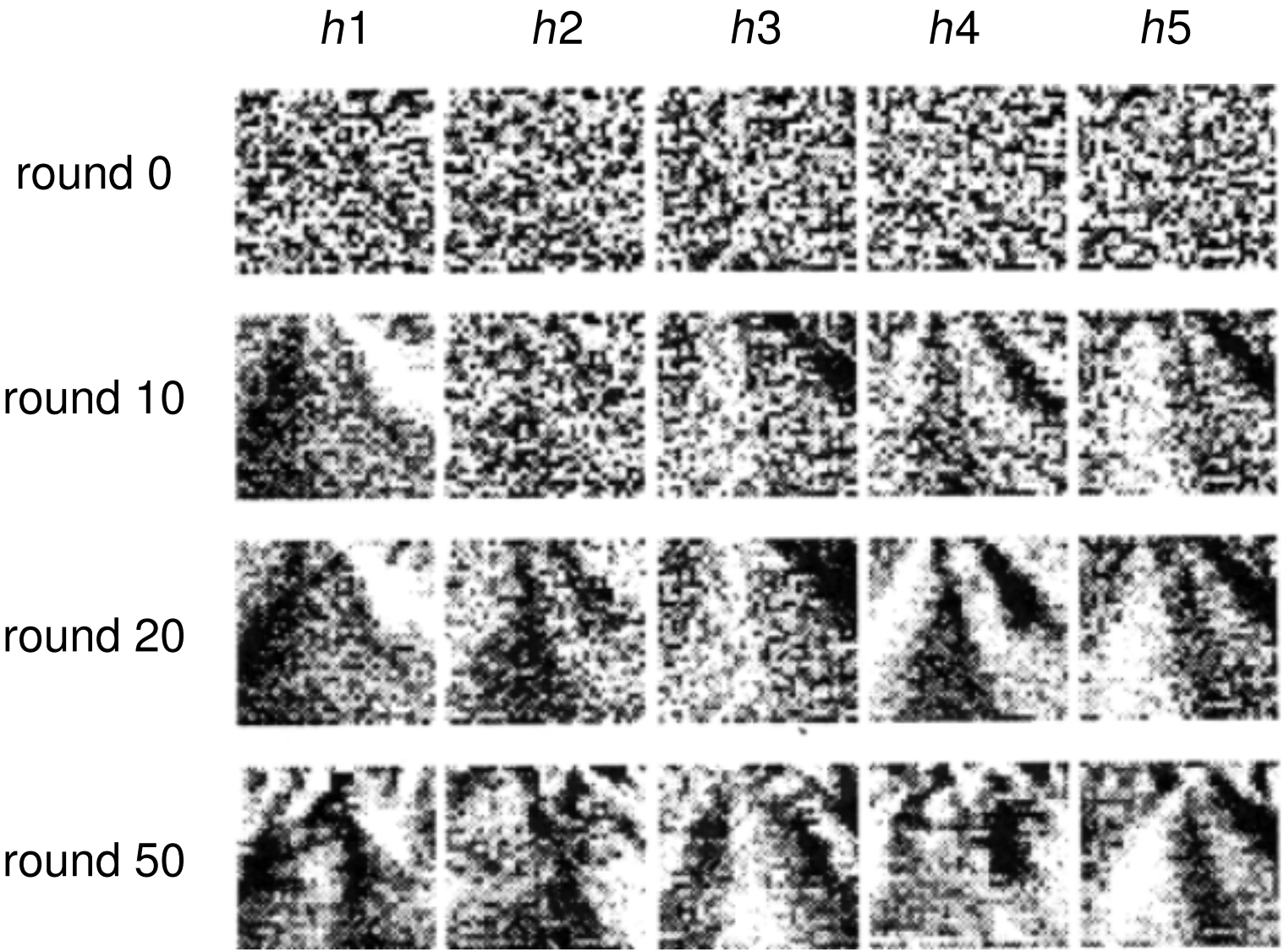
- pure backpropagation
- constant learning rate
- momentum, slowly increasing.

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.

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.

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

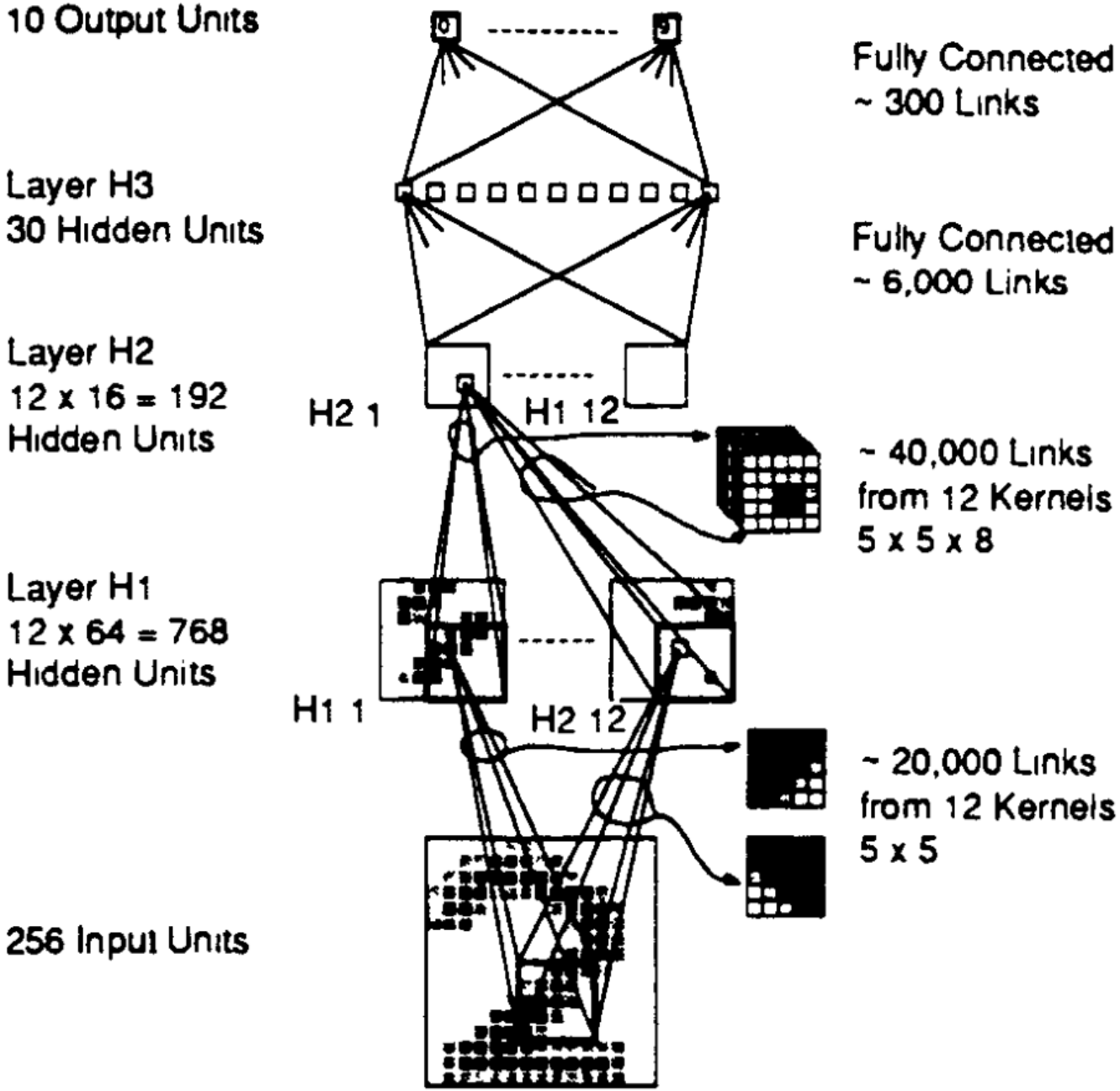
One of the best "old" results is the following:

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

Abstract: 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

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

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

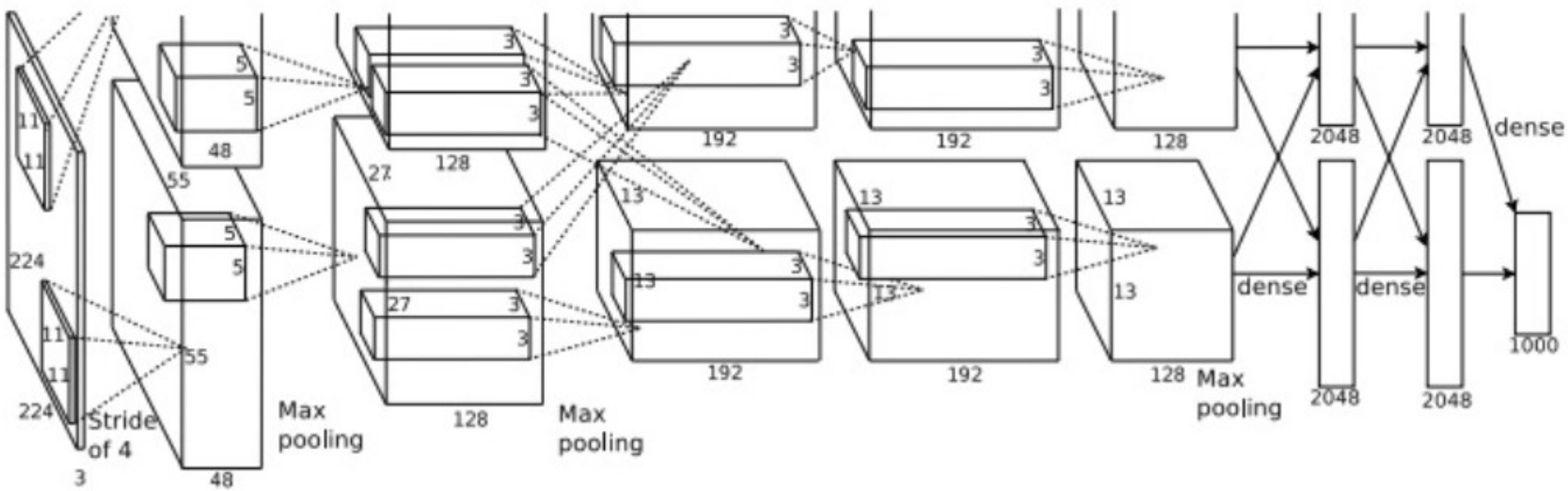
The rest of the lecture is based on the online book Neural Networks and Deep Learning by Michael Nielsen.
`http://neuralnetworksanddeeplearning.com/index.html`

- 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

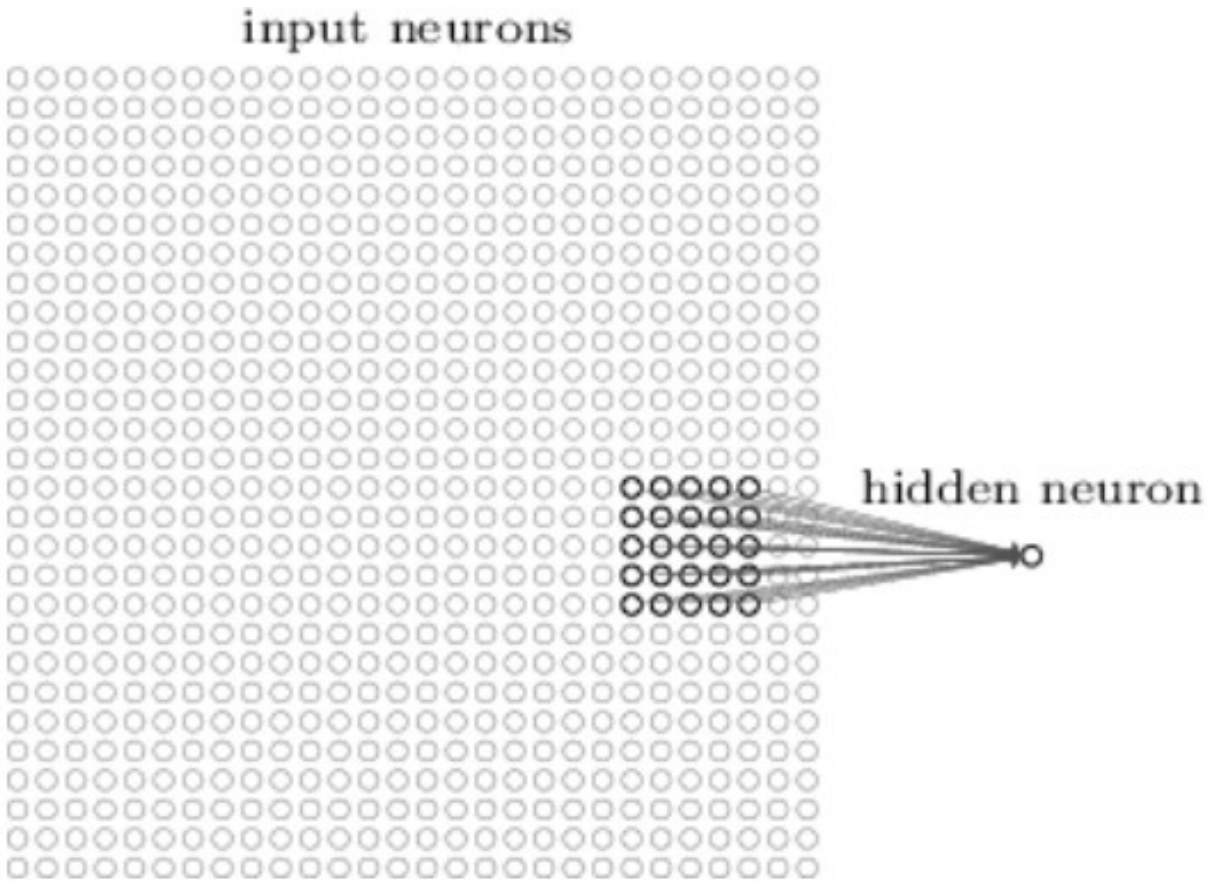
AlexNet

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

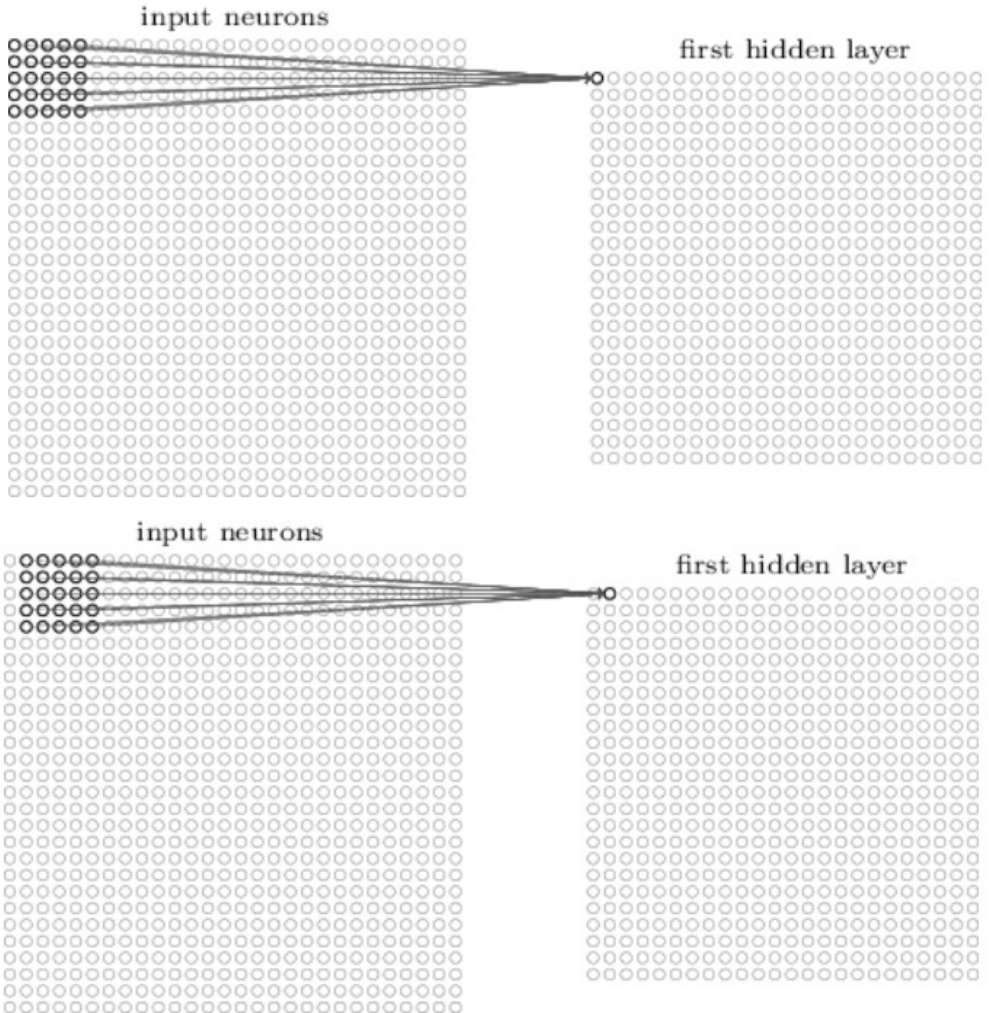


Every neuron is connected with a field of $k \times k$ (in this case 5×5) neurons in the lower layer (this filed is *receptive field*).

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

Convolutional networks - stride length

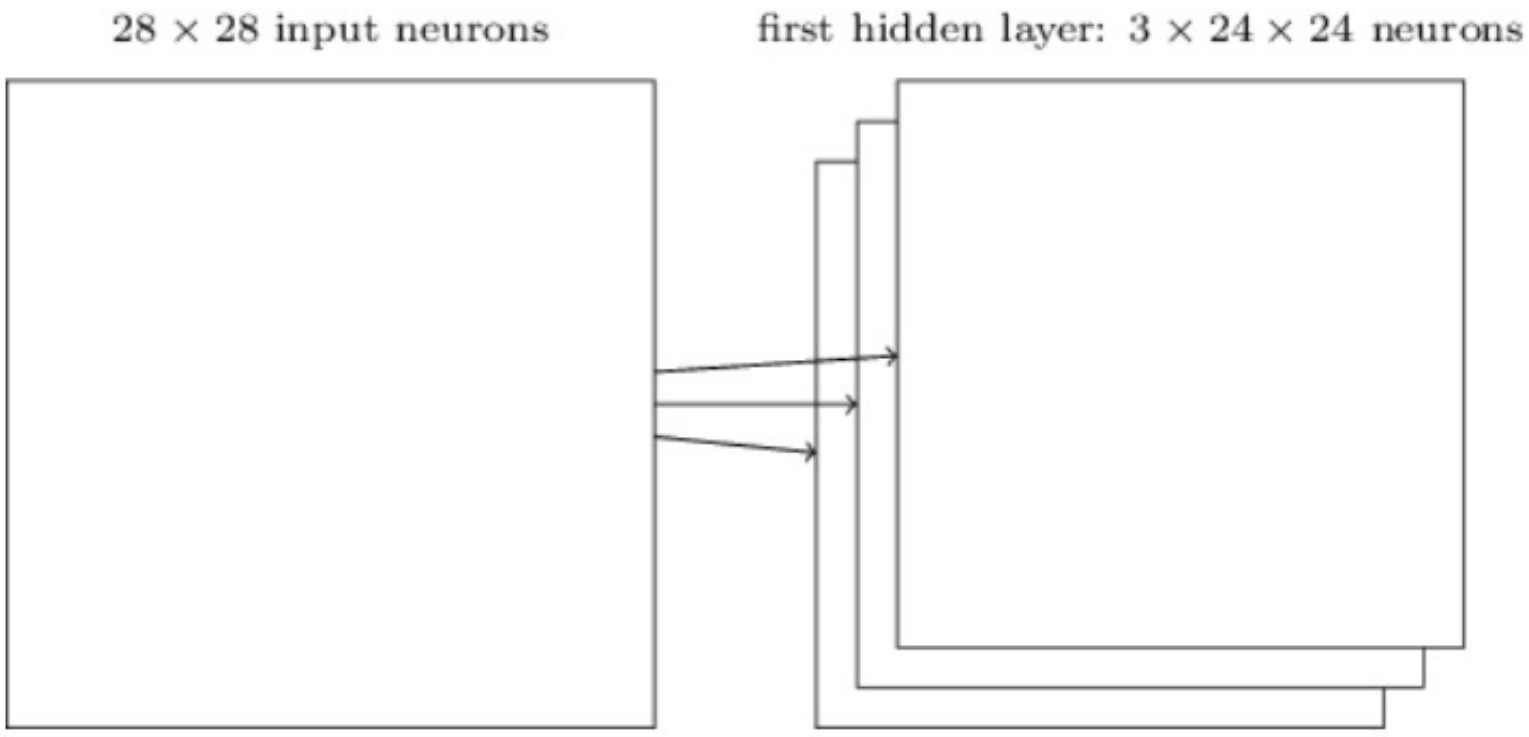
Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:



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*!

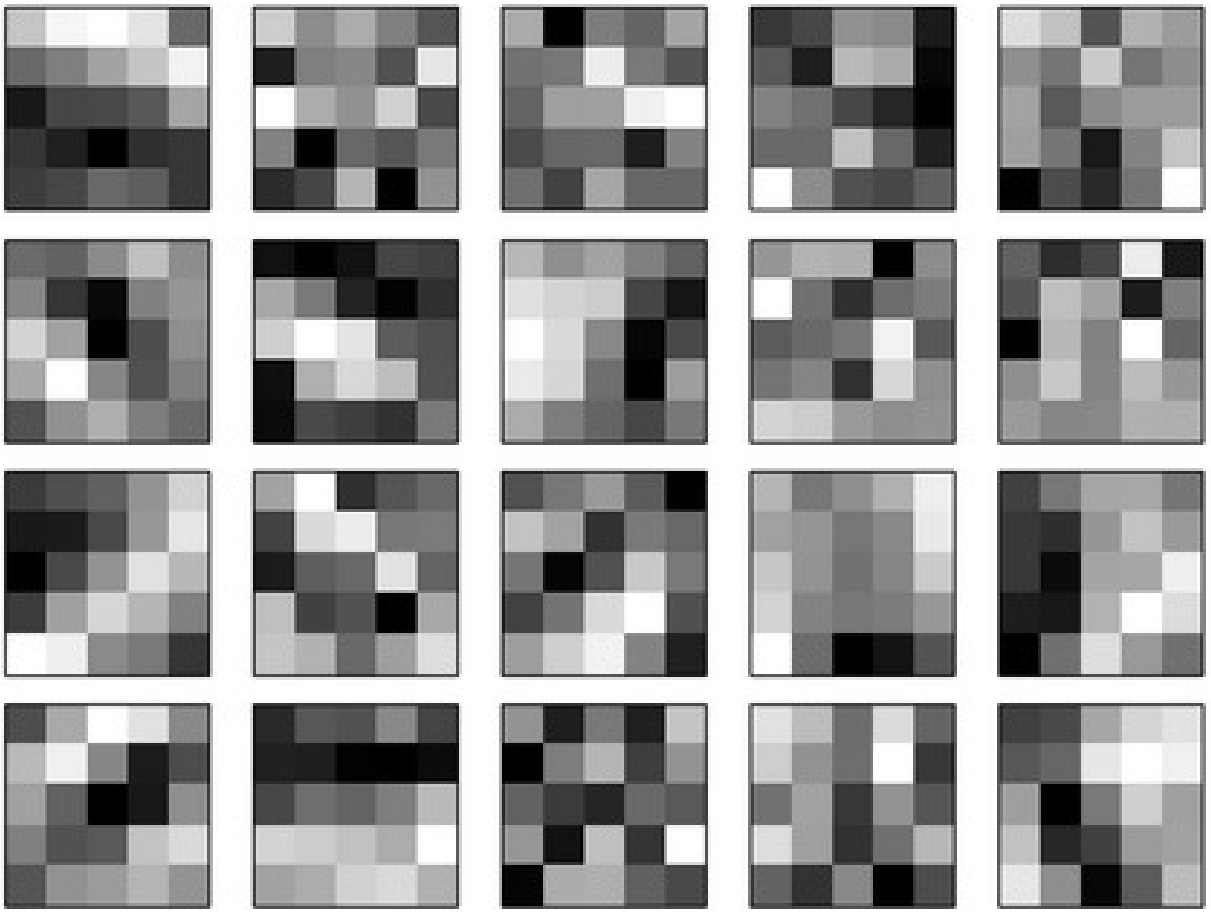
Feature maps



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

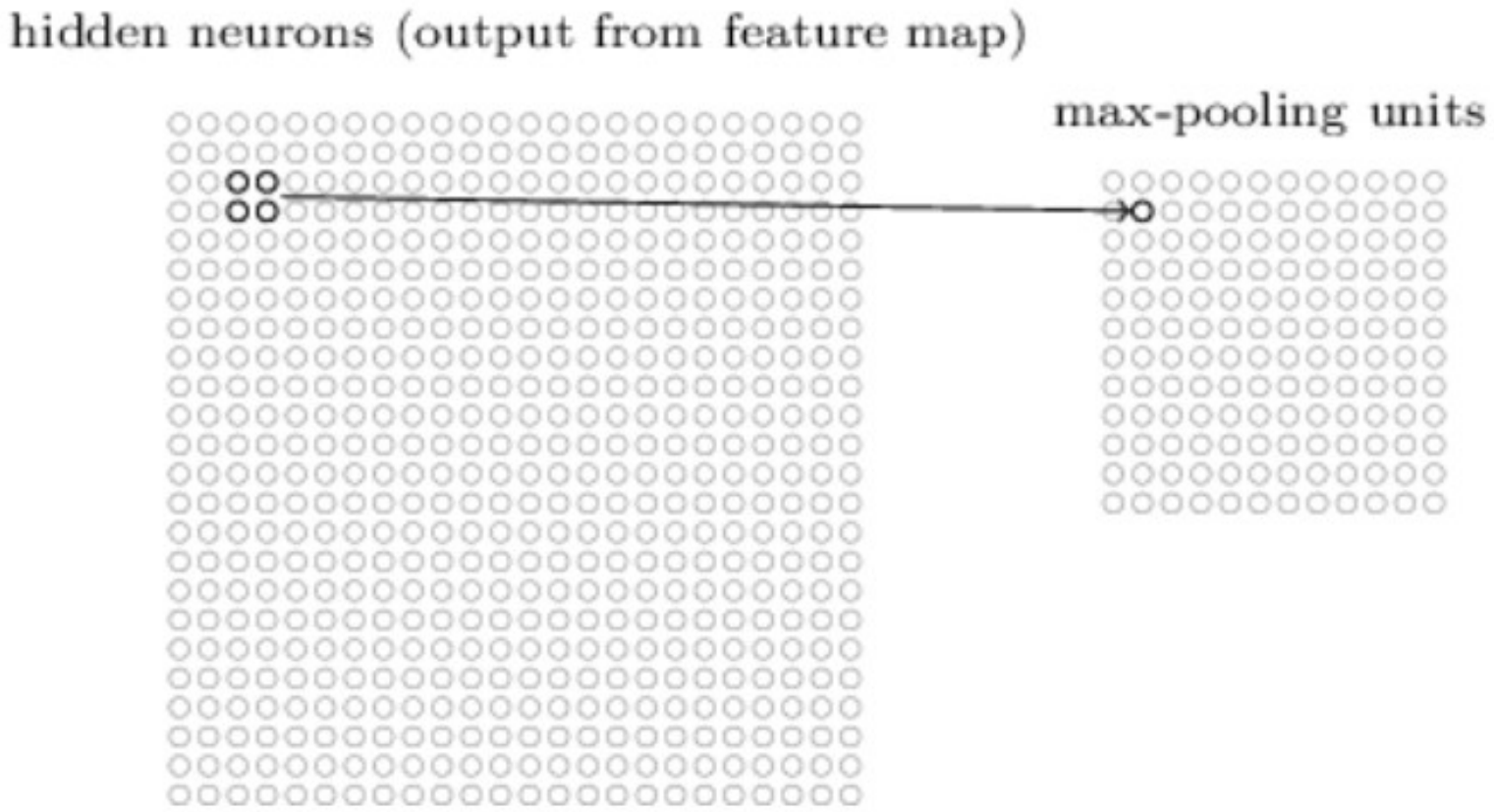
Typically, we consider several feature maps in a single layer.

Trained feature maps



(20 feature maps, receptive fields 5×5)

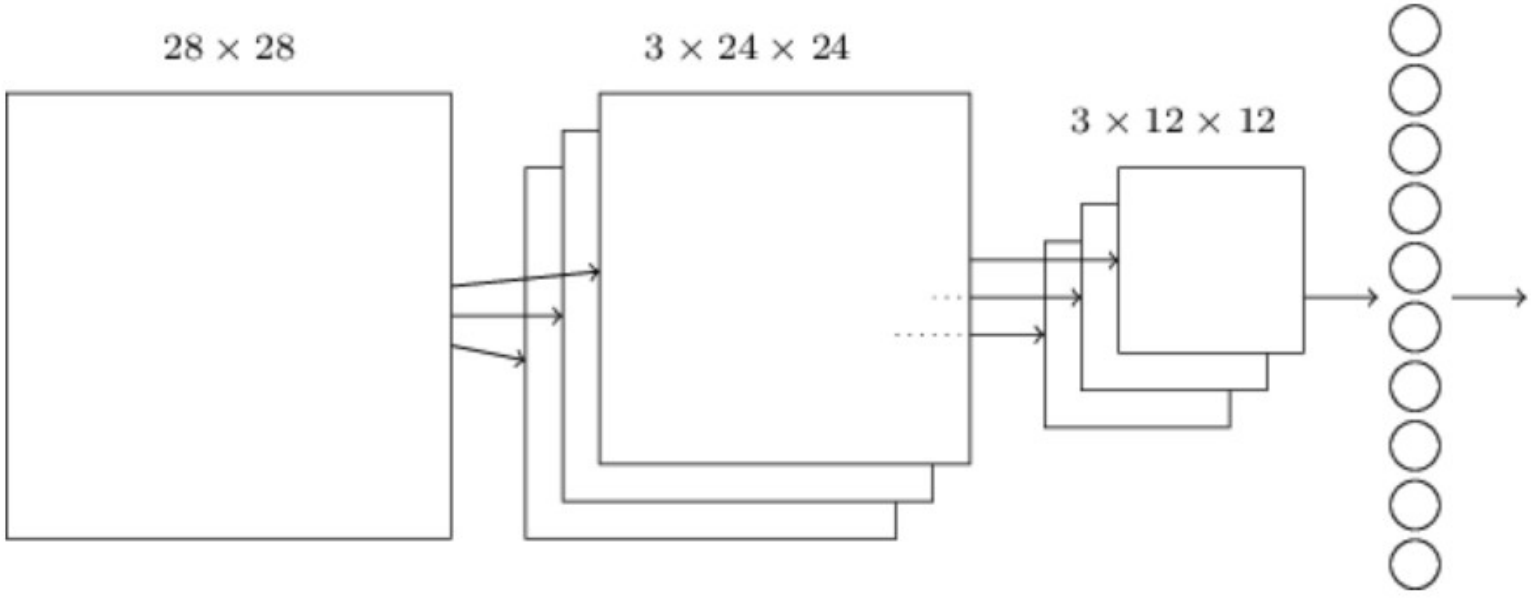
Pooling



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

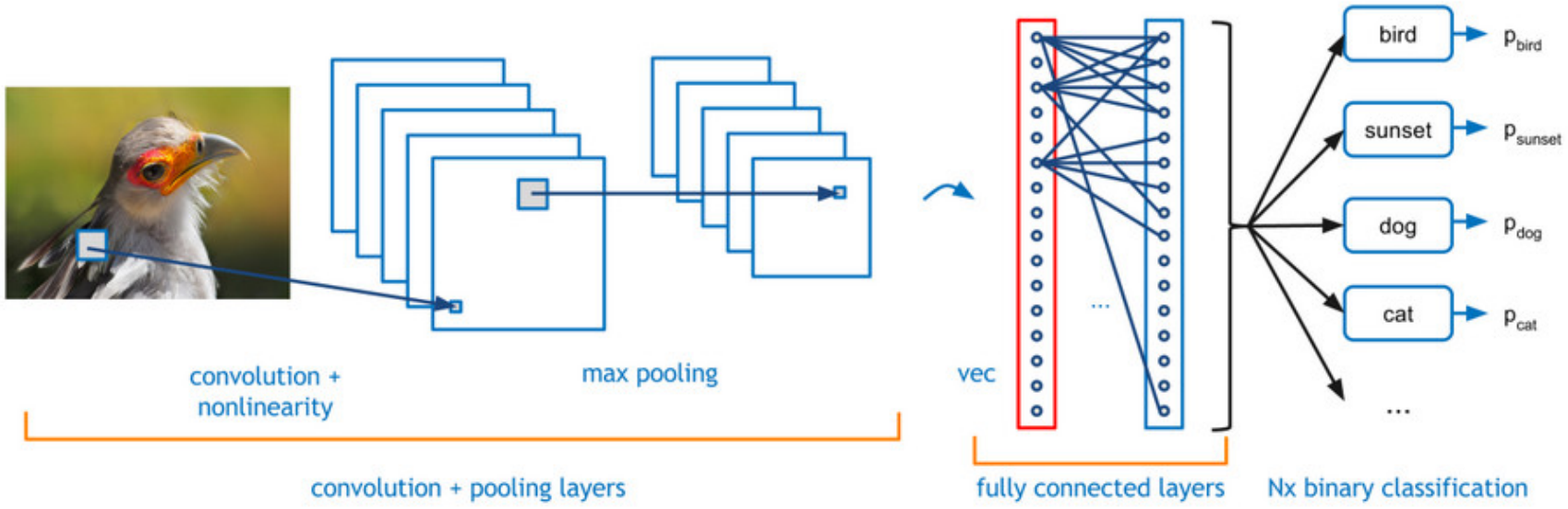


28×28 input image, 3 feature maps, each feature map has its own max-pooling (field 5×5, stride = 1), 10 output neurons.

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



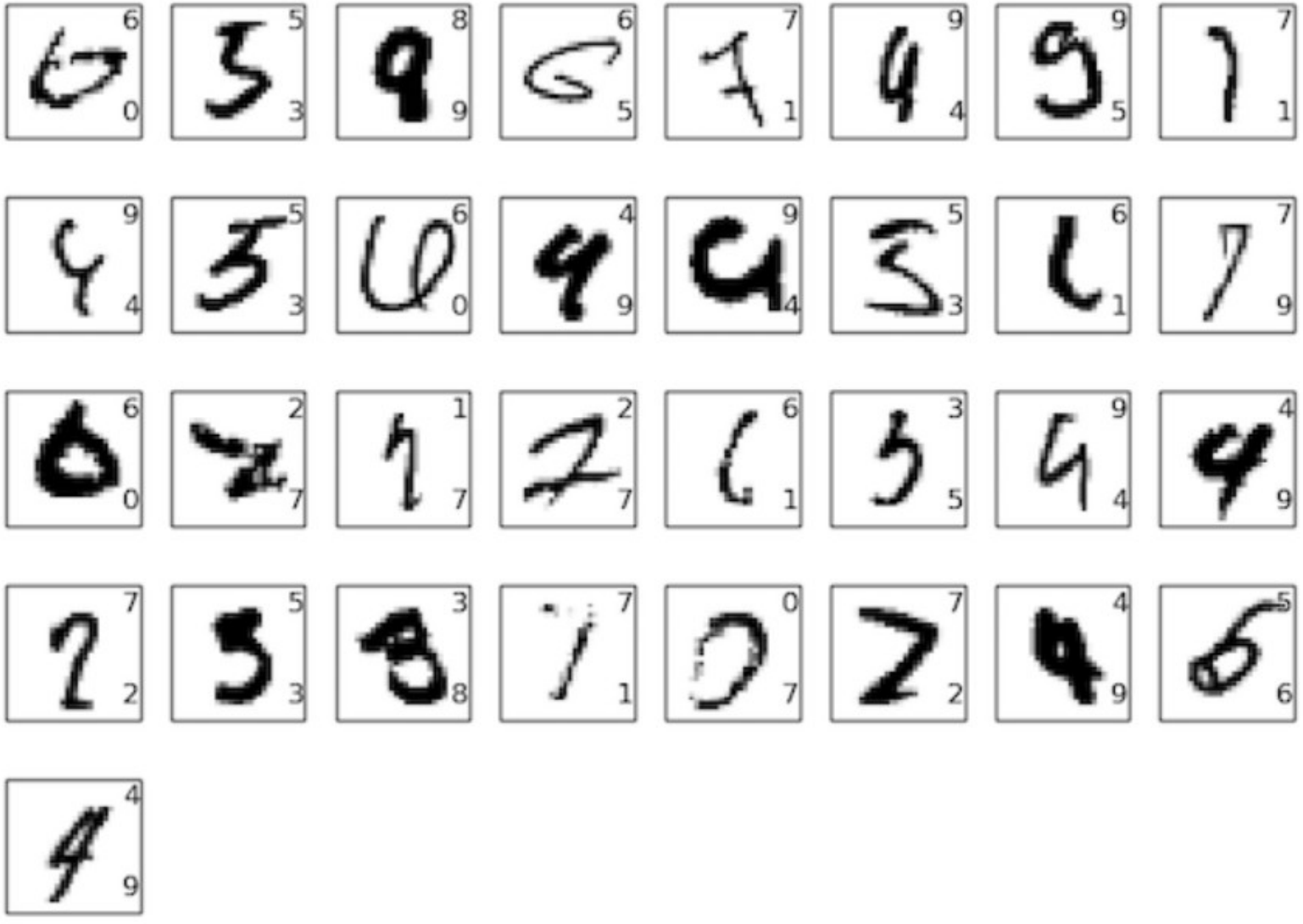
Simple convolutional network vs MNIST

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.

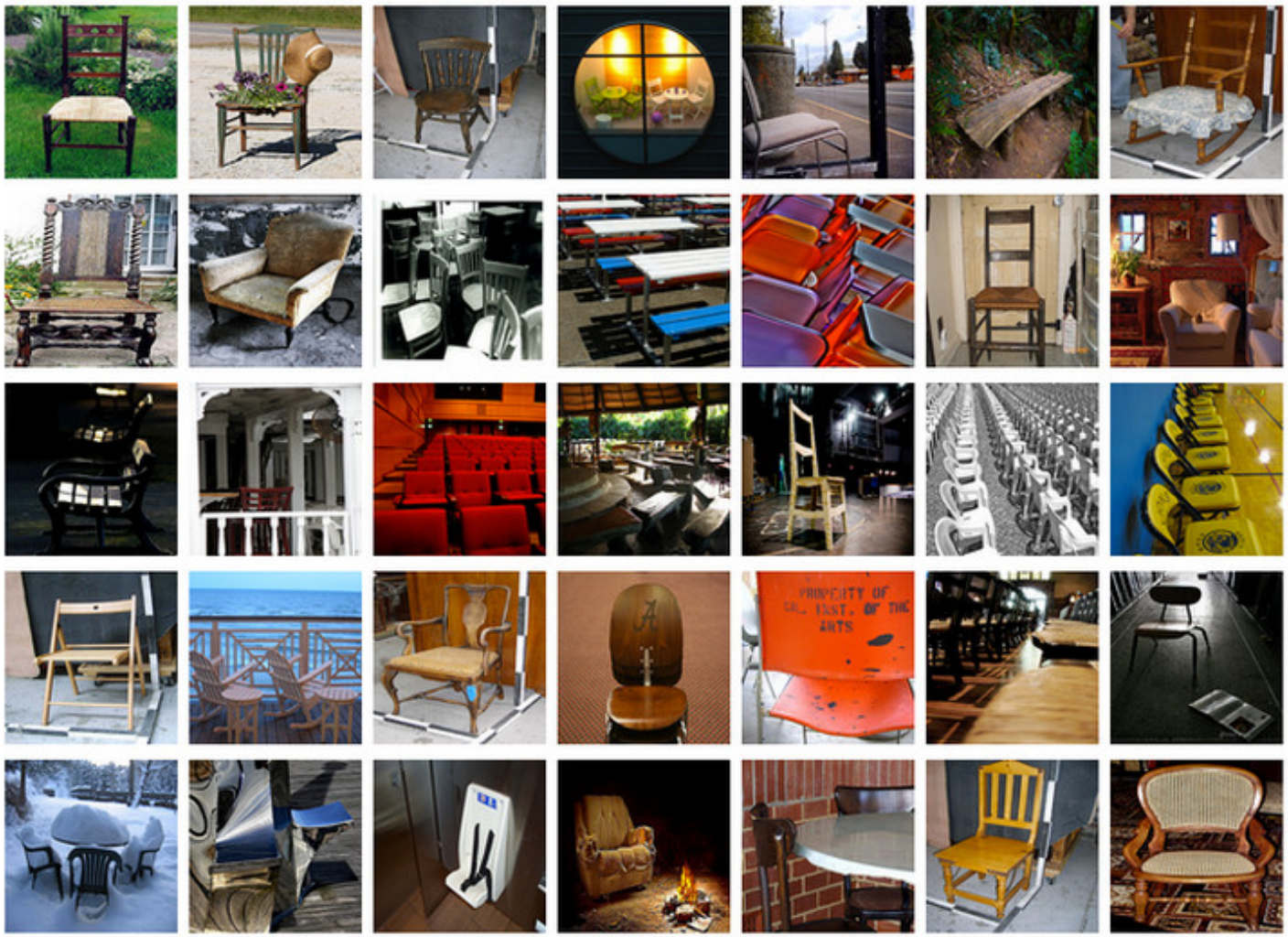
MNIST

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.

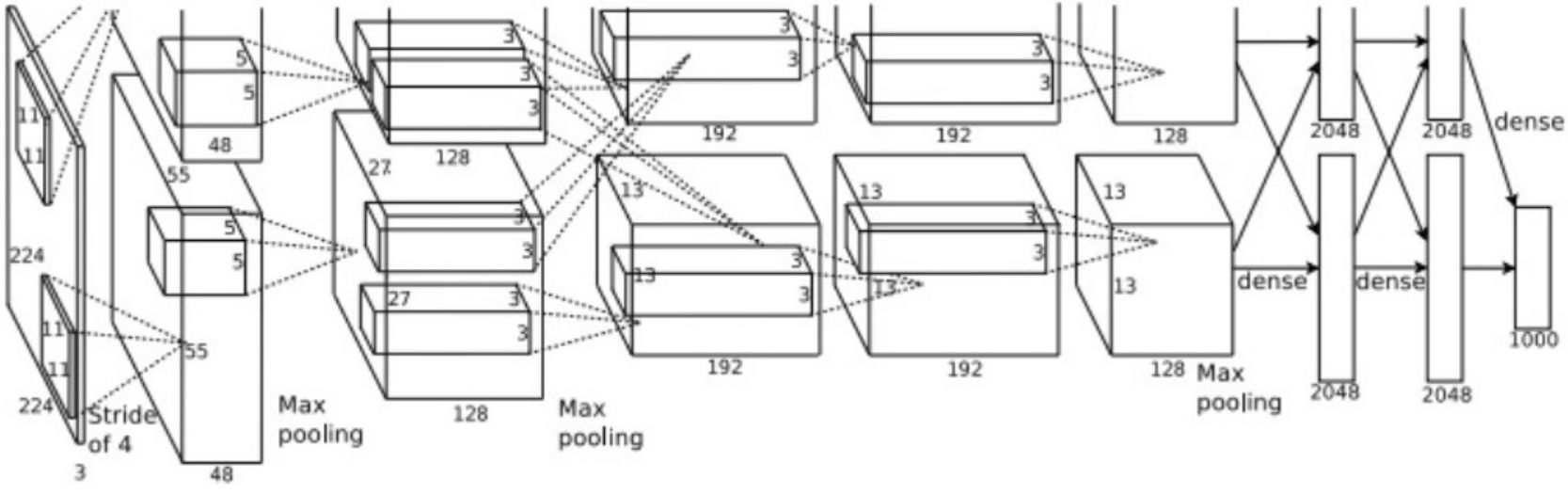
Started in 2010, assisted in breakthrough in image recognition.

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)

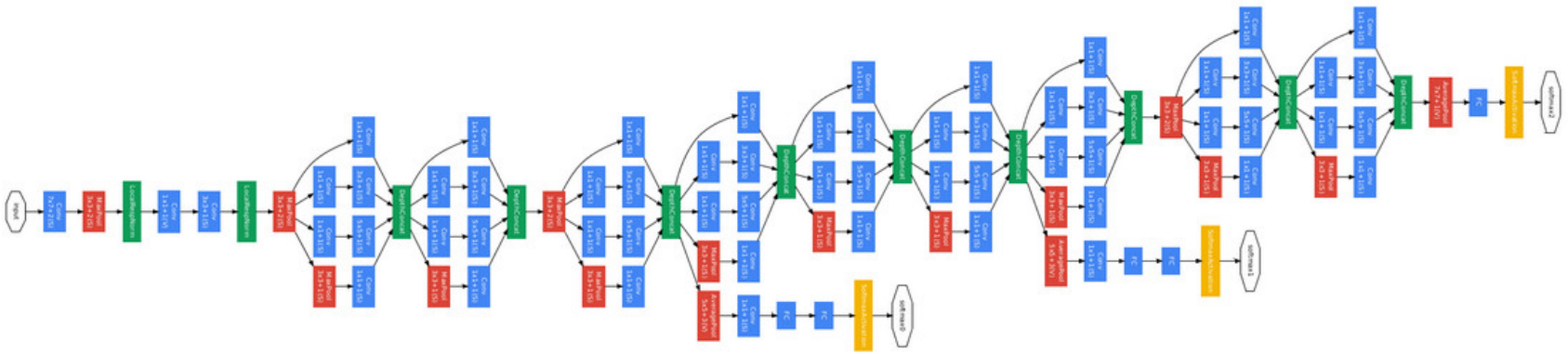
Výsledky:

- 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.

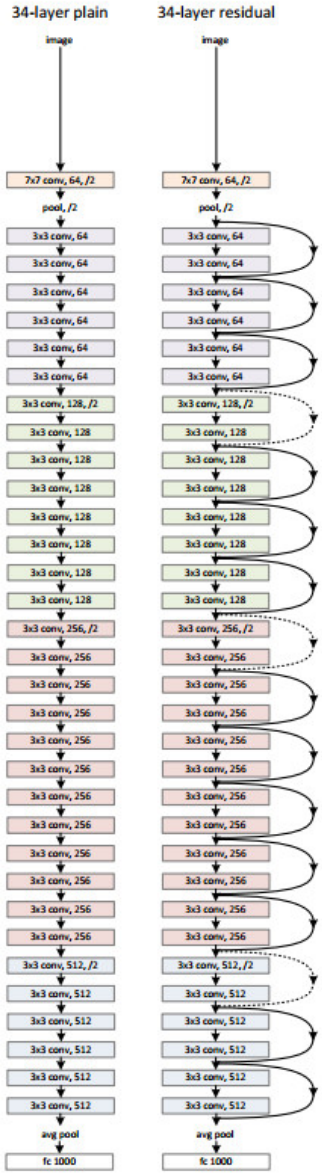
GoogLeNet: deep convolutional network, 22 layers



Results:

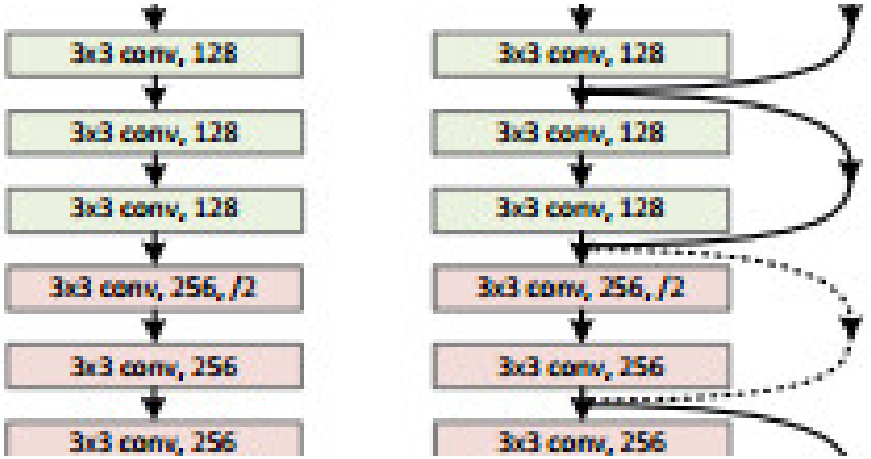
- Accuracy 93.33% top-5

ILSVRC 2015



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

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

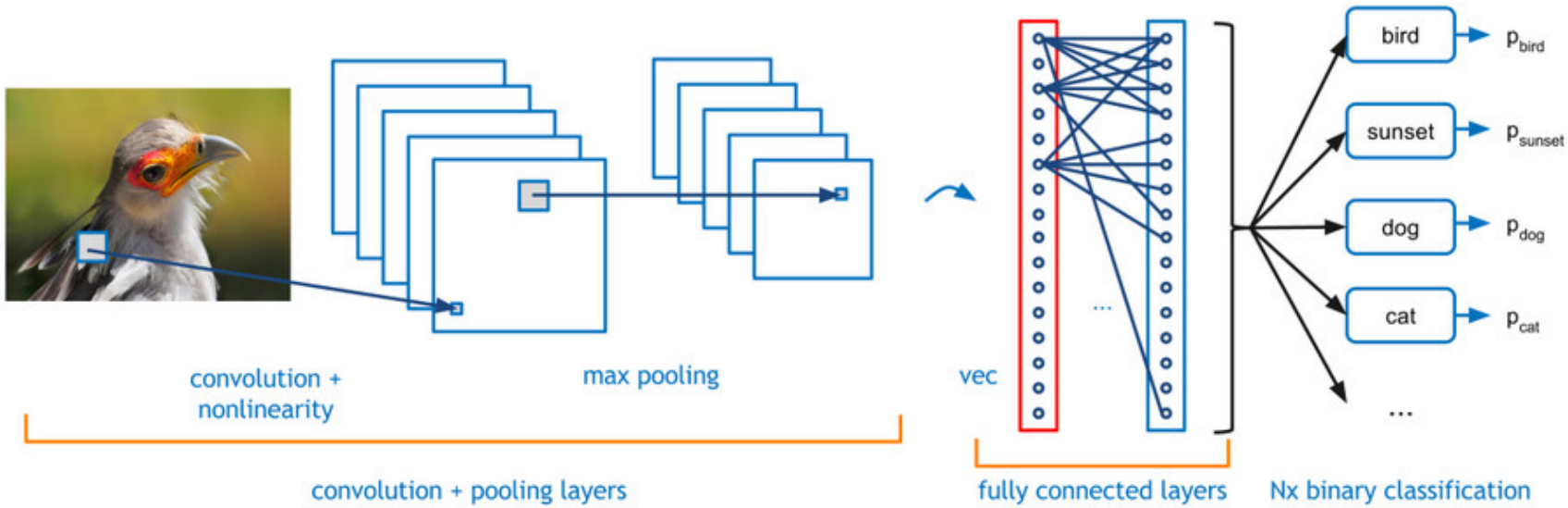


Superhuman convolutional nets?!

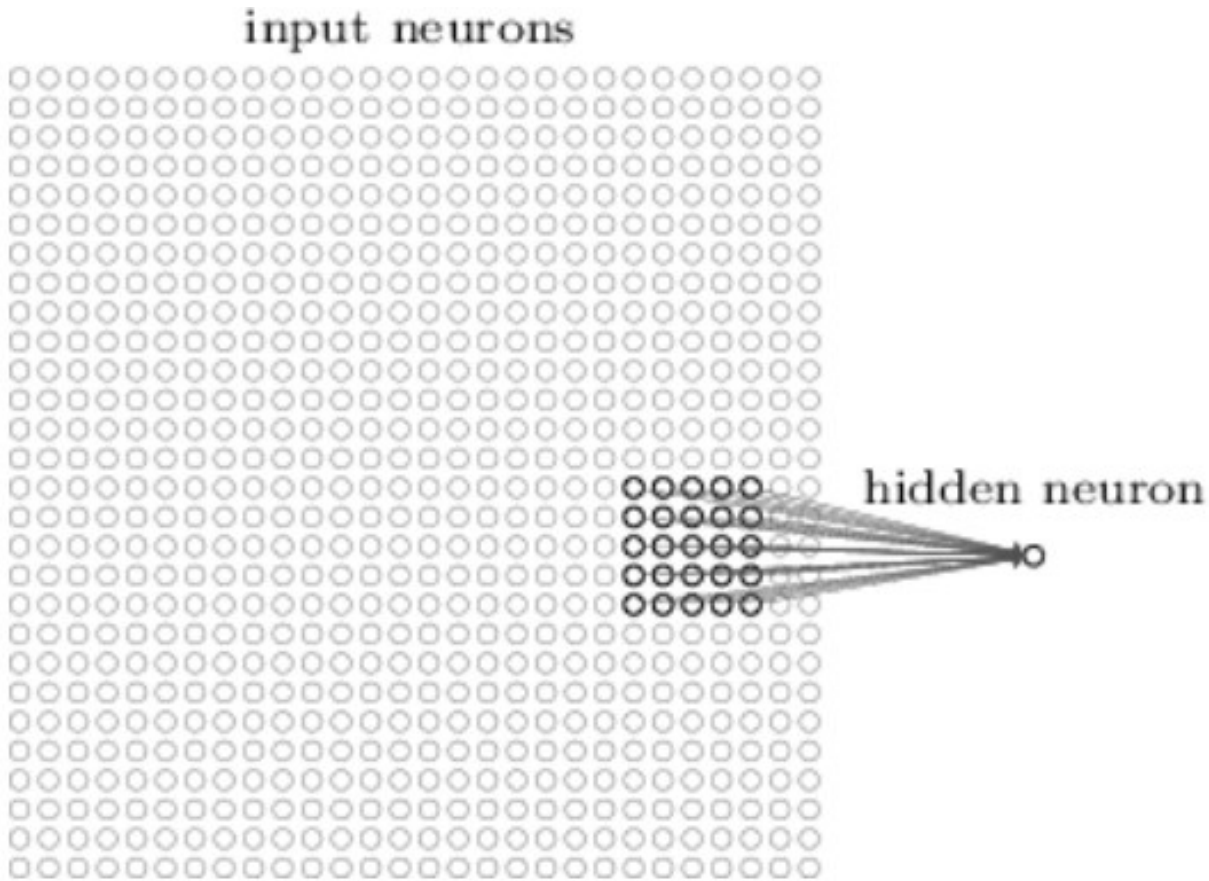
Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought 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 networks – theory

Convolutional network



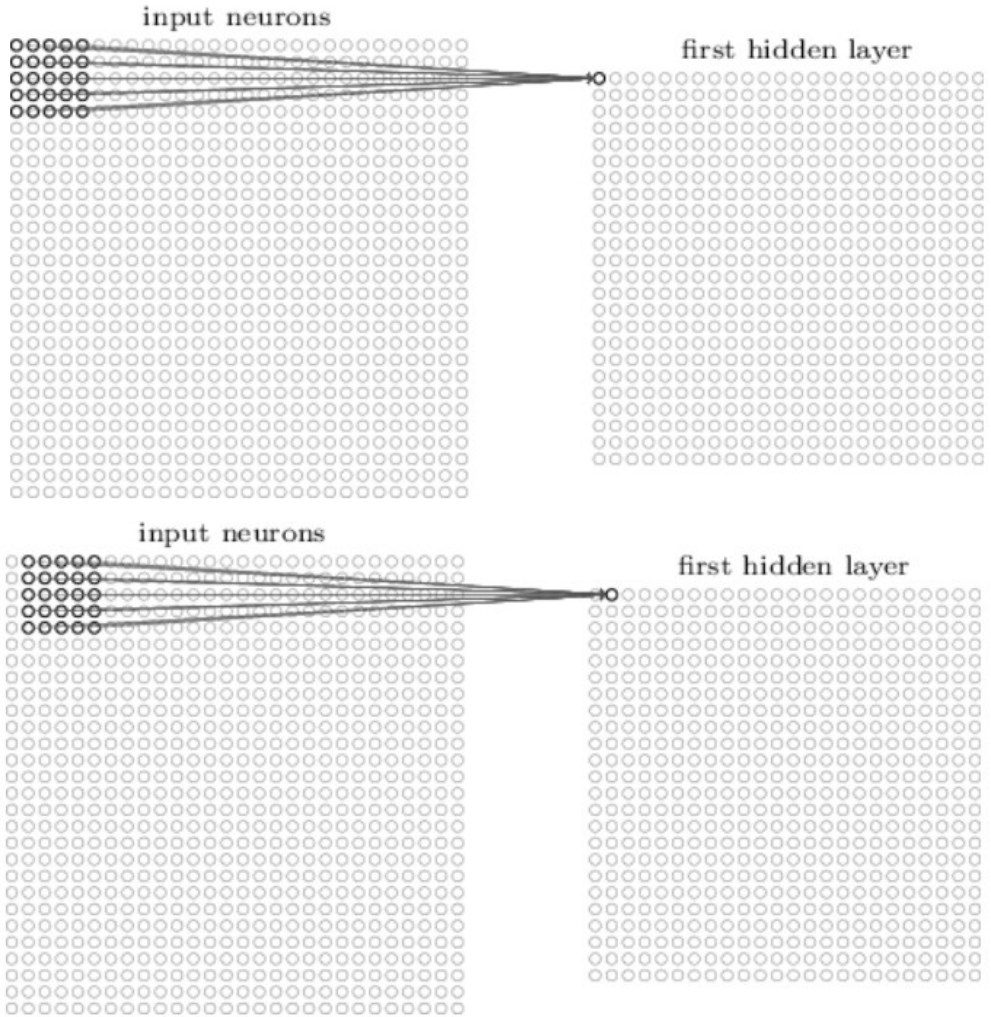
Convolutional layers



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

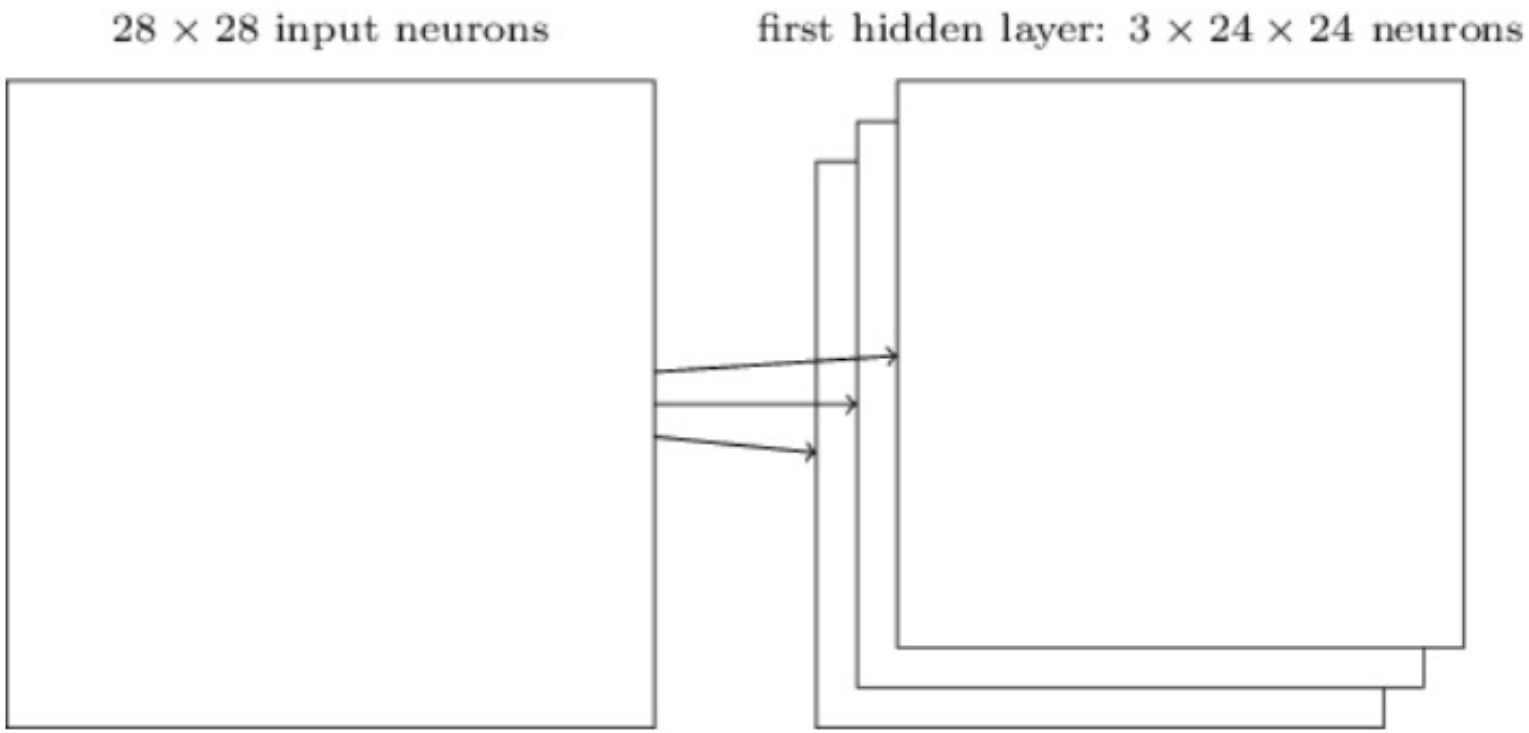
Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

Convolutional layers



Neurons grouped into *feature maps* sharing weights.

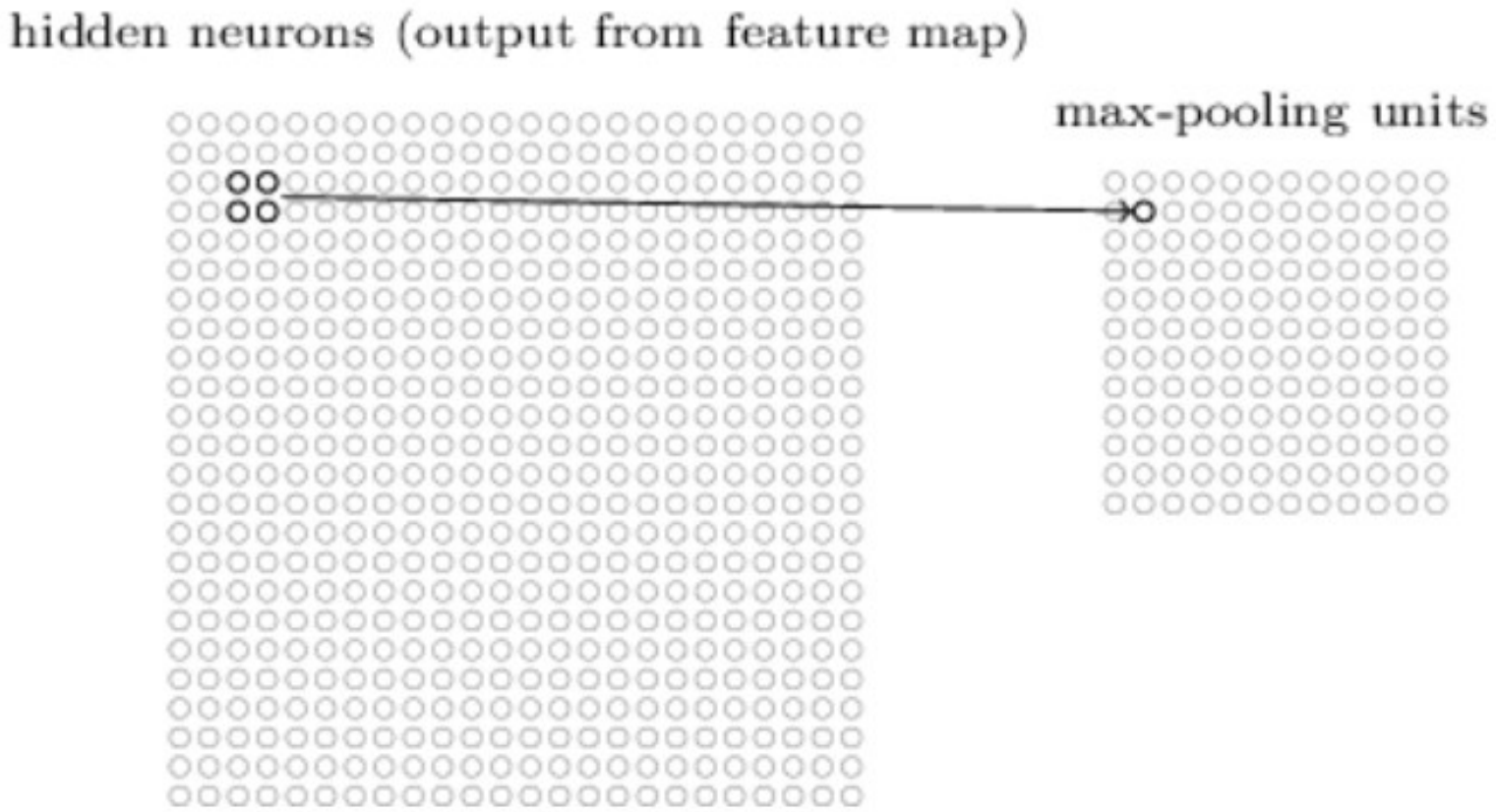
Convolutional layers



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}.

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

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 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)
- $[ji]$ is a set of all connections (i.e. pairs of neurons) sharing the weight w_{ji}.

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):

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

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 squared 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 $\frac{\partial E_k}{\partial w_{ji}}$?

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:

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

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:

$$\frac{\partial E_k}{\partial w_{ji}} = \sum_{r\ell \in [ji]} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot y_\ell$$

- 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 squared error, for other error functions the derivative w.r.t. outputs will be different.

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 squared 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:

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

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 squared 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

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

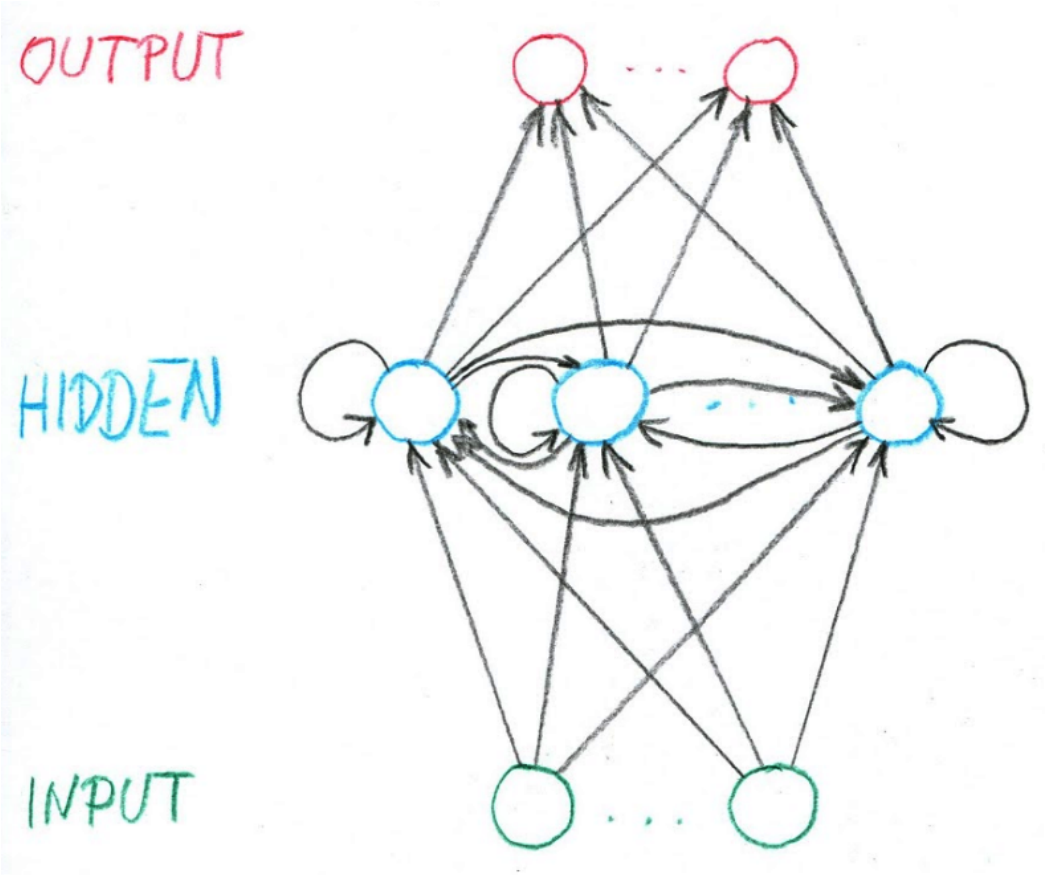
I.e. gradient can be propagated from the output layer downwards as in MLP.

Convolutional networks – summary

- 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.

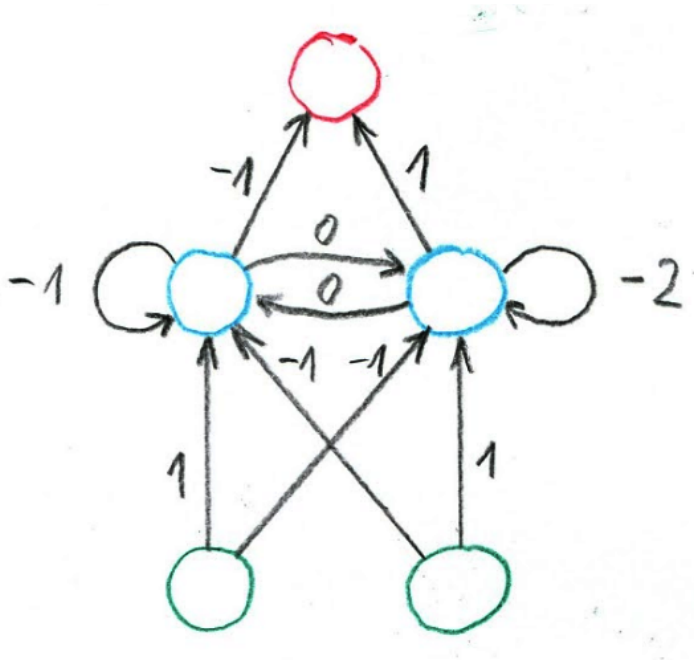
Recurrent Neural Networks - LSTM

RNN



- Input:
  $\vec{x} = (x_1, \ldots, x_M)$
- Hidden:
  $\vec{h} = (h_1, \ldots, h_H)$
- Output:
  $\vec{y} = (y_1, \ldots, y_N)$

RNN example

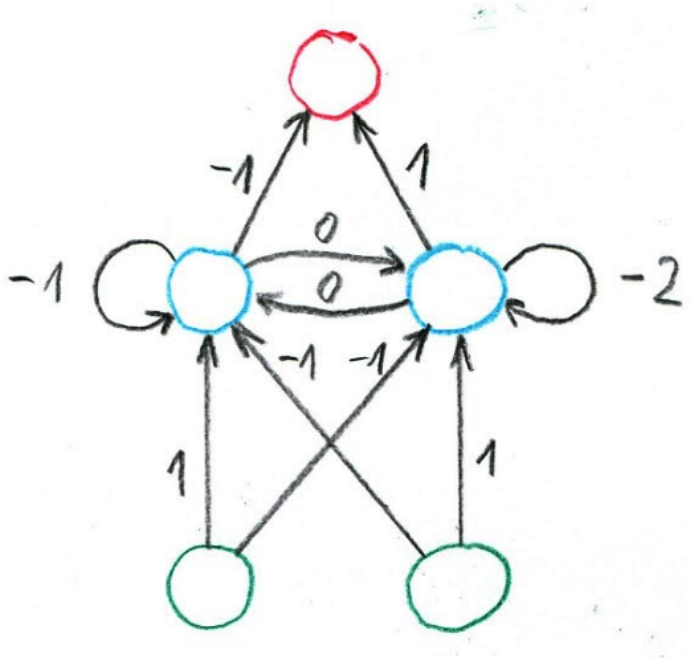


Activation function:

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

y		1	0	1	
h	(0,0)	(1,1)	(1,0)	(0,1)	$\cdots$
x		(0,0)	(1,0)	(1,1)	

RNN example

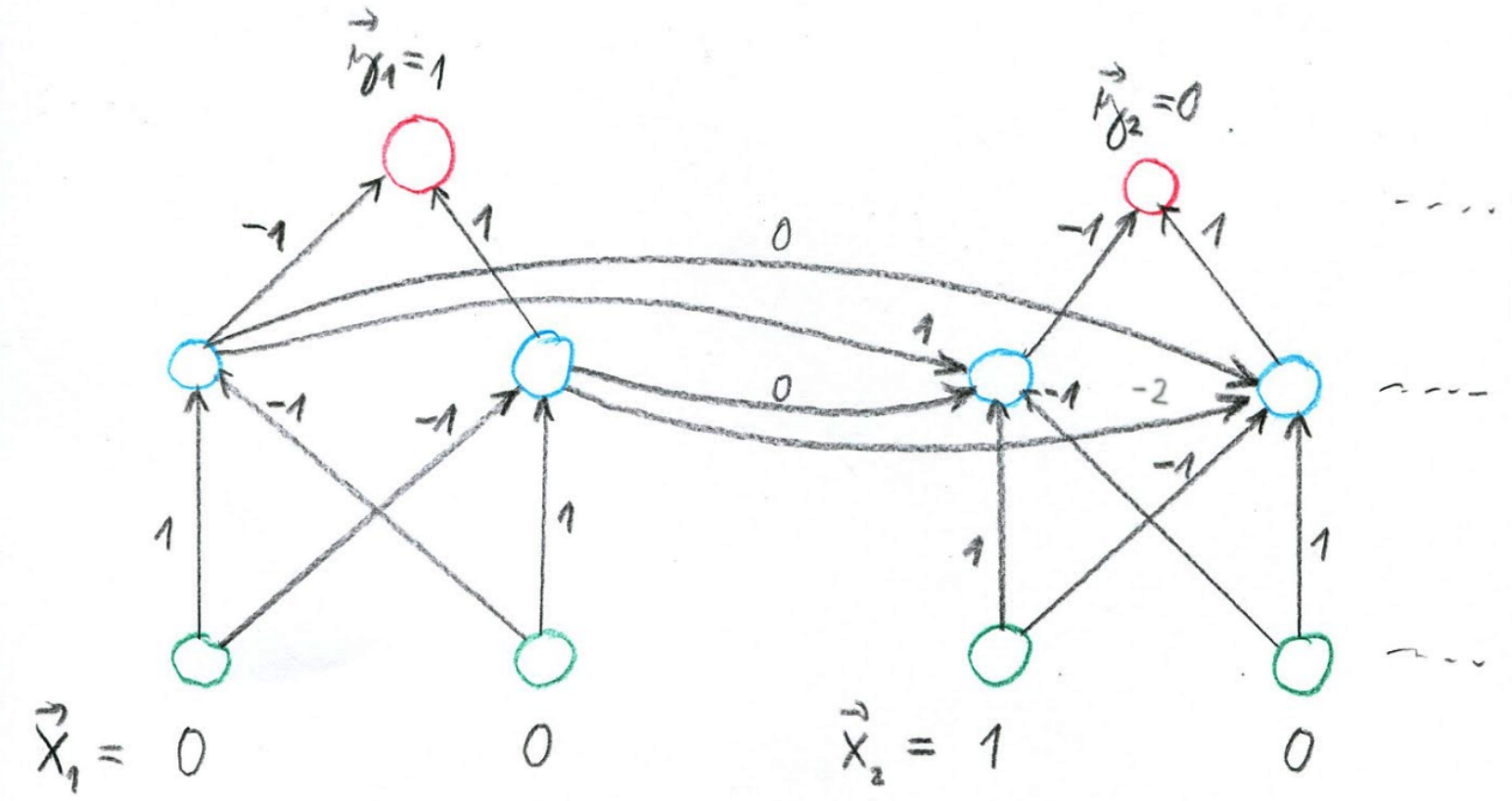


Activation function:

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

y		$\vec{y}_1 = 1$	$\vec{y}_2 = 0$	$\vec{y}_3 = 1$	
h	$\vec{h}_0 = (0,0)$	$\vec{h}_1 = (1,1)$	$\vec{h}_2 = (1,0)$	$\vec{h}_3 = (0,1)$	$\cdots$
x		$\vec{x}_1 = (0,0)$	$\vec{x}_2 = (1,0)$	$\vec{x}_3 = (1,1)$	

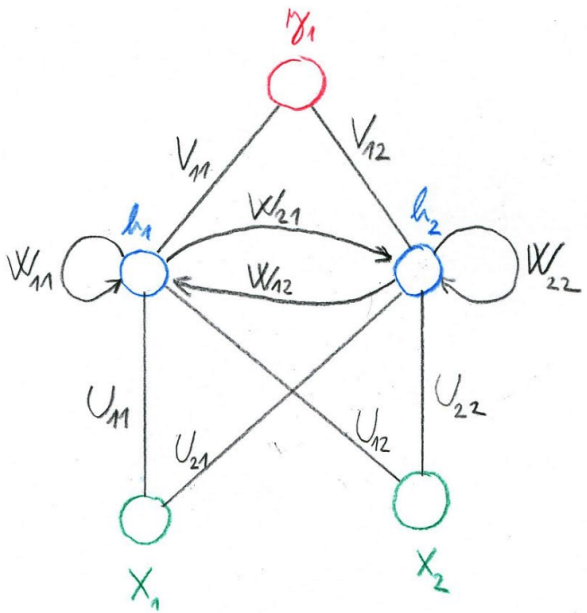
RNN example



y		$\vec{y}_1 = 1$	$\vec{y}_2 = 0$	$\vec{y}_3 = 1$	
h	$\vec{h}_0 = (0,0)$	$\vec{h}_1 = (1,1)$	$\vec{h}_2 = (1,0)$	$\vec{h}_3 = (0,1)$	$\cdots$
x		$\vec{x}_1 = (0,0)$	$\vec{x}_2 = (1,0)$	$\vec{x}_3 = (1,1)$	

RNN – formally

- M inputs: $\vec{x} = (x_1, \ldots, x_M)$
- H hidden neurons: $\vec{h} = (h_1, \ldots, h_H)$
- N output neurons: $\vec{y} = (y_1, \ldots, y_N)$
- Weights:
  - $U_{kk'}$ from input $x_{k'}$ to hidden h_k
  - $W_{kk'}$ from hidden $h_{k'}$ to hidden h_k
  - $V_{kk'}$ from hidden $h_{k'}$ to output y_k



RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

$$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

  $$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$

  $$\vec{h}_t = (h_{t1}, \ldots, h_{tH})$$

  We have $\vec{h}_0 = (0, \ldots, 0)$ and

  $$\vec{h}_{tk} = \sigma\left(\sum_{k'=1}^{M} U_{kk'} x_{tk'} + \sum_{k'=1}^{H} W_{kk'} h_{(t-1)k'}\right)$$

RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

  $$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$

  $$\vec{h}_t = (h_{t1}, \ldots, h_{tH})$$

  We have $\vec{h}_0 = (0, \ldots, 0)$ and

  $$\vec{h}_{tk} = \sigma\left(\sum_{k'=1}^{M} U_{kk'} x_{tk'} + \sum_{k'=1}^{H} W_{kk'} h_{(t-1)k'}\right)$$

- Output sequence: $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$

  $$\vec{y}_t = (y_{t1}, \ldots, y_{tN})$$

  where $y_{tk} = \sigma\left(\sum_{k'=1}^{H} V_{kk'} h_{tk'}\right)$.

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$
- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$ where

$$\vec{h}_0 = (0, \ldots, 0)$$

and

$$\vec{h}_t = \sigma(U\vec{x}_t + W\vec{h}_{t-1})$$

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$
- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$ where

$$\vec{h}_0 = (0, \ldots, 0)$$

and

$$\vec{h}_t = \sigma(U\vec{x}_t + W\vec{h}_{t-1})$$

- Output sequence: $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$ where

$$y_t = \sigma(Vh_t)$$

RNN – Comments

- $\vec{h}_t$ is the memory of the network, captures what happened in all previous steps (with decaying quality).
- RNN **shares weights** U, V, W along the sequence.
  Note the similarity to convolutional networks where the weights were shared spatially over images, here they are shared temporally over sequences.
- RNN can deal with **sequences of variable length**.
  Compare with MLP which accepts only fixed-dimension vectors on input.

RNN – training

Training set

$$\mathcal{T} = \left\{(\mathbf{x}_1, \mathbf{d}_1), \ldots, (\mathbf{x}_p, \mathbf{y}_p)\right\}$$

here

- each $\mathbf{x}_\ell = \vec{x}_{\ell 1}, \ldots, \vec{x}_{\ell T_\ell}$ is an input sequence,
- each $\mathbf{d}_\ell = \vec{d}_{\ell 1}, \ldots, \vec{d}_{\ell T_\ell}$ is an expected output sequence.

Here each $\vec{x}_{\ell t} = (x_{\ell t 1}, \ldots, x_{\ell t M})$ is an input vector and each $\vec{d}_{\ell t} = (d_{\ell t 1}, \ldots, d_{\ell t N})$ is an expected output vector.

Error function

In what follows I will consider a training set with a **single element** $(\mathbf{x}, \mathbf{d})$. I.e. drop the index ℓ and have

- $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$ where $\vec{x}_t = (x_{t1}, \ldots, x_{tM})$
- $\mathbf{d} = \vec{d}_1, \ldots, \vec{d}_T$ where $\vec{d}_t = (d_{t1}, \ldots, d_{tN})$

The squared error of $(\mathbf{x}, \mathbf{d})$ is defined by

$$E_{(\mathbf{x},\mathbf{d})} = \sum_{t=1}^{T} \sum_{k=1}^{N} \frac{1}{2}(y_{tk} - d_{tk})^2$$

Recall that we have a sequence of network outputs $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$ and thus y_{tk} is the k-th component of $\vec{y}_t$

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.
- In the step $\ell + 1$ (here $\ell = 0, 1, 2, \ldots$) compute "new" weights $U^{(\ell+1)}, V^{(\ell+1)}, W^{(\ell+1)}$ from the "old" weights $U^{(\ell)}, V^{(\ell)}, W^{(\ell)}$ as follows:

$$U_{kk'}^{(\ell+1)} = U_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}}$$

$$V_{kk'}^{(\ell+1)} = V_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}}$$

$$W_{kk'}^{(\ell+1)} = W_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}}$$

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.
- In the step $\ell + 1$ (here $\ell = 0, 1, 2, \ldots$) compute "new" weights $U^{(\ell+1)}, V^{(\ell+1)}, W^{(\ell+1)}$ from the "old" weights $U^{(\ell)}, V^{(\ell)}, W^{(\ell)}$ as follows:

$$U_{kk'}^{(\ell+1)} = U_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}}$$

$$V_{kk'}^{(\ell+1)} = V_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}}$$

$$W_{kk'}^{(\ell+1)} = W_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}}$$

The above is THE learning algorithm that modifies weights!

Backpropagation

Computes the derivatives of E, no weights are modified!

Backpropagation

Computes the derivatives of E, no weights are modified!

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot x_{tk'} \qquad k' = 1, \ldots, M$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} \cdot \sigma' \cdot h_{tk'} \qquad k' = 1, \ldots, H$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot h_{(t-1)k'} \qquad k' = 1, \ldots, H$$

Backpropagation

Computes the derivatives of E, no weights are modified!

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot x_{tk'} \qquad k' = 1, \ldots, M$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} \cdot \sigma' \cdot h_{tk'} \qquad k' = 1, \ldots, H$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot h_{(t-1)k'} \qquad k' = 1, \ldots, H$$

Backpropagation:

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} = y_{tk} - d_{tk} \qquad \text{(assuming squared error)}$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} = \sum_{k'=1}^{N} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk'}} \cdot \sigma' \cdot V_{k'k} + \sum_{k'=1}^{H} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{(t+1)k'}} \cdot \sigma' \cdot W_{k'k}$$

Long-term dependencies

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} = \sum_{k'=1}^{N} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk'}} \cdot \sigma' \cdot V_{k'k} + \sum_{k'=1}^{H} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{(t+1)k'}} \cdot \sigma' \cdot W_{k'k}$$

- Unless $\sum_{k'=1}^{H} \sigma' \cdot W_{k'k} \approx 1$, the gradient either vanishes, or explodes.
- For a large T (long-term dependency), the gradient "deeper" in the past tends to be too small (large).
- A solution: LSTM

LSTM

$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t) \qquad \text{output}$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t \qquad \text{memory}$$

$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t) \qquad \text{new memory contents}$$

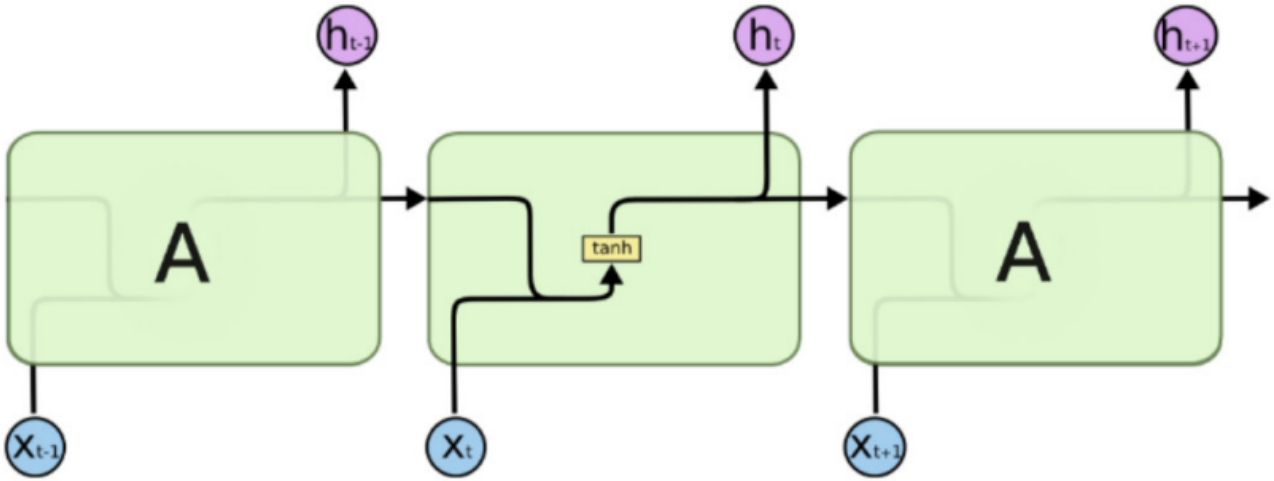
$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t) \qquad \text{output gate}$$

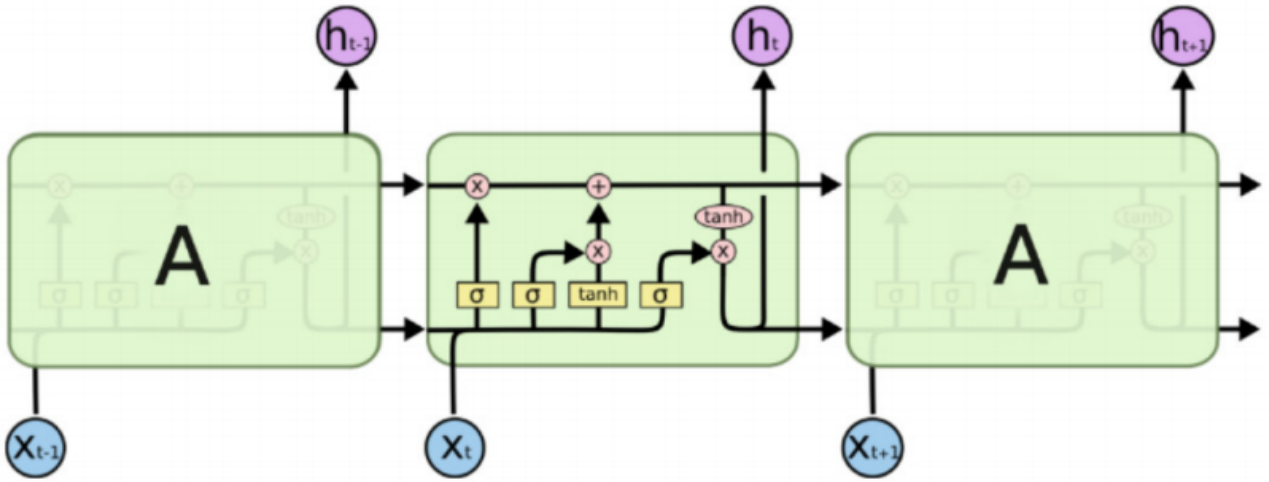
$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t) \qquad \text{forget gate}$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t) \qquad \text{input gate}$$

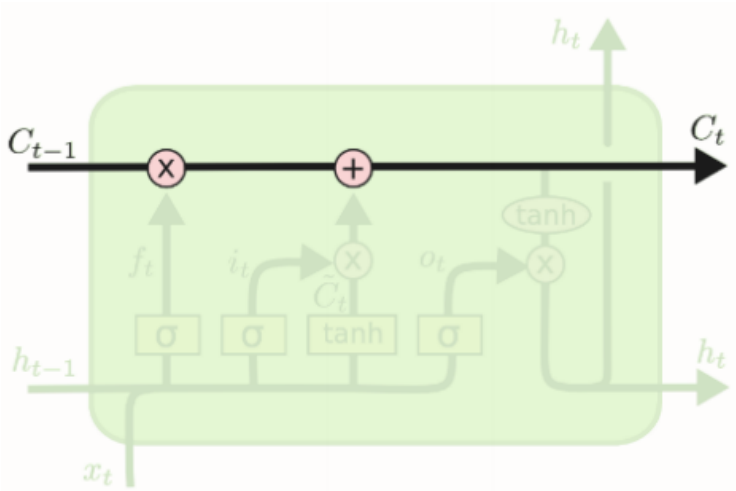
- $\circ$ is the component-wise product of vectors
- $\cdot$ is the matrix-vector product
- σ_h hyperbolic tangents (applied component-wise)
- σ_g logistic sigmoid (aplied component-wise)

RNN vs LSTM


h_{t-1}
h_t
h_{t+1}
A
tanh
A
x_{t-1}
x_t
x_{t+1}


h_{t-1}
h_t
h_{t+1}
A
σ
σ
tanh
σ
tanh
A
x_{t-1}
x_t
x_{t+1}

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

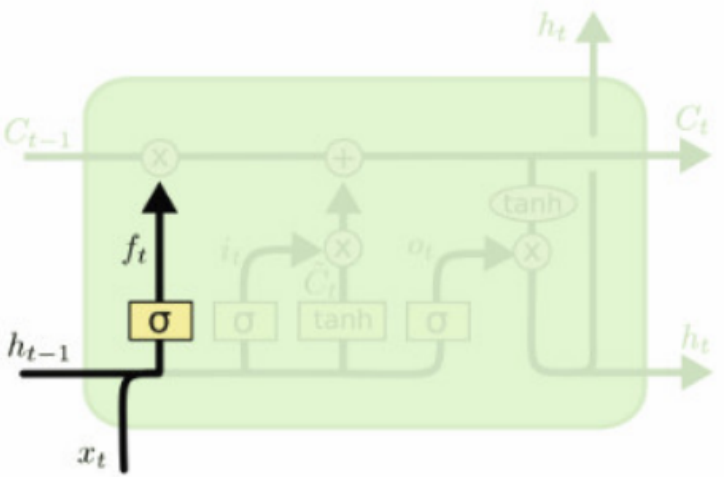
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

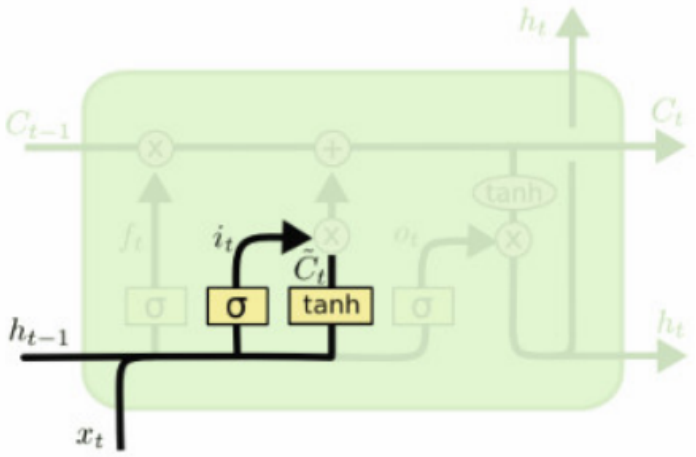
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

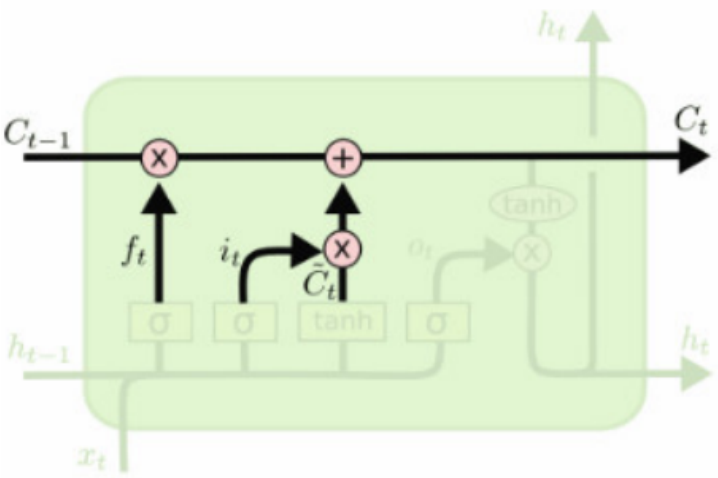
$$\Rightarrow \tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

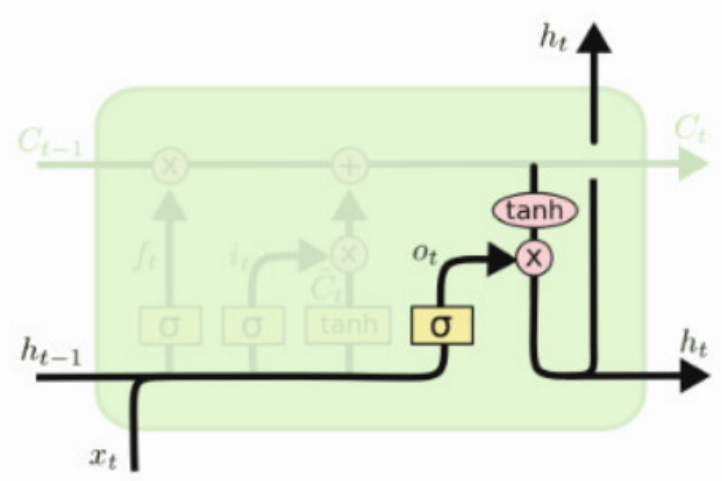
$$\Rightarrow \tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

LSTM



$$\Rightarrow \vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\Rightarrow \vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

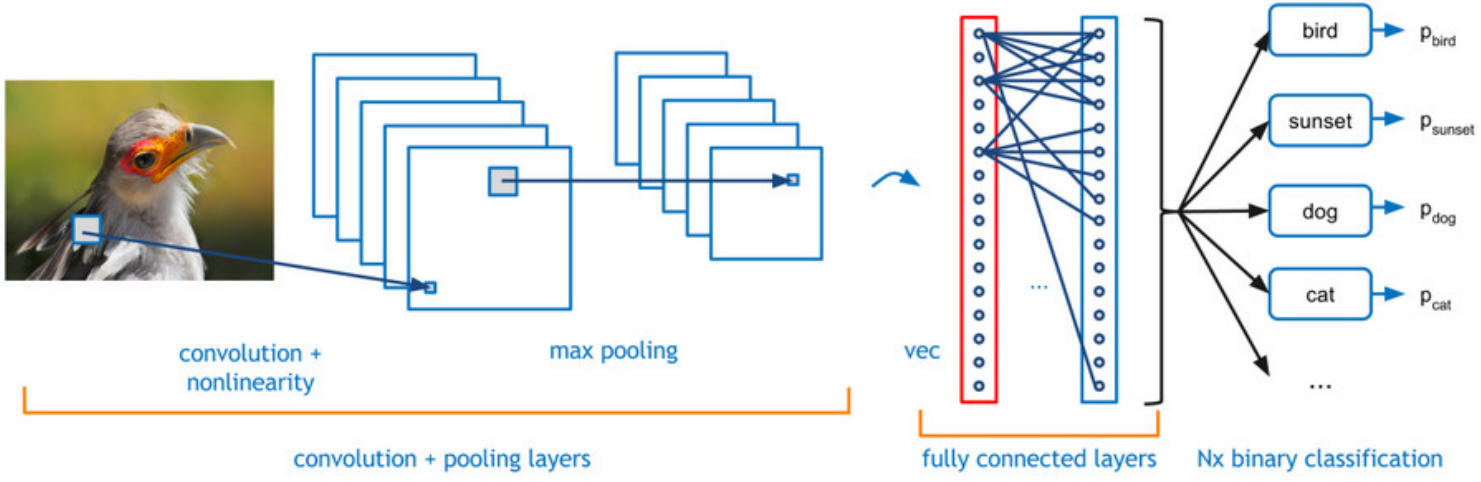
$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

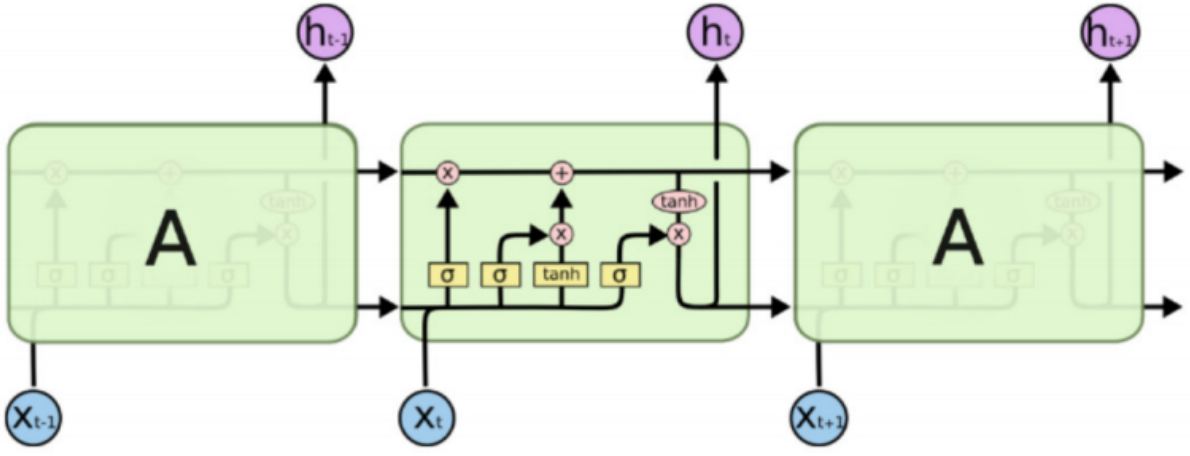
$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

LSTM – summary

- LSTM (almost) solves the vanishing gradient problem w.r.t. the "internal" state of the network.
- Learns to control its own memory (via forget gate).
- Revolution in machine translation and text processing.

Convolutions & LSTM in action – cancer research





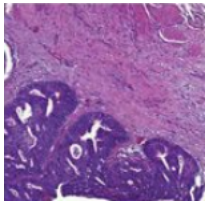
Colorectal cancer outcome prediction

The problem: Predict 5-year survival probability from an image of a small region of tumour tissue (1 mm diameter).

Colorectal cancer outcome prediction

The problem: Predict 5-year survival probability from an image of a small region of tumour tissue (1 mm diameter).

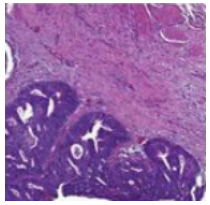
Input: Digitized haematoxylin-eosin-stained tumour tissue microarray samples.
Output: Estimated survival probability.



Colorectal cancer outcome prediction

The problem: Predict 5-year survival probability from an image of a small region of tumour tissue (1 mm diameter).

Input: Digitized haematoxylin-eosin-stained tumour tissue microarray samples.
Output: Estimated survival probability.



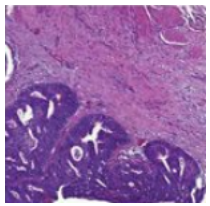
Data:

- Training set: 420 patients of Helsinki University Centre Hospital, diagnosed with colorectal cancer, underwent primary surgery.
- Test set: 182 patients
- Follow-up time and outcome known for each patient.

Colorectal cancer outcome prediction

The problem: Predict 5-year survival probability from an image of a small region of tumour tissue (1 mm diameter).

Input: Digitized haematoxylin-eosin-stained tumour tissue microarray samples.
Output: Estimated survival probability.



Data:

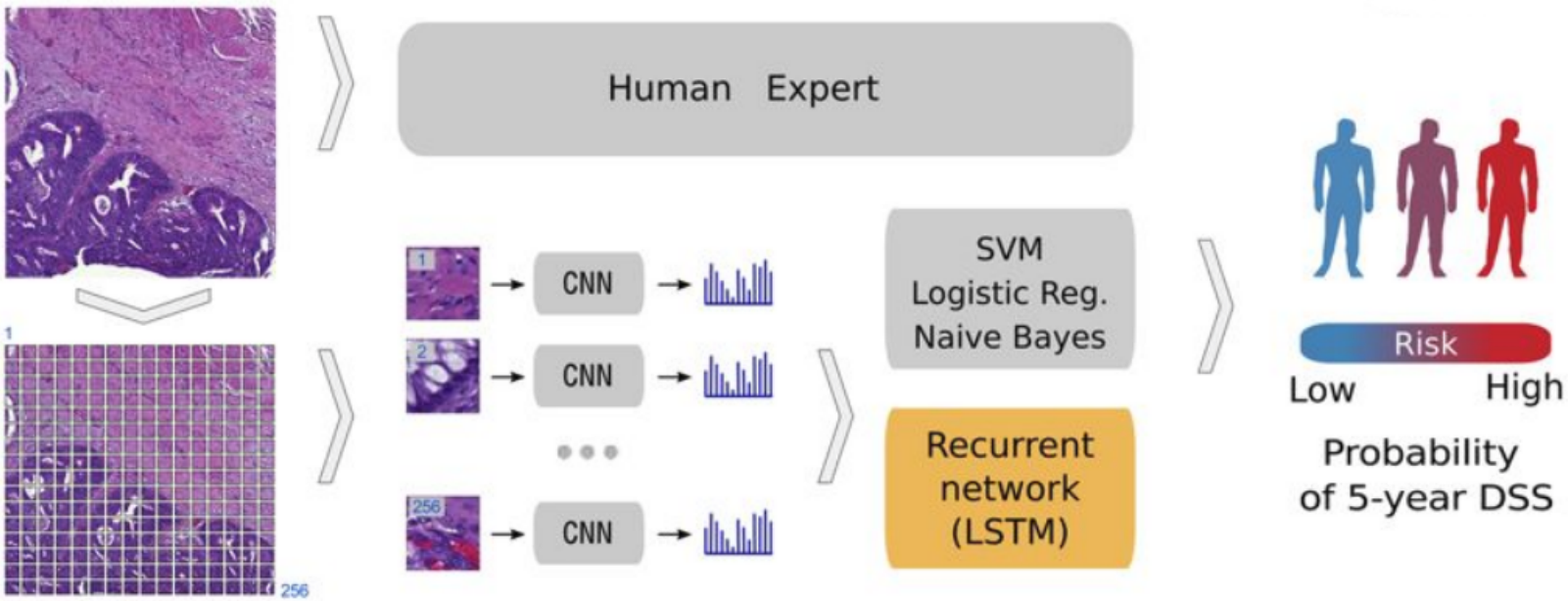
- Training set: 420 patients of Helsinki University Centre Hospital, diagnosed with colorectal cancer, underwent primary surgery.
- Test set: 182 patients
- Follow-up time and outcome known for each patient.

Human expert comparison:

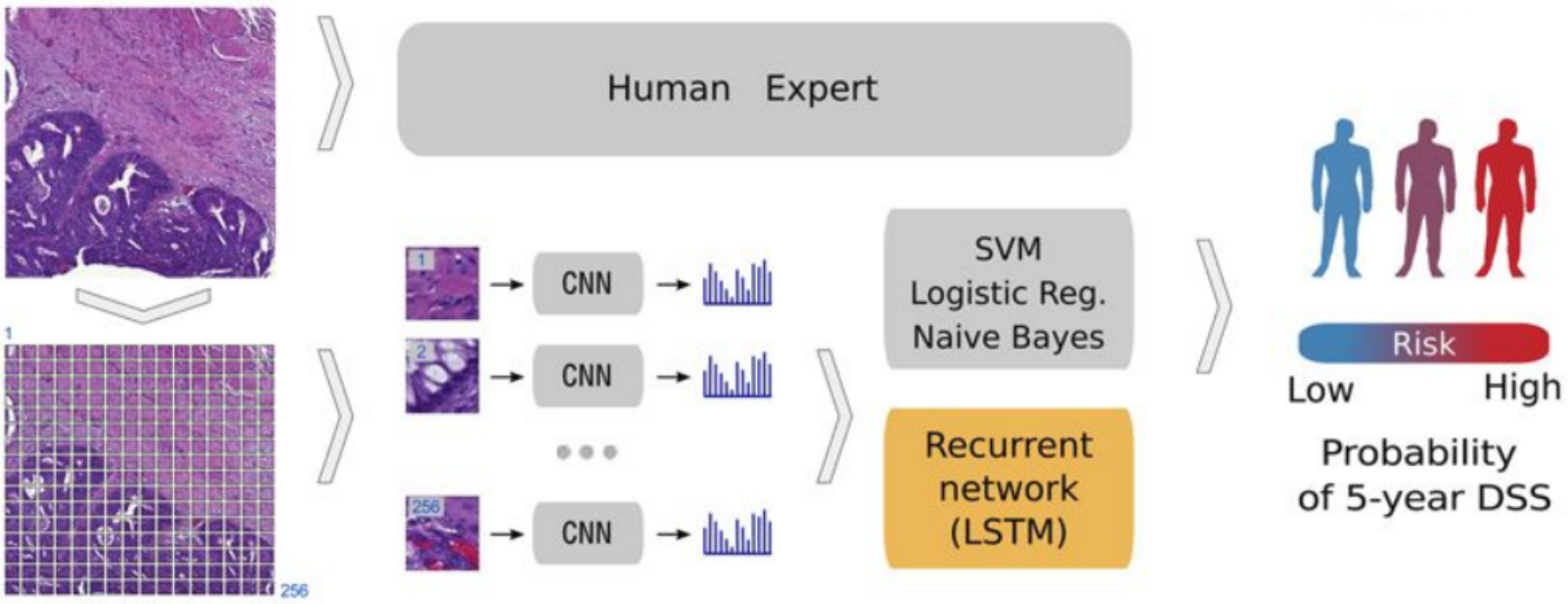
- Histological grade assessed at the time of diagnosis.
- Visual Risk Score: Three pathologists classified to high/low-risk categories (by majority vote).

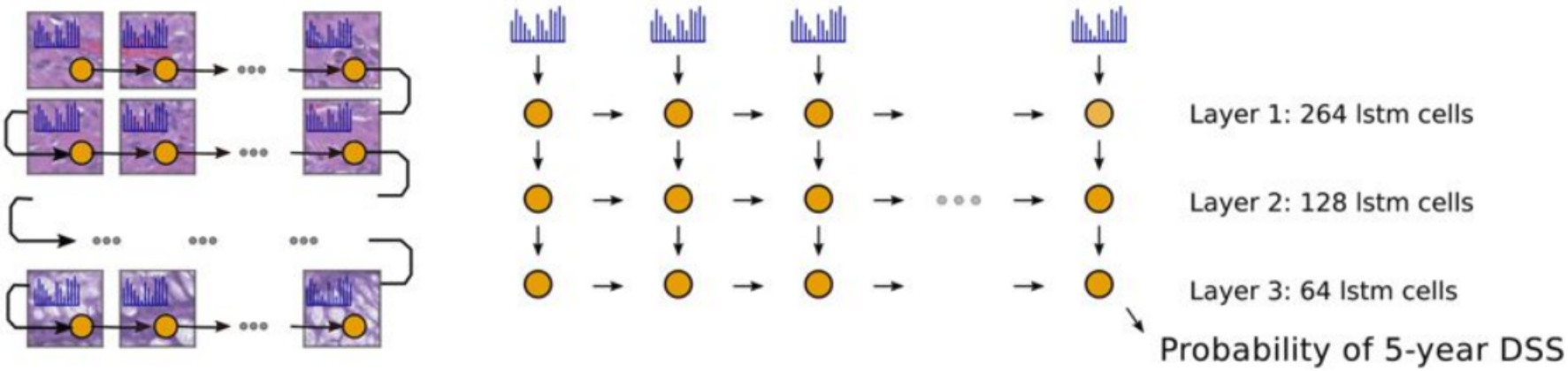
Source: D. Bychkov et al. Deep learning based tissue analysis predicts outcome in colorectal cancer. Scientific Reports, Nature, 2018.

Colorectal cancer outcome prediction



Colorectal cancer outcome prediction





Data & workflow

- Input images: 3500 px $\times$ 3500 px
  - Cut into tiles: 224 px $\times$ 224 px $\Rightarrow$ 256 tiles
- Each tile pased to a convolutional network (CNN)
  - Ouptut of CNN: 4096 dimensional vector.
- A "string" of 256 vectors (each of the dimension 4096) pased into a LSTM.
- LSTM outputs the probability of 5-year survival.

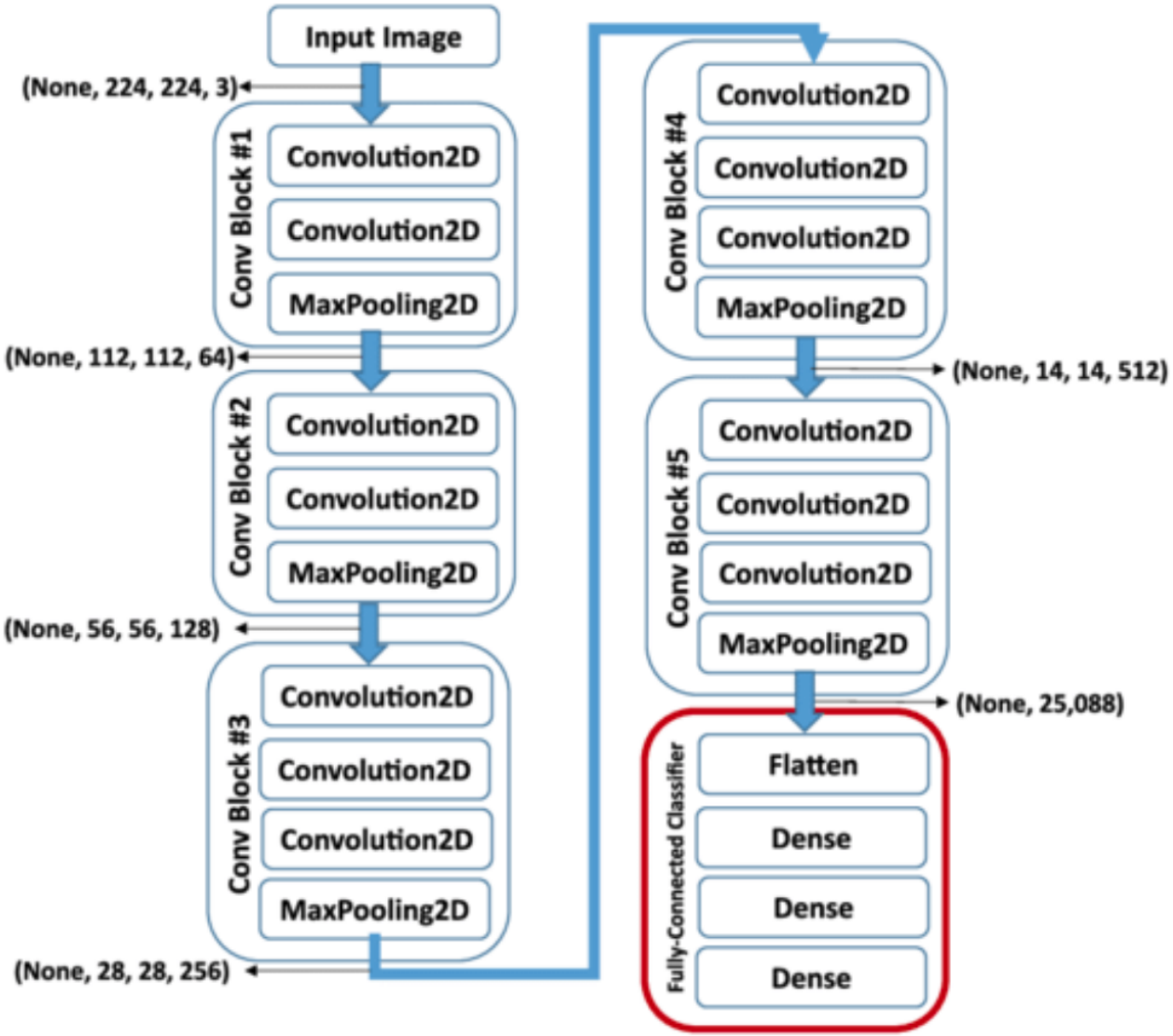
Data & workflow

- Input images: 3500 px $\times$ 3500 px
  - Cut into tiles: 224 px $\times$ 224 px $\Rightarrow$ 256 tiles
- Each tile pased to a convolutional network (CNN)
  - Ouptut of CNN: 4096 dimensional vector.
- A "string" of 256 vectors (each of the dimension 4096) pased into a LSTM.
- LSTM outputs the probability of 5-year survival.

The authors also tried to substitute the LSTM on top of CNN with

- logistic regression
- naive Bayes
- support vector machines

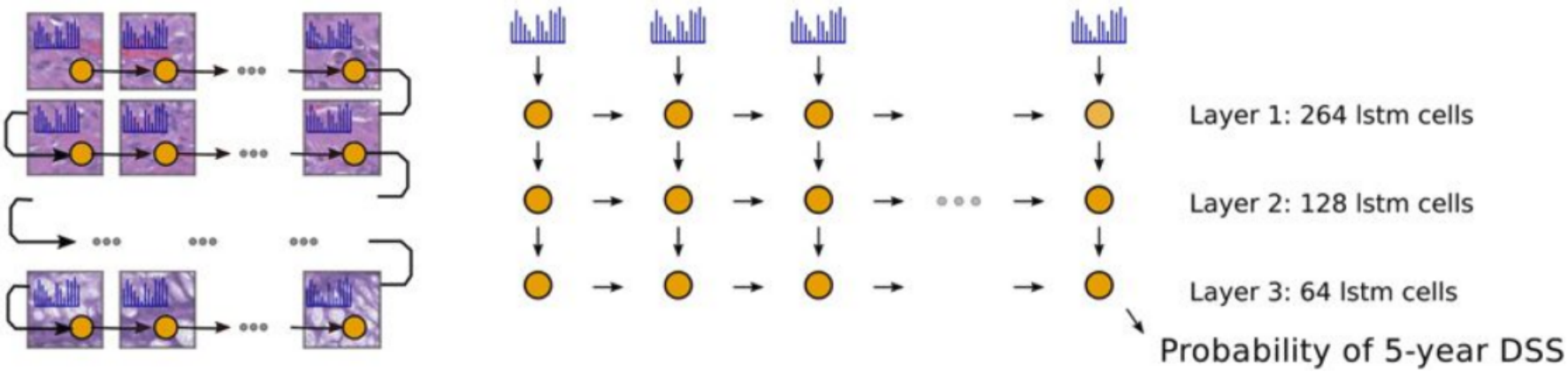
CNN architecture – VGG-16



(Pre)trained on ImageNet (cats, dogs, chairs, etc.)

LSTM architecture

- LSTM has three layers (264, 128, 64 cells)

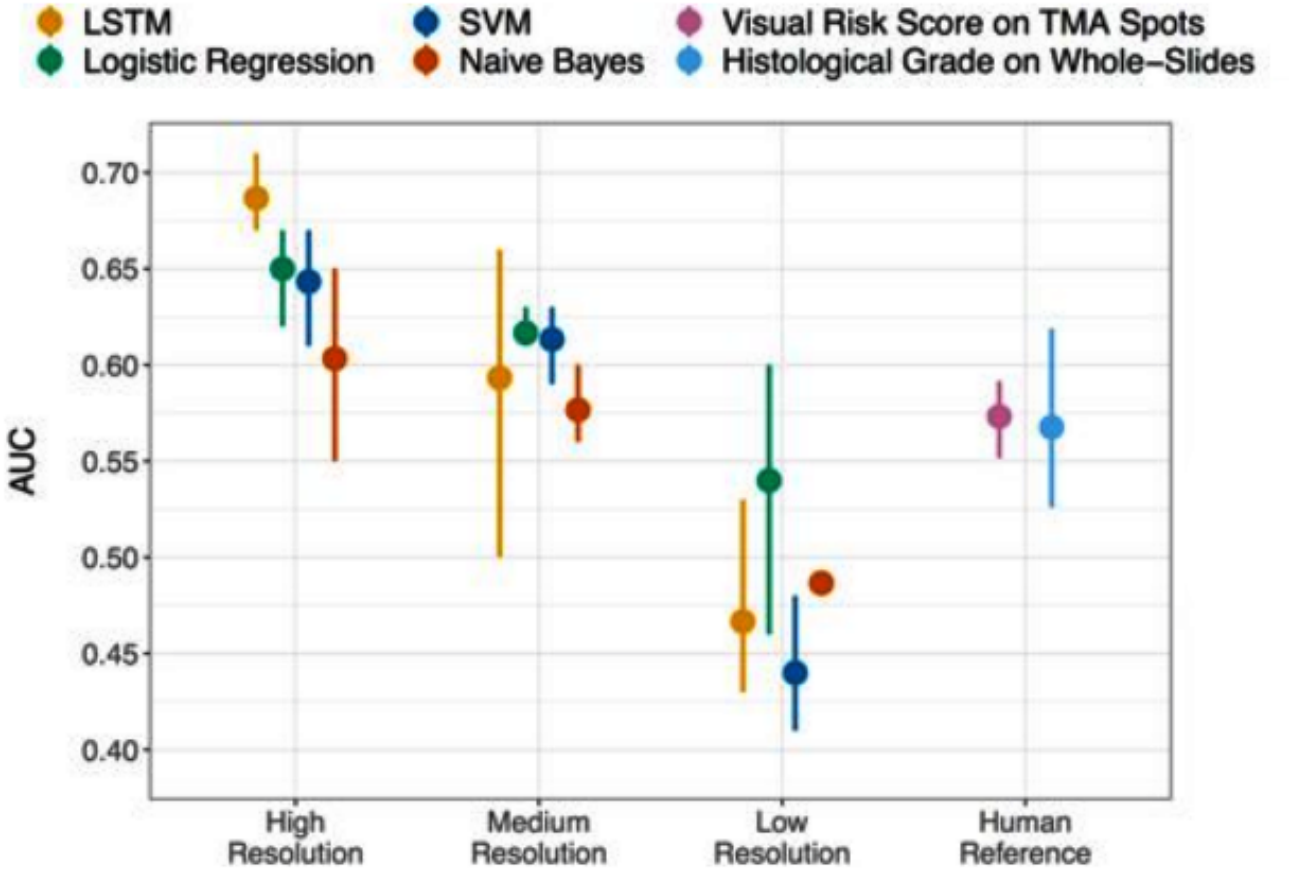


LSTM – training

- L1 regularization (0.005) at each hidden layer of LSTM
  i.e. 0.005 times the sum of absolute values of weights added to the error
- L2 regularization (0.005) at each hidden layer of LSTM
  i.e. 0.005 times the sum of squared values of weights added to the error
- Dropout 5% at the input and the last hidden layers of LSTM

- Datasets:
  - Training: 220 samples,
  - Validation 60 samples,
  - Test 140 samples.

Colorectal cancer outcome prediction



Source: D. Bychkov et al. Deep learning based tissue analysis predicts outcome in colorectal cancer. Scientific Reports, Nature, 2018.

Feed-forward networks summary

Architectures:

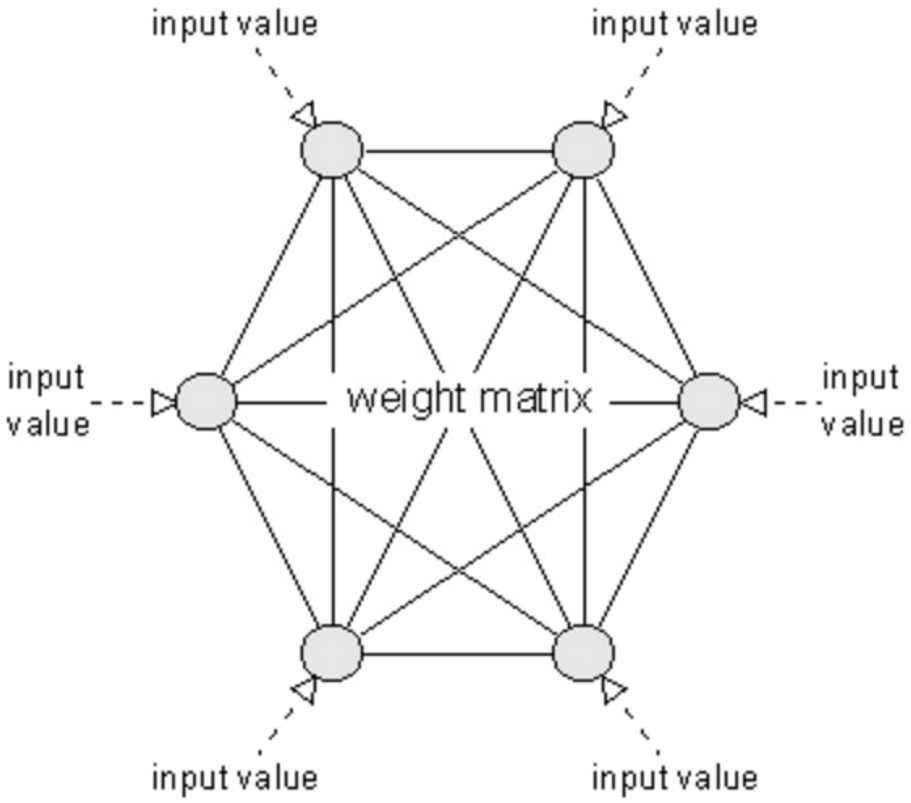
- Multi-layer perceptron (MLP):
  - dense connections between layers
- Convolutional networks (CNN):
  - local receptors, feature maps
  - pooling
- Recurrent networks (RNN, LSTM):
  - self-loops but still feed-forward through time

Training:

- gradient descent algorithm + heuristics

Hopfield Network

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 $j \in \{1, \ldots, n\}$
  We assume $w_{ji} = w_{ij}$, i.e. symmetric connections.
- assume $w_{jj} = 0$ for every $j = 1, \ldots, n$
- **For now:** no neuron has a bias

Hopfield network

Learning: Training set

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

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:

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.

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

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.

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).

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.

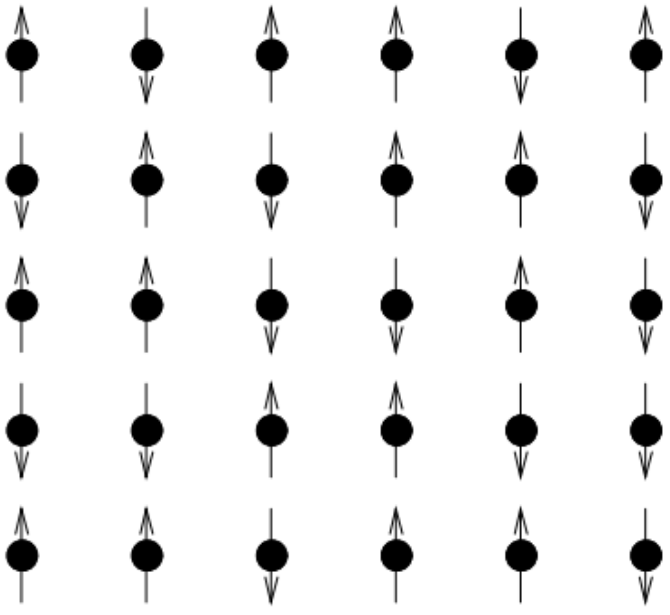
If W is clear from the context, we write only $y(\vec{x})$ a $y_j(\vec{x})$.

Ising model – an analogy

Simple models of magnetic materials resemble Hopfield network.

Ising model – an analogy

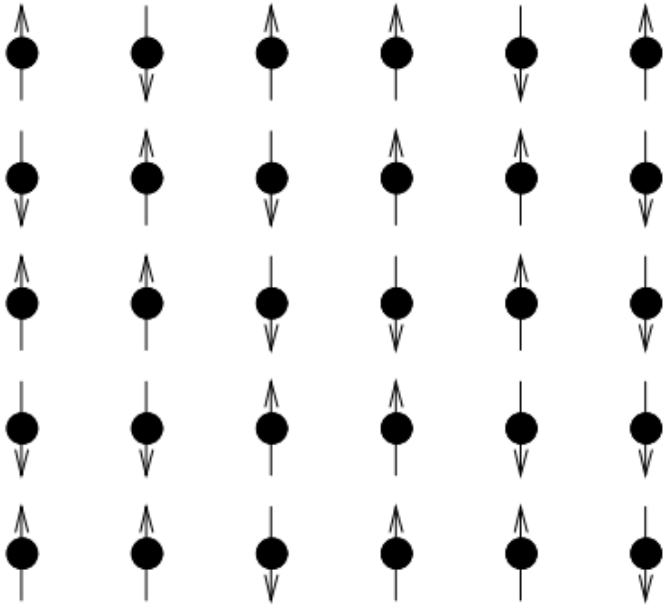
Simple models of magnetic materials resemble Hopfield network.



- atomic magnets organized into square-lattice

Ising model – an analogy

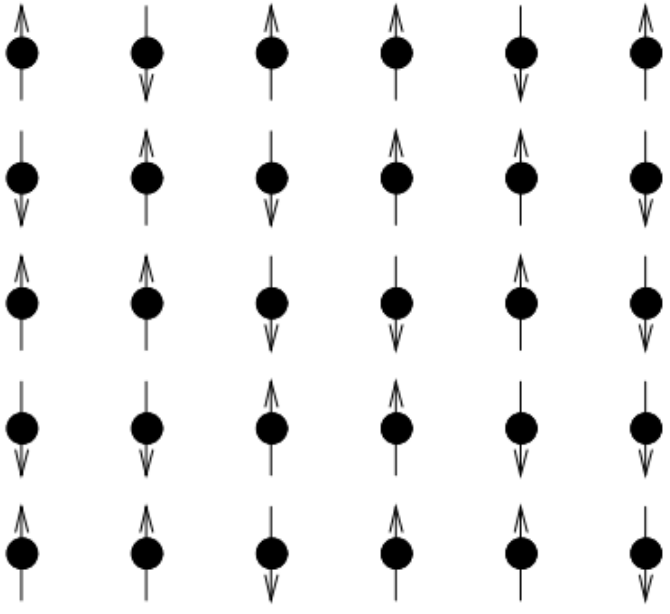
Simple models of magnetic materials resemble Hopfield network.



- atomic magnets organized into square-lattice
- each magnet may have only one of two possible orientations (in the Hopfield network $+1$ a -1)

Ising model – an analogy

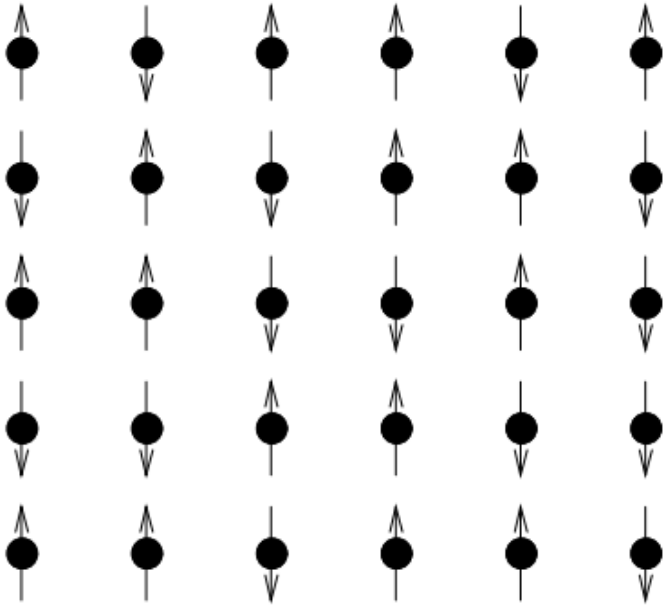
Simple models of magnetic materials resemble Hopfield network.



- 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

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:

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

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

Energy function

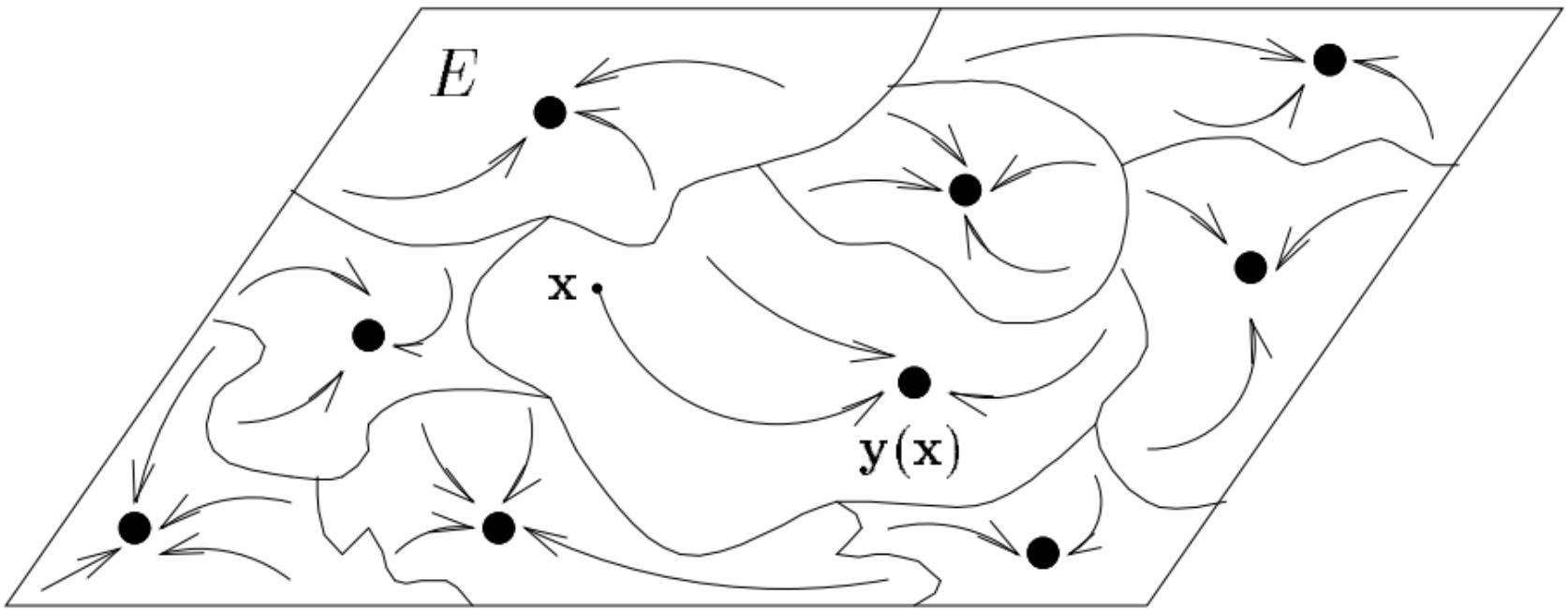
Energy function E assigns to every state $\vec{y} \in \{-1, 1\}^n$ a (potential) energy:

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

- 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 network – convergence

Observe that

- the energy does not increase during computation: $E(\vec{y}^{(t)}) \geq E(\vec{y}^{(t+1)})$

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 $E(\vec{y}^{(t)}) > E(\vec{y}^{(t+1)})$

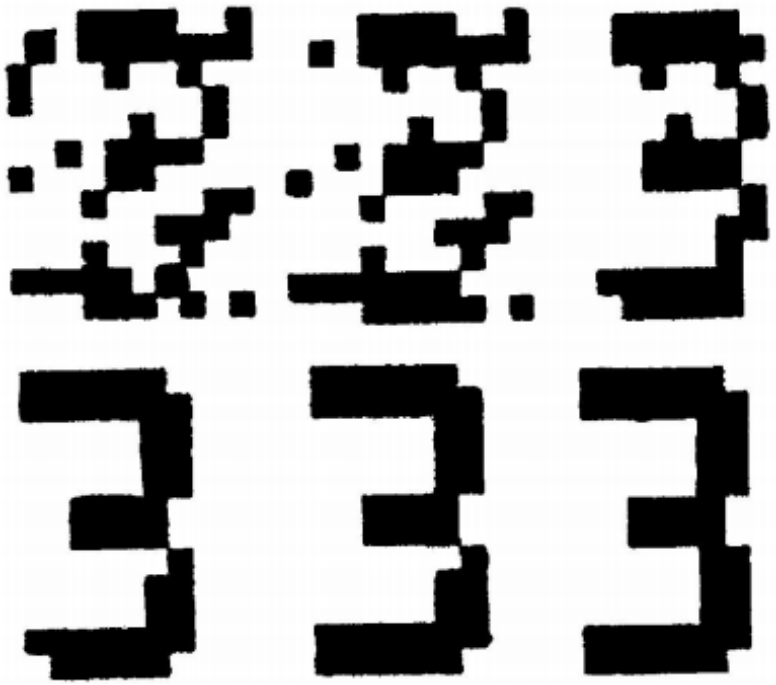
Hopfield network – convergence

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.

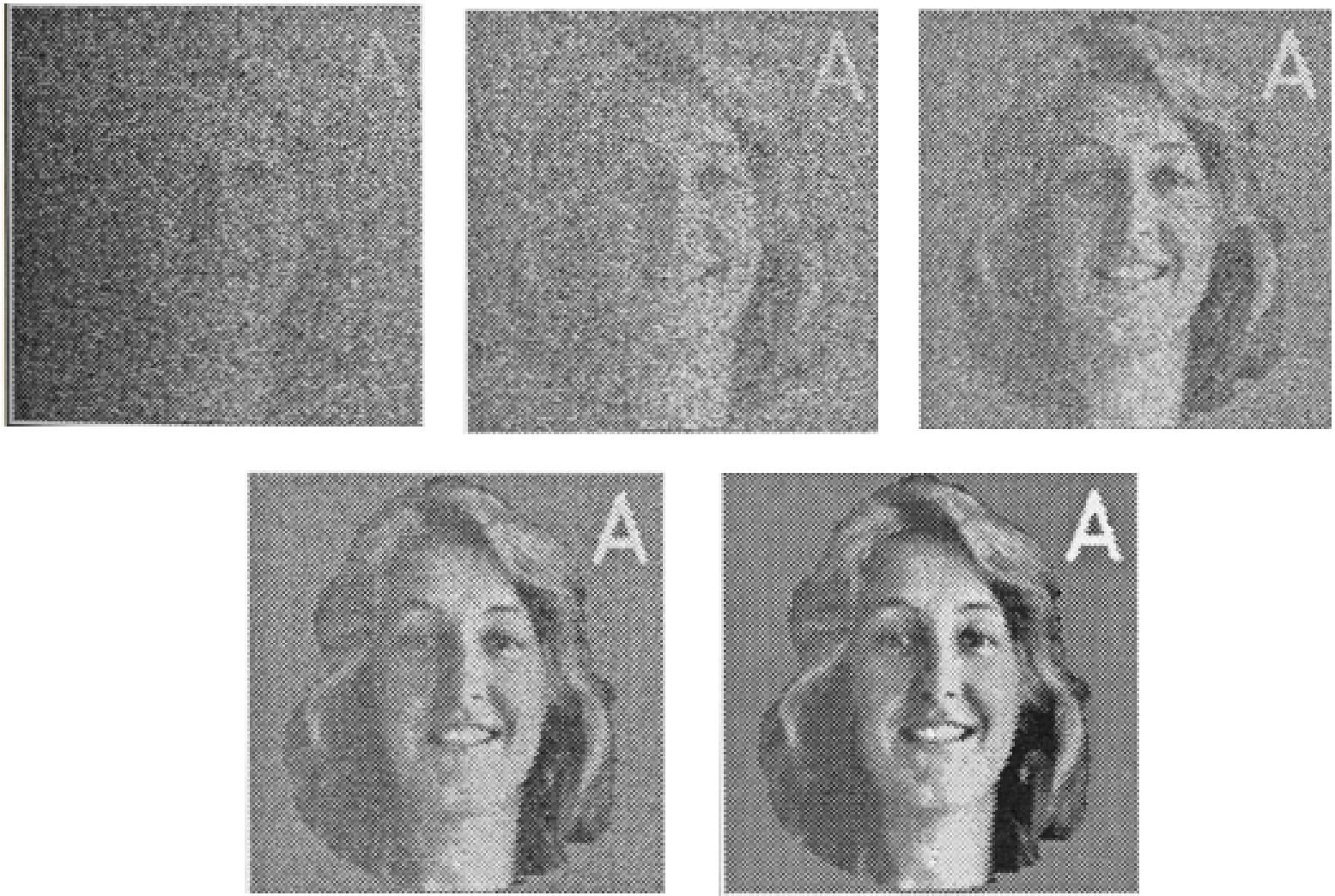
This proves that computation of a Hopfield network always stops.

Hopfield network – example

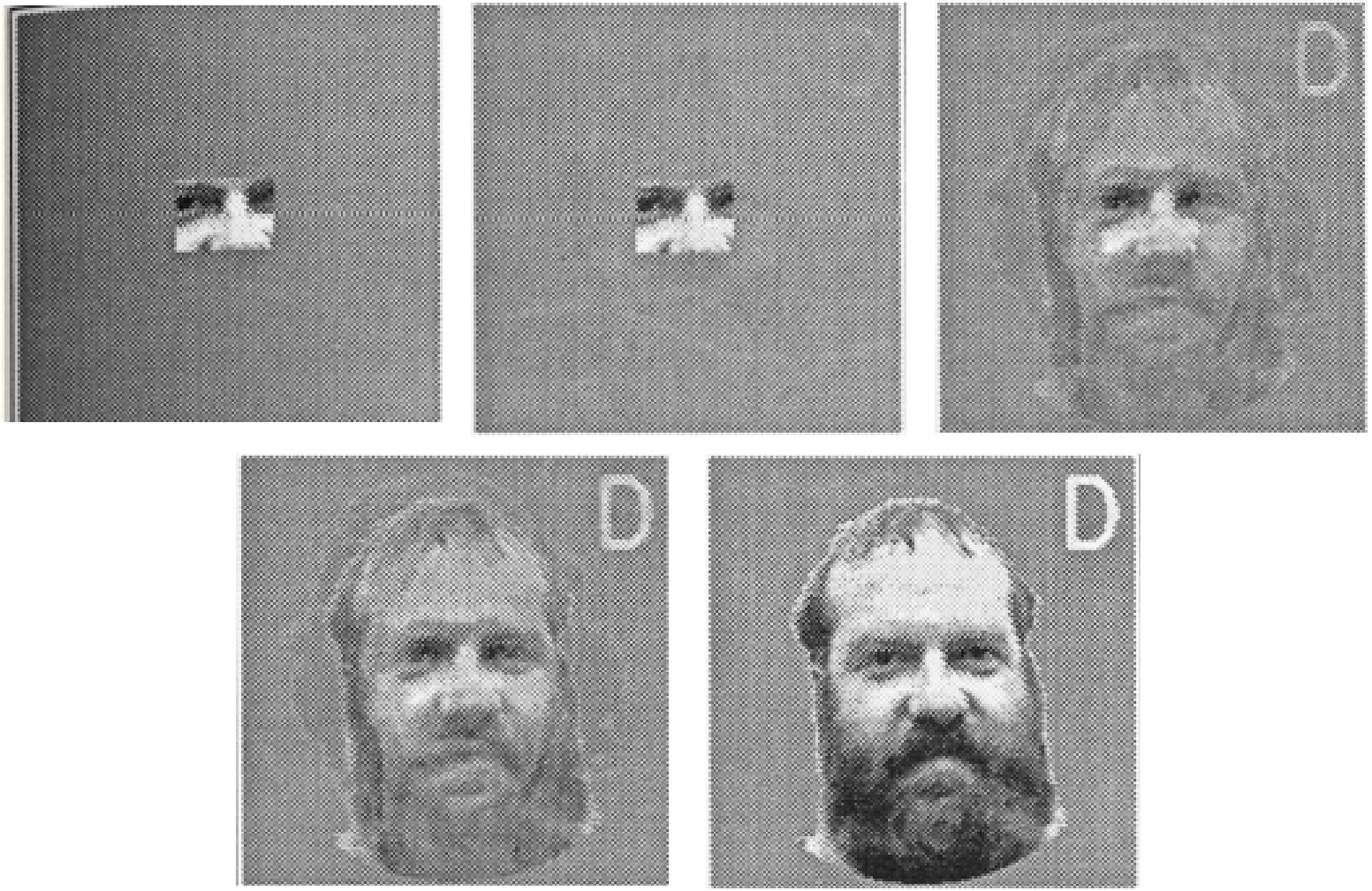


- figures 12×10
  (120 neurons, -1 is white and 1 is black)
- learned 8 figures
- input generated with 25% noise
- image shows the activity of the Hopfield network

Hopfield network – example



Hopfield network – example



Restricted Boltzmann Machines

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 $j \in N$.

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 $j \in V$

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

Equilibrium

Theorem

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

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

where

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

and

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

Here y_i^γ is the value of the neuron i in the state γ.

RBM – Probability distribution

RBM defines the following probability distribution on $\{0,1\}^{|N|}$ (recall that N is the set of all neurons):

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

We obtain a distribution on states of visible neurons by marginalization:

$$p_V(\alpha) = \sum_{\beta\in\{0,1\}^{|H|}} p_N(\alpha\beta) \quad \text{for every } \alpha \in \{0,1\}^{|V|}$$

Here $\alpha\beta \in \{0,1\}^{|N|}$ is a vector of values of all states obtained by concatenating values α of visible neurons and values β of hidden neurons.

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):

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

$\mathcal{E}$ is minimized using the gradient descent algorithm.

RBM – learning

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

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

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:

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

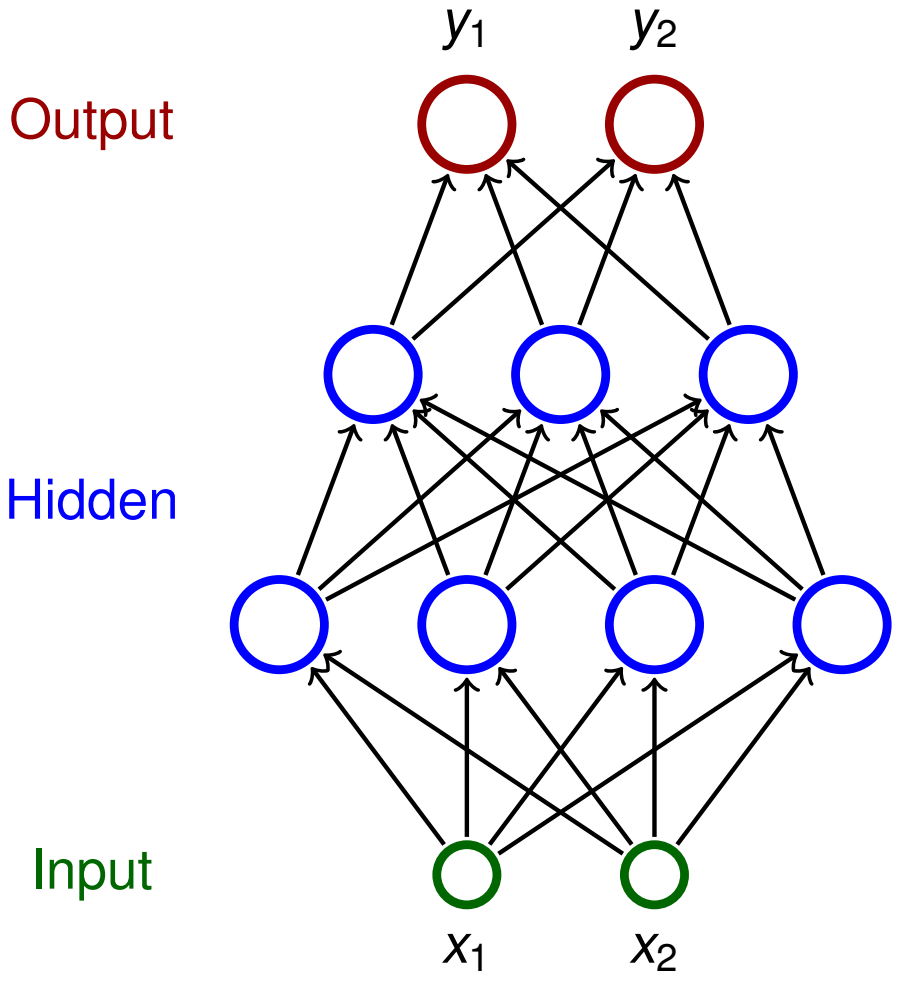
where

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

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 $\frac{\partial \mathcal{E}}{\partial w_{ji}}(W)$ (skipped).

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)

Why deep networks

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

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.

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 – pretraining

Assume k layers. Denote

- W_i the weight matrix between layers $i - 1$ and i

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).

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.

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.

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.

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

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).

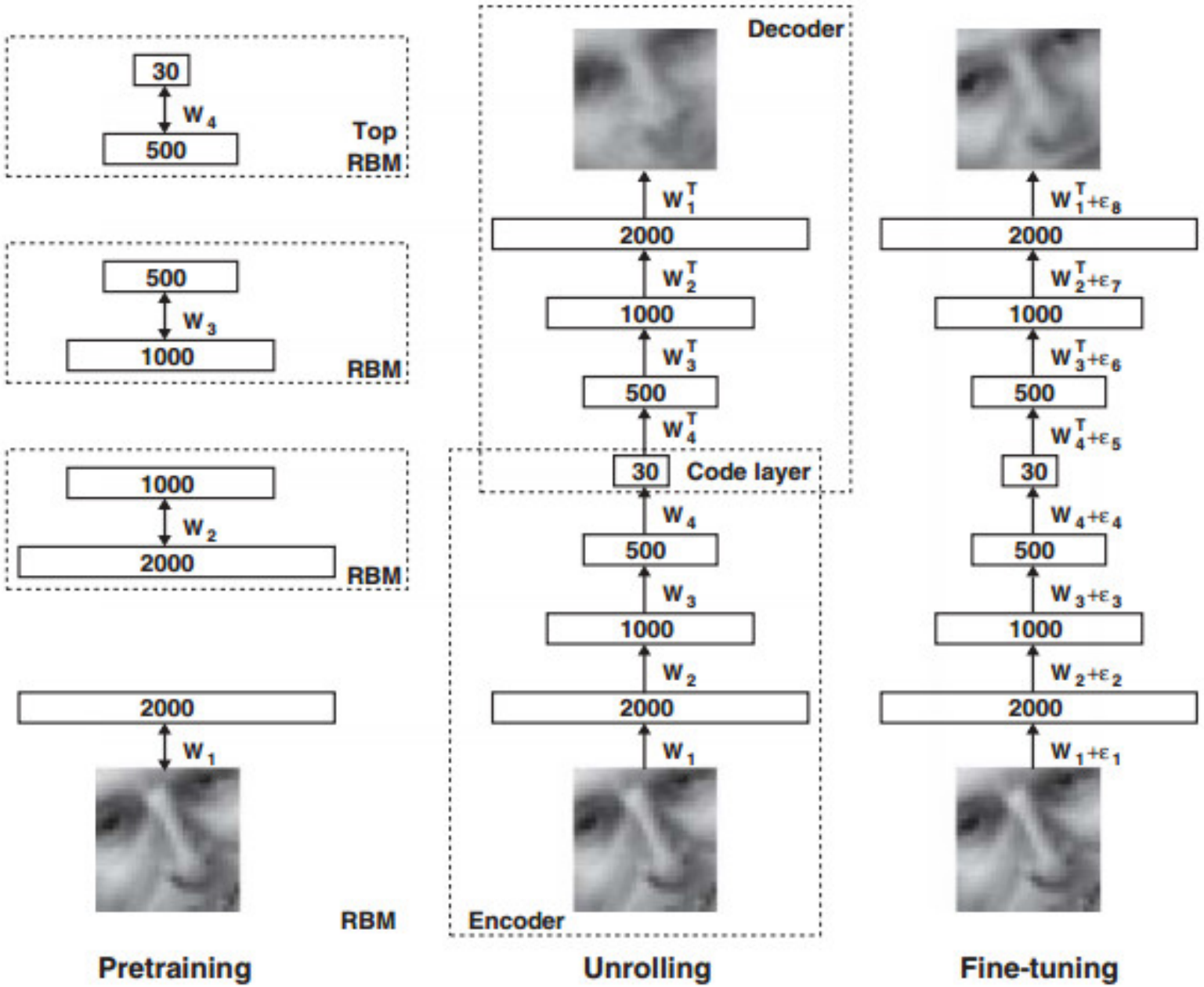
Deep MLP – dimensionality reduction

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 $[0, 1]$.

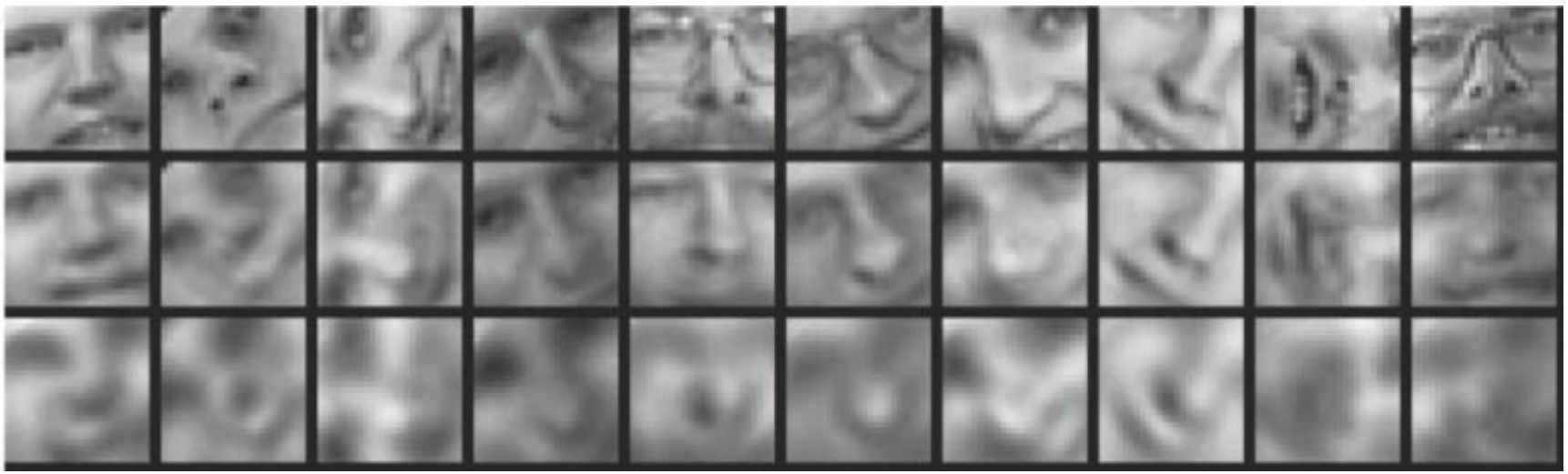
Images – fine-tuning

- 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)

Kohonen's Map

Vector quantization

- Assume we are given a probability density function $p(\vec{x})$ on input vectors $\vec{x} \in \mathbb{R}^n$.

  I.e. assume that the inputs are randomly generated according to $p(\vec{x})$.
- Our goal is to approximate $p(\vec{x})$ using finitely many **centres** $\vec{w}_i \in \mathbb{R}^n$ where $i = 1, \ldots, h$.

  Roughly speaking: We want more centres in areas of higher density and less in areas of low density.

Vector quantization

- Assume we are given a probability density function $p(\vec{x})$ on input vectors $\vec{x} \in \mathbb{R}^n$.

  I.e. assume that the inputs are randomly generated according to $p(\vec{x})$.
- Our goal is to approximate $p(\vec{x})$ using finitely many **centres** $\vec{w}_i \in \mathbb{R}^n$ where $i = 1, \ldots, h$.

  Roughly speaking: We want more centres in areas of higher density and less in areas of low density.
- Formally: To every input $\vec{x}$ we assign its *closest* centre $\vec{w}_{c(\vec{x})}$:

$$c(\vec{x}) = \arg \min_{i=1,\ldots,h} \left\{ \left\| \vec{x} - \vec{w}_i \right\| \right\}$$

and then minimize the error

$$E = \int \left\| \vec{x} - \vec{w}_{c(\vec{x})} \right\|^2 p(\vec{x}) d\vec{x}$$

Caution! $c(\vec{x})$ depends on $\vec{x}$.

Vector quantization

In practice, $p(\vec{x})$ is obtained by *sampling uniformly* from a given training (multi)set:

$$\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$$

Vector quantization

In practice, $p(\vec{x})$ is obtained by *sampling uniformly* from a given training (multi)set:

$$\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$$

The error then corresponds to

$$E = \frac{1}{\ell} \sum_{j=1}^{\ell} \left\| \vec{x}_j - \vec{w}_{c(\vec{x}_j)} \right\|^2$$

(keep in mind that $c(\vec{x}_j) = \arg\min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i \right\| \right\}$.)

Vector quantization

In practice, $p(\vec{x})$ is obtained by *sampling uniformly* from a given training (multi)set:

$$\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$$

The error then corresponds to

$$E = \frac{1}{\ell} \sum_{j=1}^{\ell} \left\| \vec{x}_j - \vec{w}_{c(\vec{x}_j)} \right\|^2$$

(keep in mind that $c(\vec{x}_j) = \arg\min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i \right\| \right\}$.)

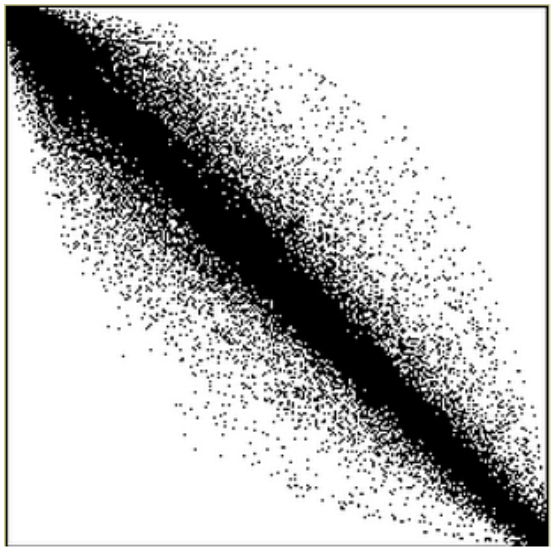
If $\mathcal{T}$ has been randomly selected according to $p(\vec{x})$ and ℓ is large eough, then

$$\frac{1}{\ell} \sum_{j=1}^{\ell} \left\| \vec{x}_j - \vec{w}_{c(\vec{x}_j)} \right\|^2 \approx \int \left\| \vec{x} - \vec{w}_{c(\vec{x})} \right\|^2 p(\vec{x}) d\vec{x}$$

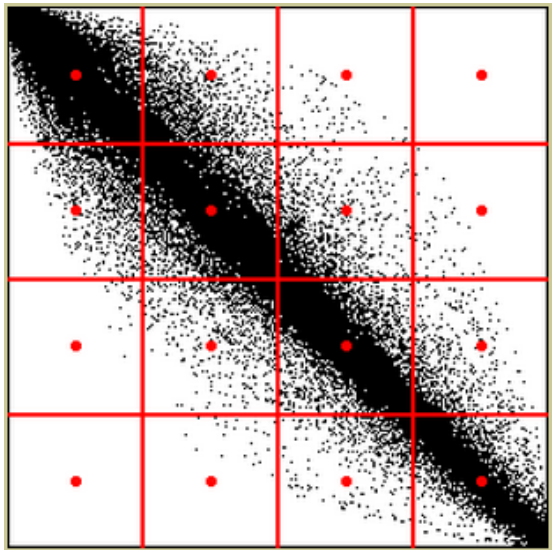
Example – image compression

- Every pixel has 256 shades of grey,
- each pair of neighbouring pixels is a two-dimensional vector from $\{0,\ldots,255\} \times \{0,\ldots,255\}$,
- our compression finds a small set of centres that will encode shades of grey of *pairs of pixels*,
- image is then encoded by simple substitution of pairs of pixels with their centres.

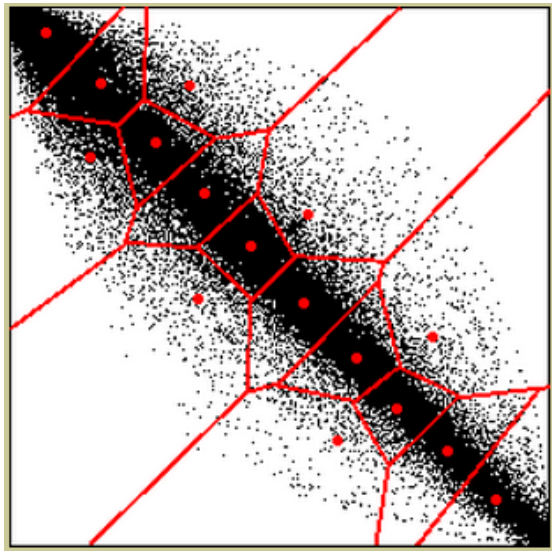
Example – image compression



pair distribution



naive quantization



smart quantization

k-means clustering algorithm

Assume a finite training set: $\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$

The algorithm moves centres closer to the centres of mass of closest points.

k-means clustering algorithm

Assume a finite training set: $\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$

The algorithm moves centres closer to the centres of mass of closest points.

In the step t computes $\vec{w}_1^{(t)}, \ldots, \vec{w}_h^{(t)}$ as follows:

k-means clustering algorithm

Assume a finite training set: $\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$

The algorithm moves centres closer to the centres of mass of closest points.

In the step t computes $\vec{w}_1^{(t)}, \ldots, \vec{w}_h^{(t)}$ as follows:

- for every $k = 1, \ldots, h$ compute a set $\mathcal{T}_k$ of all vectors of $\mathcal{T}$ to which $\vec{w}_k^{(t-1)}$ is the closest centre:

$$\mathcal{T}_k = \left\{ \vec{x}_j \in \mathcal{T} \quad \middle| \quad k = \arg \min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i^{(t-1)} \right\| \right\} \right\}$$

k-means clustering algorithm

Assume a finite training set: $\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$

The algorithm moves centres closer to the centres of mass of closest points.

In the step t computes $\vec{w}_1^{(t)}, \ldots, \vec{w}_h^{(t)}$ as follows:

- for every $k = 1, \ldots, h$ compute a set $\mathcal{T}_k$ of all vectors of $\mathcal{T}$ to which $\vec{w}_k^{(t-1)}$ is the closest centre:

$$\mathcal{T}_k = \left\{ \vec{x}_j \in \mathcal{T} \quad \middle| \quad k = \arg \min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i^{(t-1)} \right\| \right\} \right\}$$

- compute $\vec{w}_k^{(t)}$ to be the centroid of $\mathcal{T}_k$:

$$\vec{w}_k^{(t)} = \frac{1}{|\mathcal{T}_k|} \sum_{\vec{x} \in \mathcal{T}_k} \vec{x}$$

We may stop the computation when, e.g. the error E is sufficiently small.

Kohonen's learning

The k-means algorithm is not online.

Kohonen's learning

The k-means algorithm is not online.

The following Kohonen's algorithm is online (i.e. the inputs may be generated one by one and the centres are adapted online):

Kohonen's learning

The k-means algorithm is not online.

The following Kohonen's algorithm is online (i.e. the inputs may be generated one by one and the centres are adapted online):

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

Kohonen's learning

The k-means algorithm is not online.

The following Kohonen's algorithm is online (i.e. the inputs may be generated one by one and the centres are adapted online):

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

If $\vec{w}_k^{(t-1)}$ is the closest centre to $\vec{x}_t$, i.e.
$k = \arg\min_i \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ **then**

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \theta \cdot (\vec{x}_t - \vec{w}_k^{(t-1)})$$

else $\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)}$

Kohonen's learning

The k-means algorithm is not online.

The following Kohonen's algorithm is online (i.e. the inputs may be generated one by one and the centres are adapted online):

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

If $\vec{w}_k^{(t-1)}$ is the closest centre to $\vec{x}_t$, i.e.
$k = \arg\min_i \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ **then**

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \theta \cdot (\vec{x}_t - \vec{w}_k^{(t-1)})$$

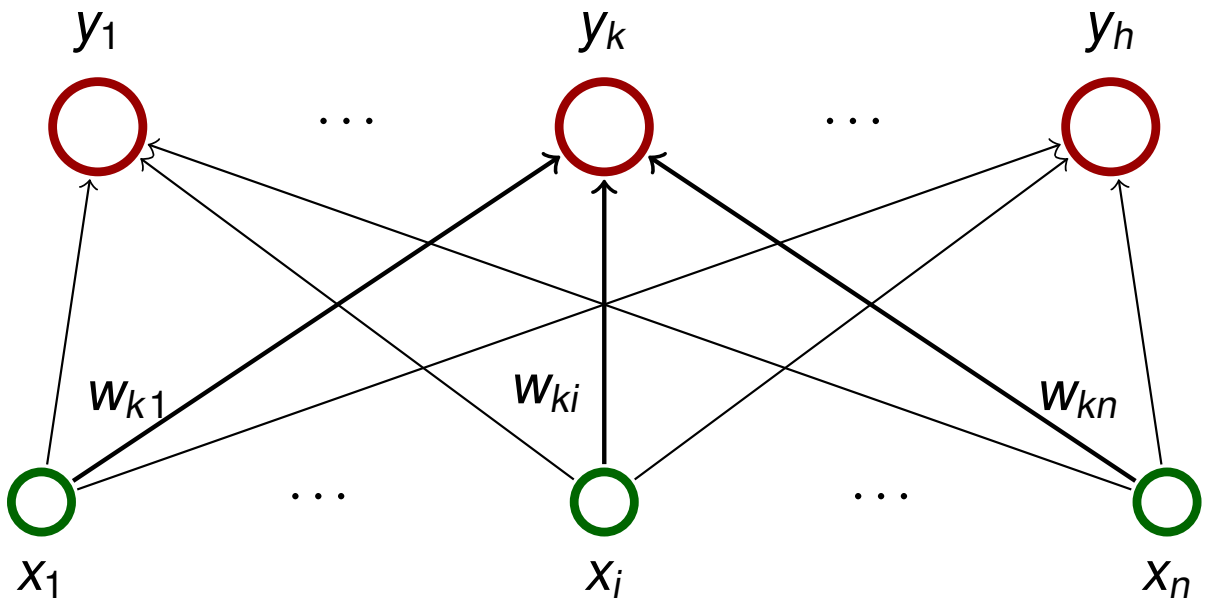
else $\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)}$

$0 < \theta \leq 1$ determines how much to move the centre towards the input.

Let us formulate this algorithm in the language of neural networks.

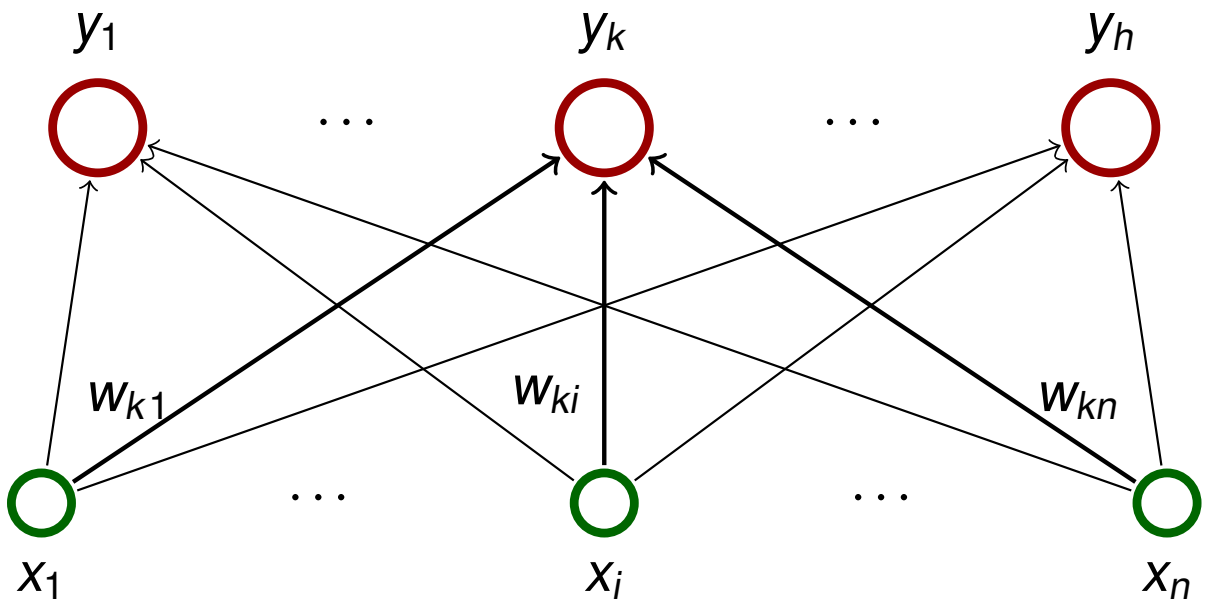
Kohonen's learning – neural network

Architecture: Single layer



Kohonen's learning – neural network

Architecture: Single layer



Activity: For an input $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{otherwise} \end{cases}$$

Kohonen's learning

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

Kohonen's learning

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

If $\vec{w}_k^{(t-1)}$ is the closest neuron to $\vec{x}_t$, i.e.
$k = \arg\min_i \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ **then**

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \theta \cdot (\vec{x}_t - \vec{w}_k^{(t-1)})$$

else $\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)}$

$0 < \theta \leq 1$ determines how much to move the neuron towards the input.

Kohonen's learning – efficiency

- Works well if most input vectors evenly distributed in a convex area.

Kohonen's learning – efficiency

- Works well if most input vectors evenly distributed in a convex area.
- In case of two (or more) separated clusters, the density may not correspond to $p(\vec{x})$ at all:
  - Ex. Two separated areas with the same density.
  - Assume that the centres are initially in one of the areas.

Kohonen's learning – efficiency

- Works well if most input vectors evenly distributed in a convex area.
- In case of two (or more) separated clusters, the density may not correspond to $p(\vec{x})$ at all:
  - Ex. Two separated areas with the same density.
  - Assume that the centres are initially in one of the areas.
  - The second then "drags" only one of the centres (which always wins the competition).

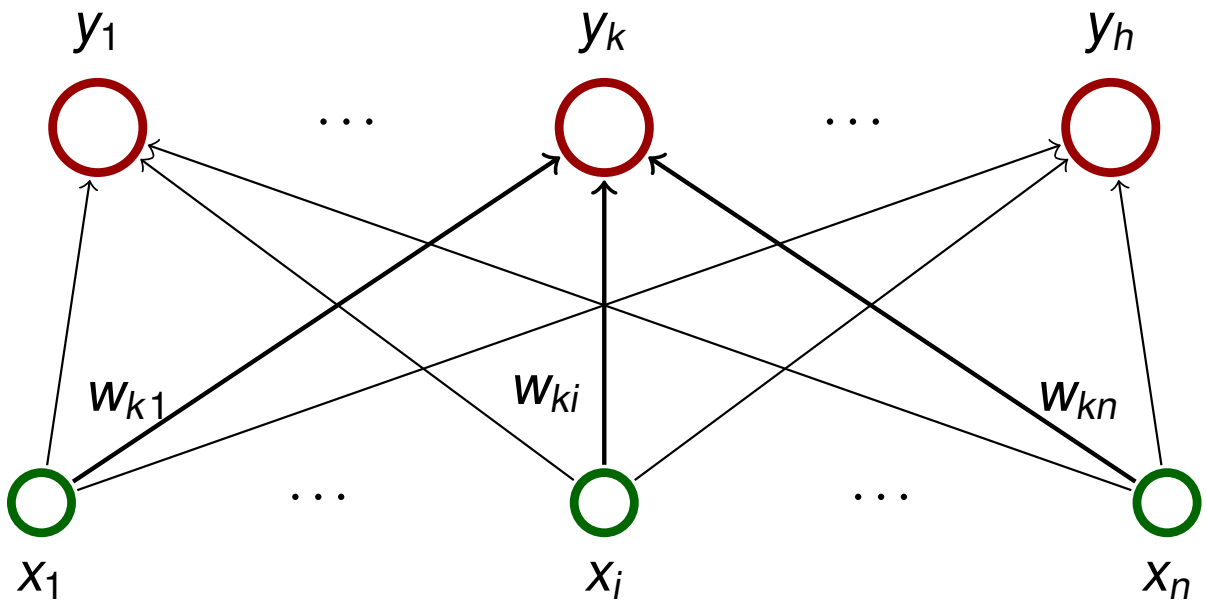
Kohonen's learning – efficiency

- Works well if most input vectors evenly distributed in a convex area.
- In case of two (or more) separated clusters, the density may not correspond to $p(\vec{x})$ at all:
  - Ex. Two separated areas with the same density.
  - Assume that the centres are initially in one of the areas.
  - The second then "drags" only one of the centres (which always wins the competition).
  - Result: One of the areas will be covered by a single centre even though it contains half of the mass of the input examples.

Solution: We tie centres together so that they have to move together.

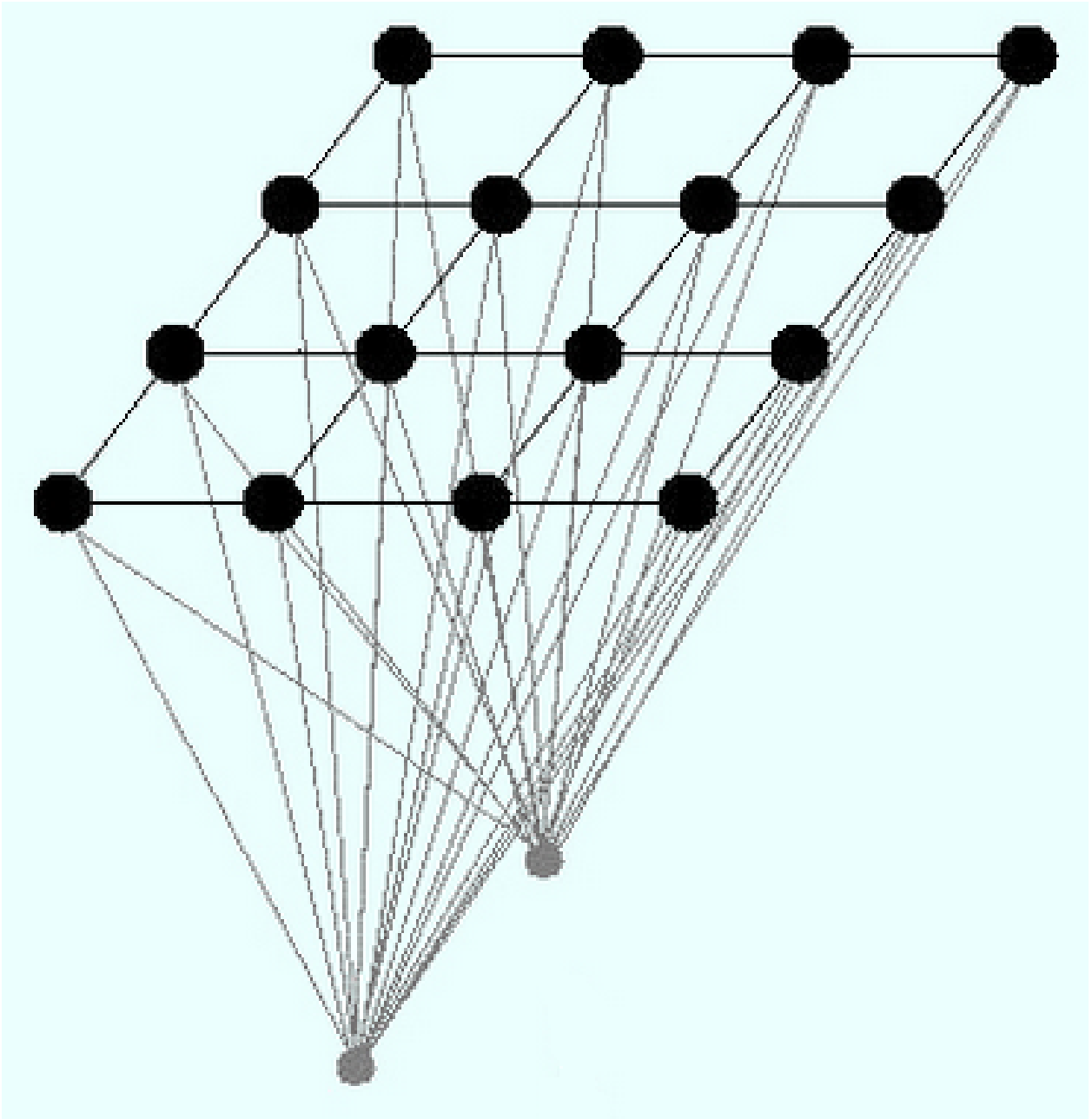
Kohonen's map

Architecture: Single layer



- **Topological structure**: neurons connected by edges so that they are nodes in an undirected graph.
- In most cases, this structure is either a one dimensional sequence or a two dimensional grid.

Kohonen's map – illustration



Kohonen's map – bio motivation

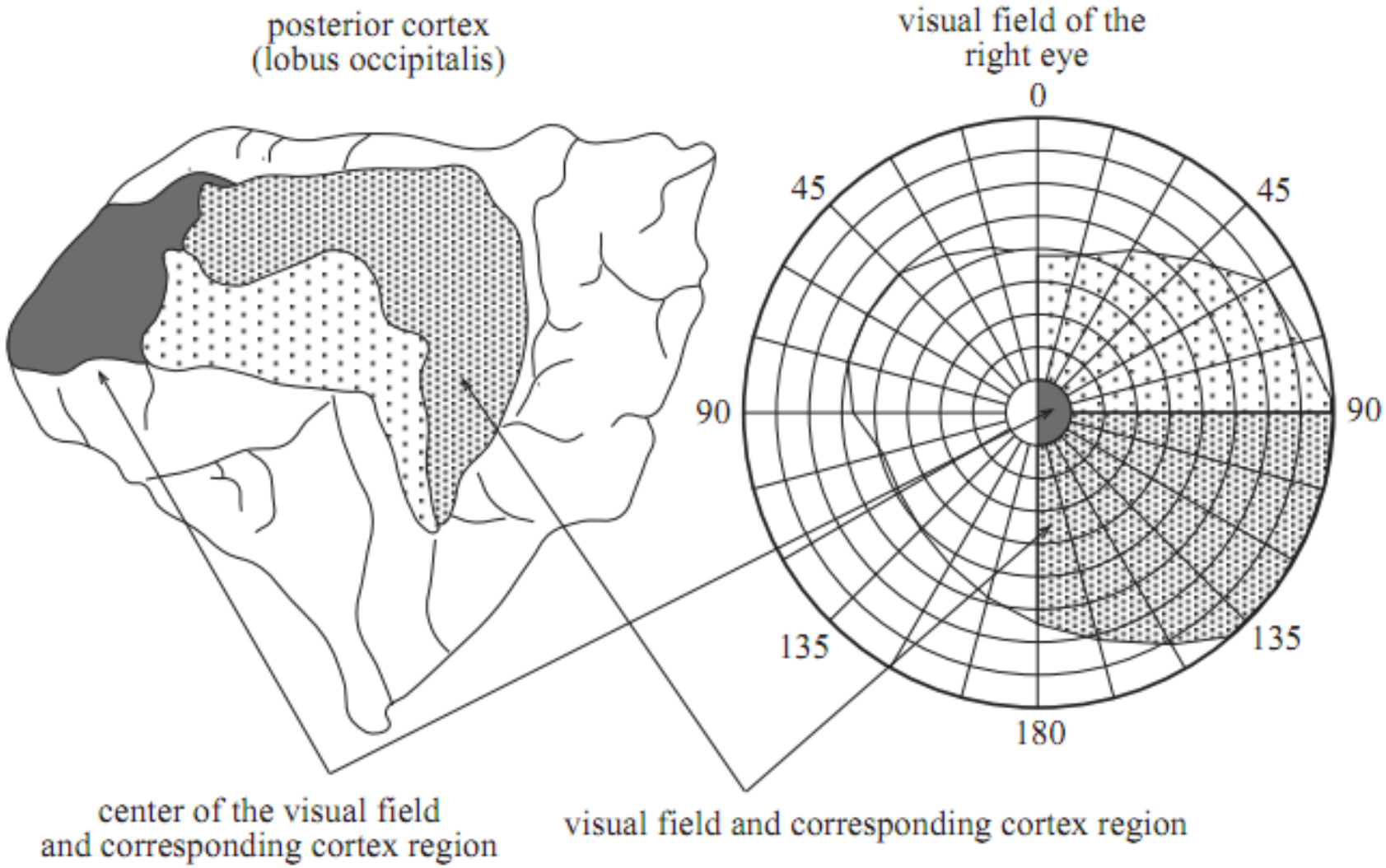


Fig. 15.2. Mapping of the visual field on the cortex

Source: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Kohonen's map

Activity: Given an input vector $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{jinak} \end{cases}$$

Kohonen's map

Activity: Given an input vector $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{jinak} \end{cases}$$

Learning: We use the topological structure.

- Denote by $d(c,k)$ the length of the shortest path from neuron c to neuron k in the *topological structure*.
- For every neuron c and a given $s \in \mathbb{N}_0$ define **topological neighbourhood** of the neuron c of size s : $N_s(c) = \{k \mid d(c,k) \leq s\}$

Kohonen's map

Activity: Given an input vector $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{jinak} \end{cases}$$

Learning: We use the topological structure.

- Denote by $d(c,k)$ the length of the shortest path from neuron c to neuron k in the *topological structure*.
- For every neuron c and a given $s \in \mathbb{N}_0$ define **topological neighbourhood** of the neuron c of size s : $N_s(c) = \{k \mid d(c,k) \leq s\}$

In step t, given training example $\vec{x}_t$ adapt $\vec{w}_k$ as follows:

$$\vec{w}_k^{(t)} = \begin{cases} \vec{w}_k^{(t-1)} + \theta \cdot \left(\vec{x}_t - \vec{w}_k^{(t-1)} \right) & k \in N_s(c(\vec{x}_t)) \\ \vec{w}_k^{(t-1)} & \text{otherwise} \end{cases}$$

where $c(\vec{x}_t) = \arg\min_{i=1,\ldots,h} \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ and $\theta \in \mathbb{R}$ and $s \in \mathbb{N}_0$ are parameters that may change during training.

Kohonen's map – learning

More general version:

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \Theta(c(\vec{x}_t), k) \cdot \left(\vec{x}_t - \vec{w}_k^{(t-1)}\right)$$

where $c(\vec{x}_t) = \arg\min_{i=1,\ldots,h} \left\|\vec{x}_t - \vec{w}_i^{(t-1)}\right\|$. The previous case then corresponds to

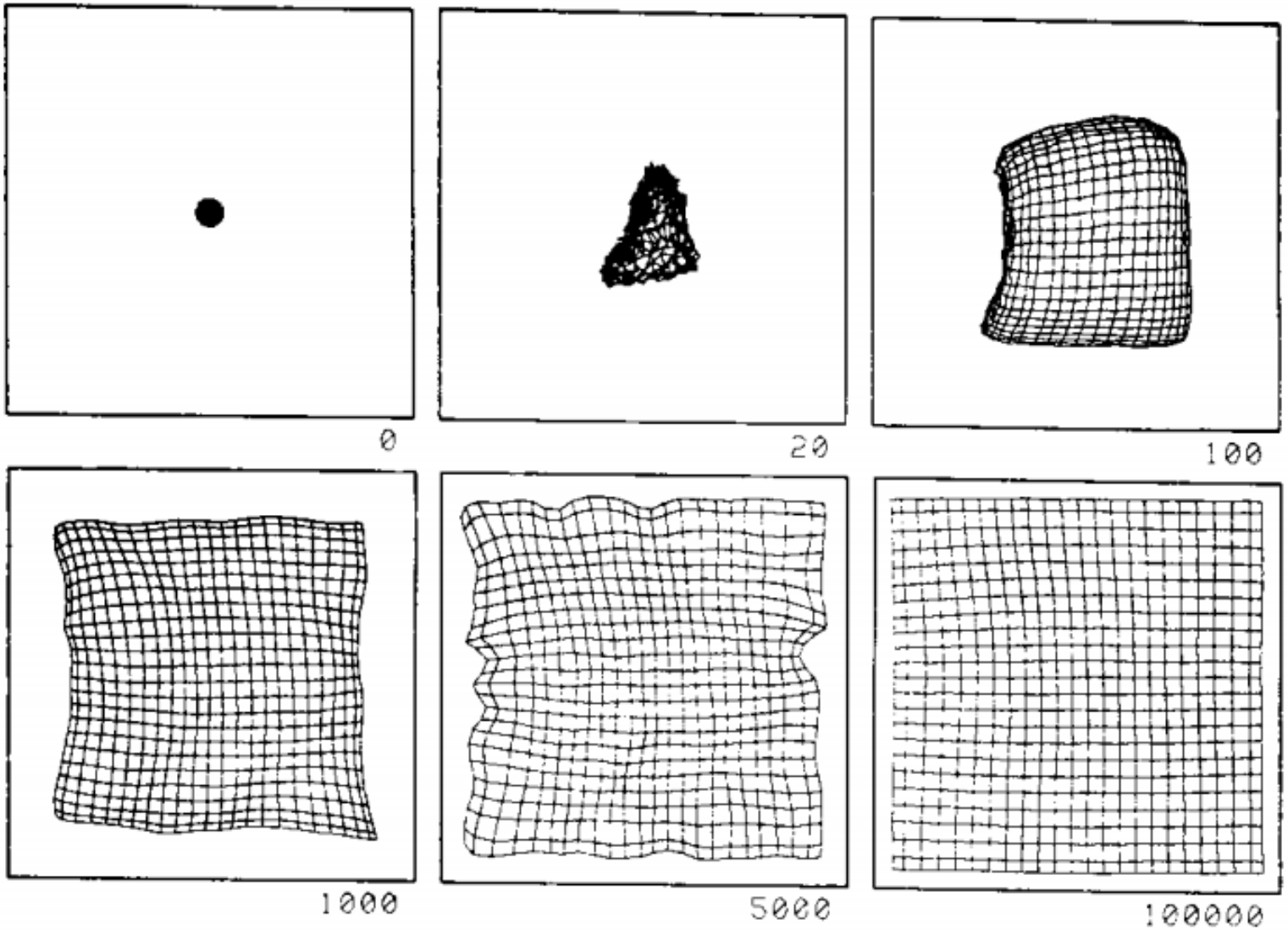
$$\Theta(c(\vec{x}_t), k) = \begin{cases} \theta & k \in N_s(c(\vec{x}_t)) \\ 0 & \text{jinak} \end{cases}$$

A smoother version:

$$\Theta(c(\vec{x}_t), k) = \theta_0 \cdot \exp\left(\frac{-d(c(\vec{x}_t), k)^2}{\sigma^2}\right)$$

where $\theta_0 \in \mathbb{R}$ is a learning rate and $\sigma \in \mathbb{R}$ is the width (both parameters may change during training).

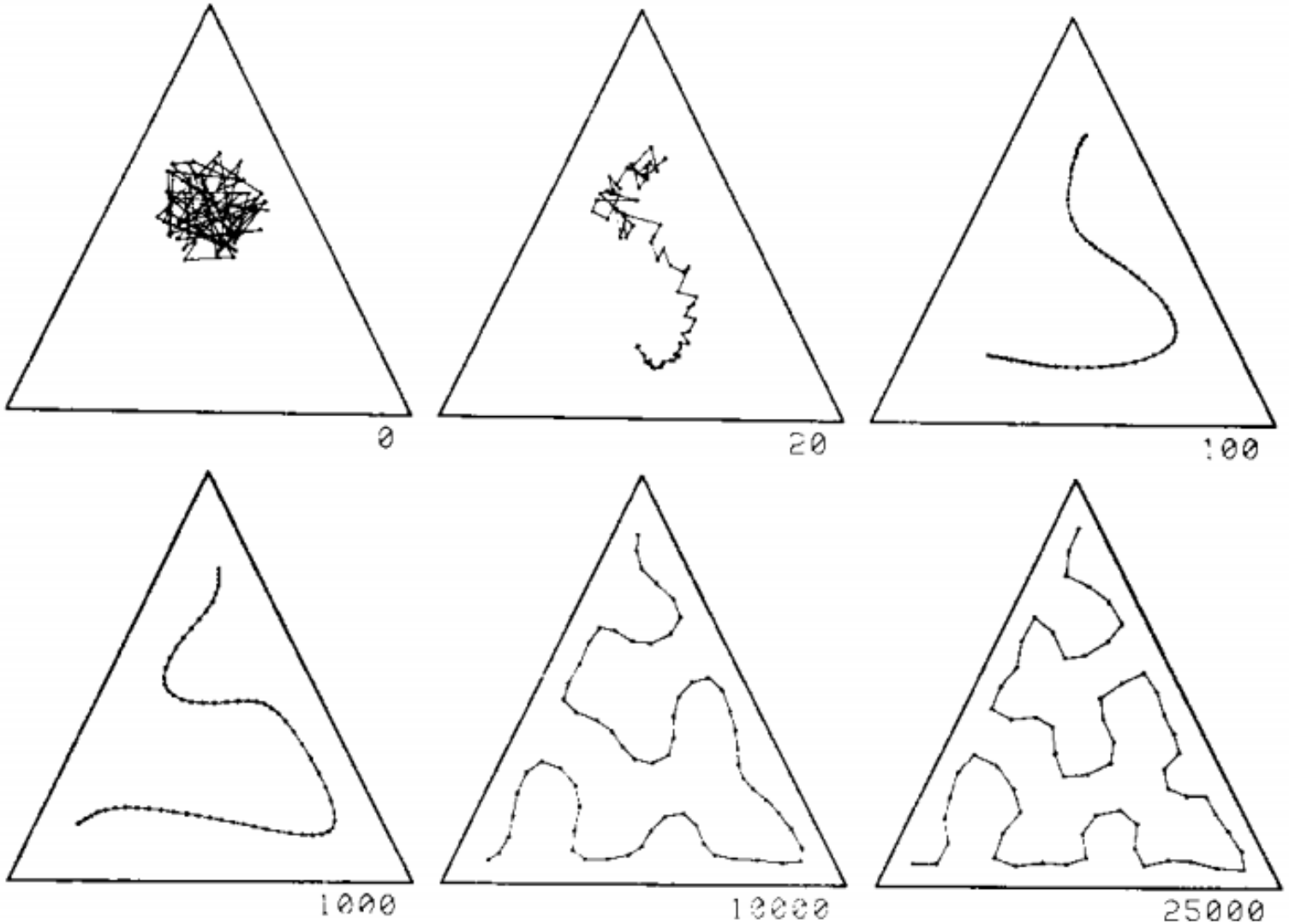
Example 1



Inputs uniformly distributed in a rectangle.

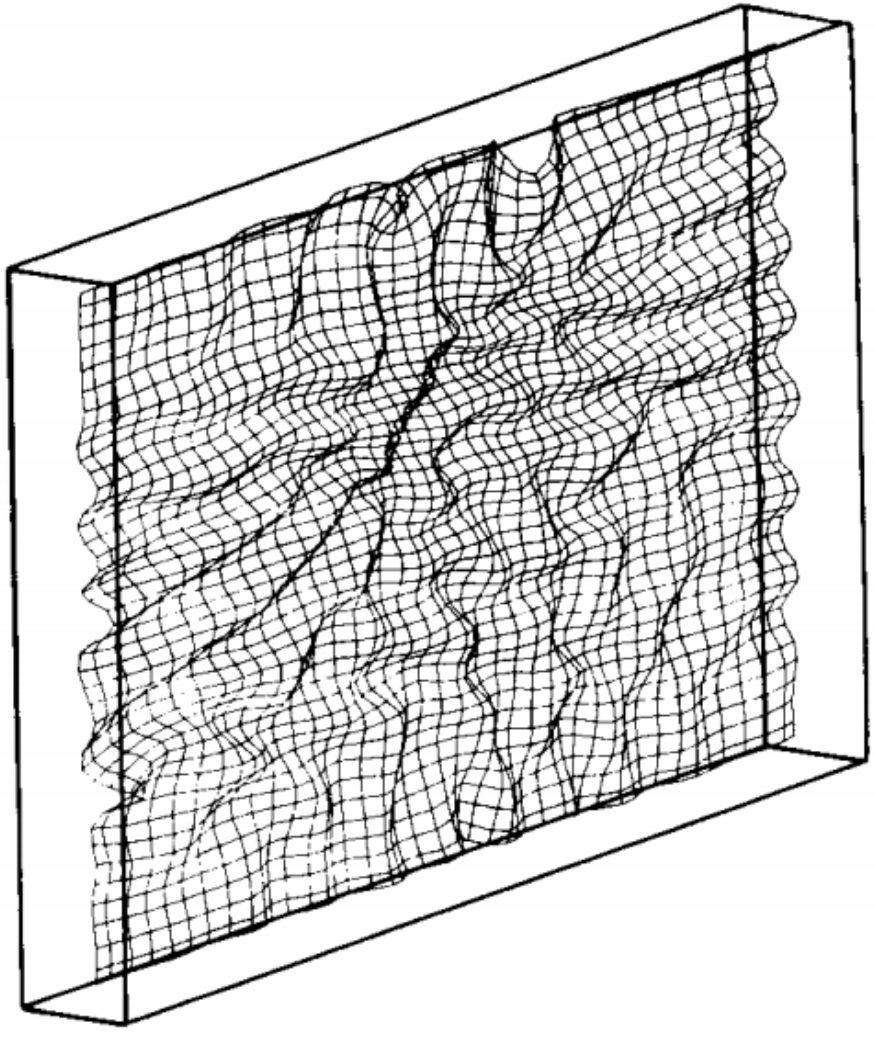
Image source: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Example 2



Inputs uniformly distributed in a triangle. Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

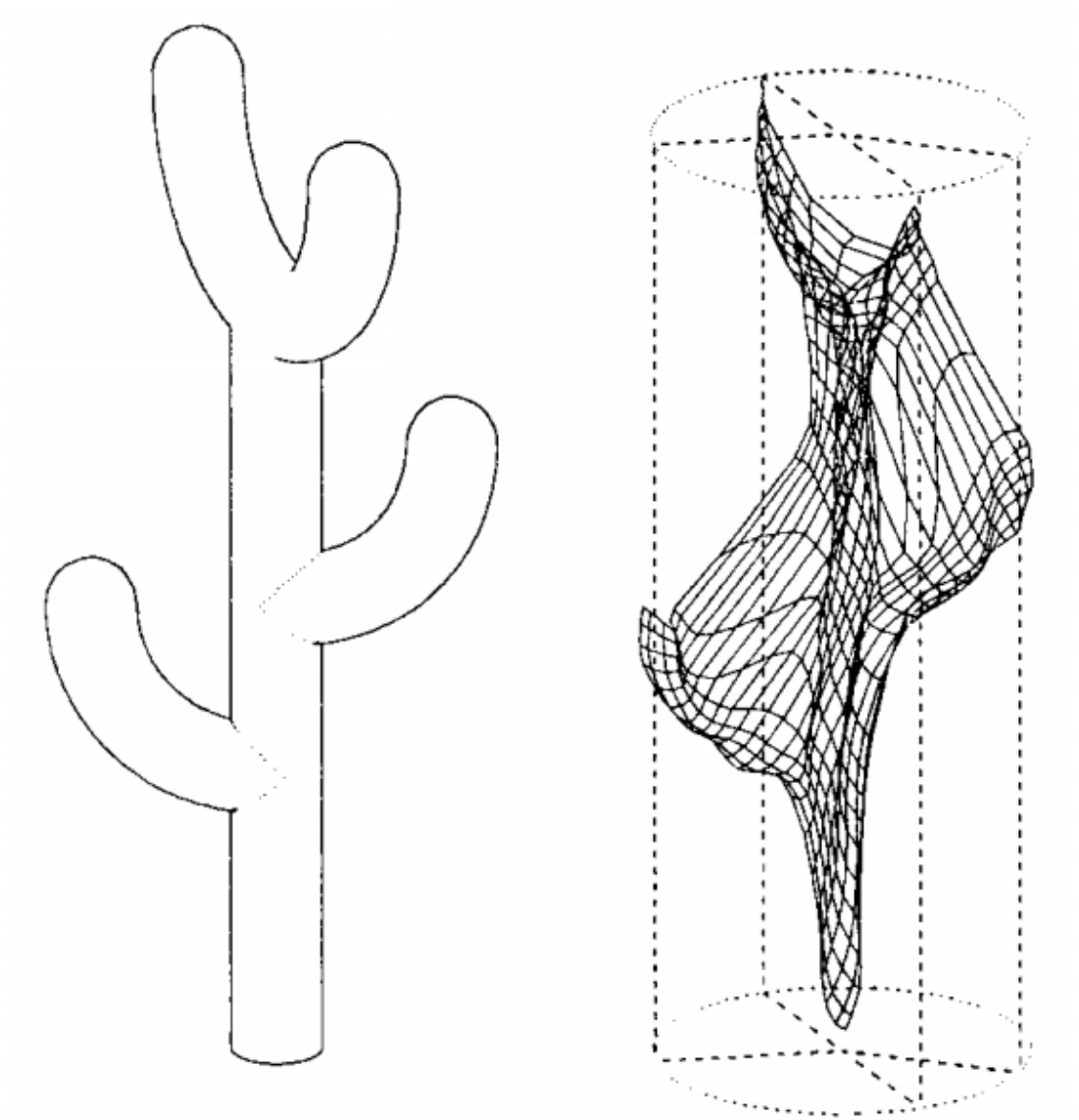
Example 3



Inputs uniformly distributed in a cuboid.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

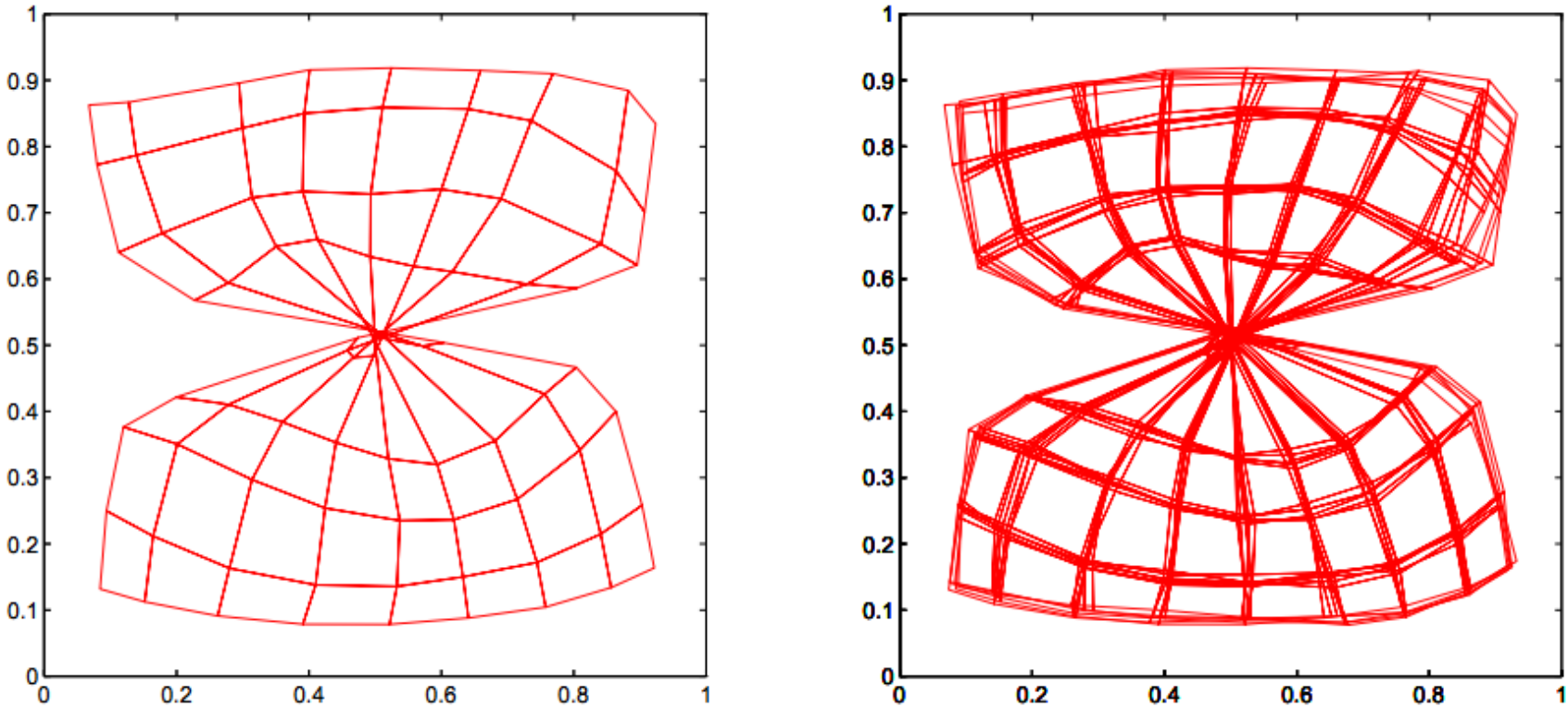
Example 4



Inputs uniformly distributed in a cactus.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Example – defect



Topological defect – twisted network.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Kohonen's map – theory

- Convergence to "ordered" state has been proved only for one dimensional maps and special cases of the distribution $p(\vec{x})$ (uniform), fixed neighbourhoods of size 1, and a fixed learning rate.

  There are simple counterexamples disproving convergence in case these assumptions are not satisfied.

Kohonen's map – theory

- Convergence to "ordered" state has been proved only for one dimensional maps and special cases of the distribution $p(\vec{x})$ (uniform), fixed neighbourhoods of size 1, and a fixed learning rate.

  There are simple counterexamples disproving convergence in case these assumptions are not satisfied.
- In more than one dimension there are no guarantees at all, convergence depends on several factors:
  - initial distribution of neurons (centres)
  - size of the neighbourhood
  - learning rate
- What dimension to choose? Typically one or two dimensional map is used (as a coarse version of dimensionality reduction).

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

Ex.: Conveyor belt with fruits, apples and oranges: Formally, $(\vec{x}_t, d_t)$ where

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

Ex.: Conveyor belt with fruits, apples and oranges: Formally, $(\vec{x}_t, d_t)$ where

- $\vec{x}_t \in \mathbb{R}^2$, here the first component is the weight and the second the diameter.

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

Ex.: Conveyor belt with fruits, apples and oranges: Formally, $(\vec{x}_t, d_t)$ where

- $\vec{x}_t \in \mathbb{R}^2$, here the first component is the weight and the second the diameter.
- d_t is either A or O depending on whether the given object is an apple or an orange.

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

Ex.: Conveyor belt with fruits, apples and oranges: Formally, $(\vec{x}_t, d_t)$ where

- $\vec{x}_t \in \mathbb{R}^2$, here the first component is the weight and the second the diameter.
- d_t is either A or O depending on whether the given object is an apple or an orange.

We allow apples and oranges with the same features.

The goal is to sort out the fruits based on their weight and diameter.

Classification using Kohonen's map

We use Kohonen's map as follows:

1. Train the map on feature vectors $\vec{x}_t$ where $t = 1, \ldots, \ell$ (ignore the classes for now).

Classification using Kohonen's map

We use Kohonen's map as follows:

1. Train the map on feature vectors $\vec{x}_t$ where $t = 1, \ldots, \ell$ (ignore the classes for now).
2. Label neurons with classes. The class v_c of a given neuron c is determined as follows:

   For every neuron c and every class C_i count the number $\#(c, C_i)$ of training examples $\vec{x}_t$ with class C_i for which the neuron c returns 1 (i.e. is the closest to them).

   To c, assign the class v_c satisfying

   $$v_c = \mathit{argmax}_{C_i} \#(c, C_i)$$

Classification using Kohonen's map

We use Kohonen's map as follows:

1. Train the map on feature vectors $\vec{x}_t$ where $t = 1, \ldots, \ell$ (ignore the classes for now).
2. Label neurons with classes. The class v_c of a given neuron c is determined as follows:

   For every neuron c and every class C_i count the number $\#(c, C_i)$ of training examples $\vec{x}_t$ with class C_i for which the neuron c returns 1 (i.e. is the closest to them).

   To c, assign the class v_c satisfying

   $$v_c = \mathit{argmax}_{C_i} \#(c, C_i)$$

3. Fine tune the network using LVQ (next slide)

The trained network is used as follows: Given a feature vector $\vec{x}$, evaluate the network with $\vec{x}$ as the input. A single neuron c has the value 1, return v_c as the class of $\vec{x}$.

LVQ

Iterate over training examples. For $(\vec{x}_t, d_t)$ find the closes neuron c

$$c = \arg \min_{i=1,\ldots,h} \left\| \vec{x}_t - \vec{w}_i \right\|$$

Adjust weights of c as follows:

$$\vec{w}_c^{(t)} = \begin{cases} \vec{w}_c^{(t-1)} + \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t = v_c \\ \vec{w}_c^{(t-1)} - \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t \neq v_c \end{cases}$$

The parameter α should be small right from the beginning (approx. 0.01 – 0.02) and go to 0 steadily.

LVQ

Iterate over training examples. For $(\vec{x}_t, d_t)$ find the closes neuron c

$$c = \arg \min_{i=1,\ldots,h} \left\| \vec{x}_t - \vec{w}_i \right\|$$

Adjust weights of c as follows:

$$\vec{w}_c^{(t)} = \begin{cases} \vec{w}_c^{(t-1)} + \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t = v_c \\ \vec{w}_c^{(t-1)} - \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t \neq v_c \end{cases}$$

The parameter α should be small right from the beginning (approx. 0.01 – 0.02) and go to 0 steadily.

By Kohonen: The border between classes should be a good approximation of the Bayes decision boundary.

What is it??

Bayes classifier

For simplicity, consider two classes C_0 and C_1 (e.g. A and O).

Let $P(C_i \mid \vec{x})$ be the probability that the object belongs to C_i assuming that it has features $\vec{x}$.

(e.g. $P(A \mid (a, b))$ is the probability that a fruit with weight a and diameter b is an apple.)

Bayes classifier assigns to $\vec{x}$ the class C_i which satisfies $P(C_i \mid \vec{x}) \geq P(C_{1-i} \mid \vec{x})$.

Denote by R_0 the set of all $\vec{x}$ satisfying $P(C_0 \mid \vec{x}) \geq P(C_1 \mid \vec{x})$ and $R_1 = \mathbb{R}^n \setminus R_0$.

Bayes classifier

For simplicity, consider two classes C_0 and C_1 (e.g. A and O).

Let $P(C_i \mid \vec{x})$ be the probability that the object belongs to C_i assuming that it has features $\vec{x}$.

(e.g. $P(A \mid (a, b))$ is the probability that a fruit with weight a and diameter b is an apple.)

Bayes classifier assigns to $\vec{x}$ the class C_i which satisfies $P(C_i \mid \vec{x}) \geq P(C_{1-i} \mid \vec{x})$.

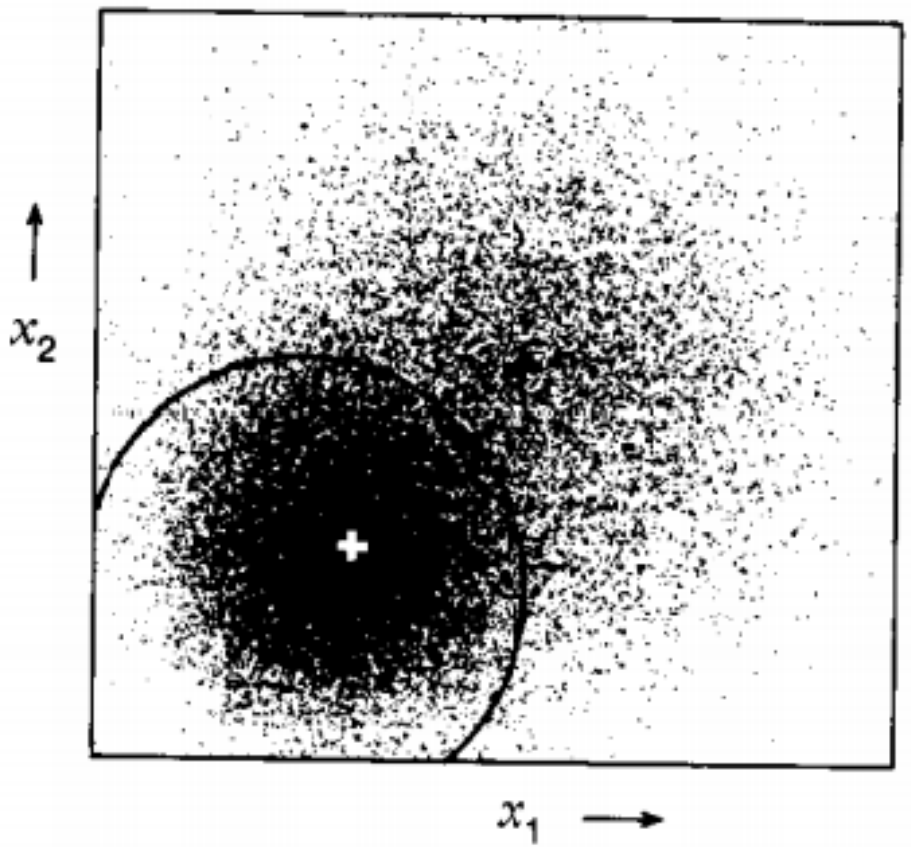
Denote by R_0 the set of all $\vec{x}$ satisfying $P(C_0 \mid \vec{x}) \geq P(C_1 \mid \vec{x})$ and $R_1 = \mathbb{R}^n \setminus R_0$.

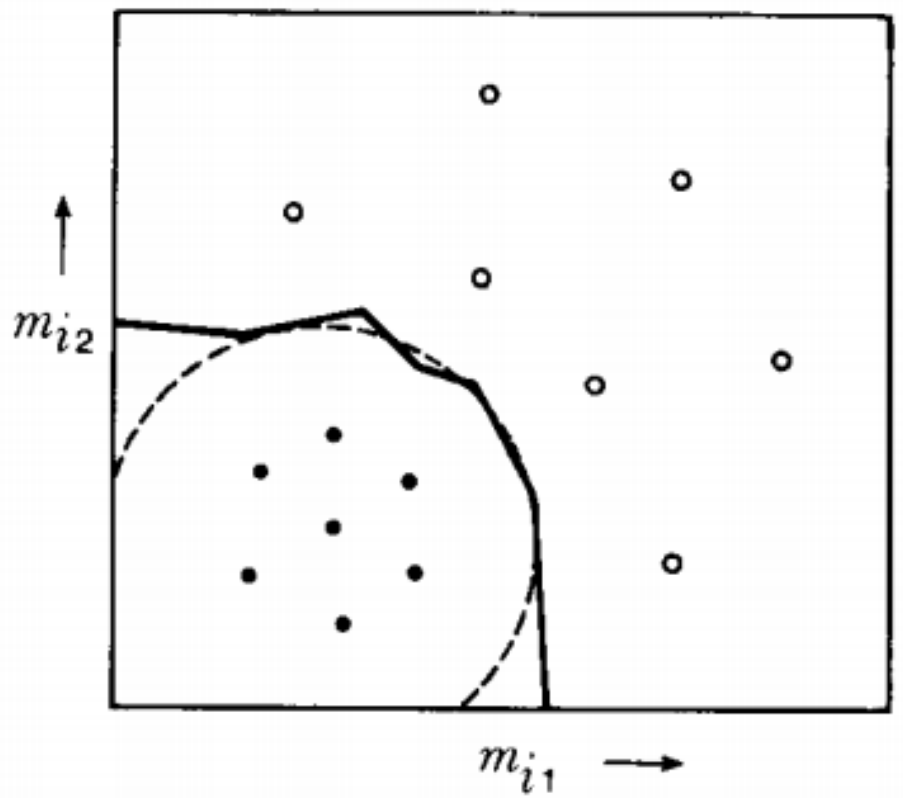
Bayes classifier minimizes the error probability:

$$P(\vec{x} \in R_0 \wedge C_1) + P(\vec{x} \in R_1 \wedge C_0)$$

Bayes decision boundary is the boundary between the sets R_0 and R_1.

Bayes decision boundary vs LVQ



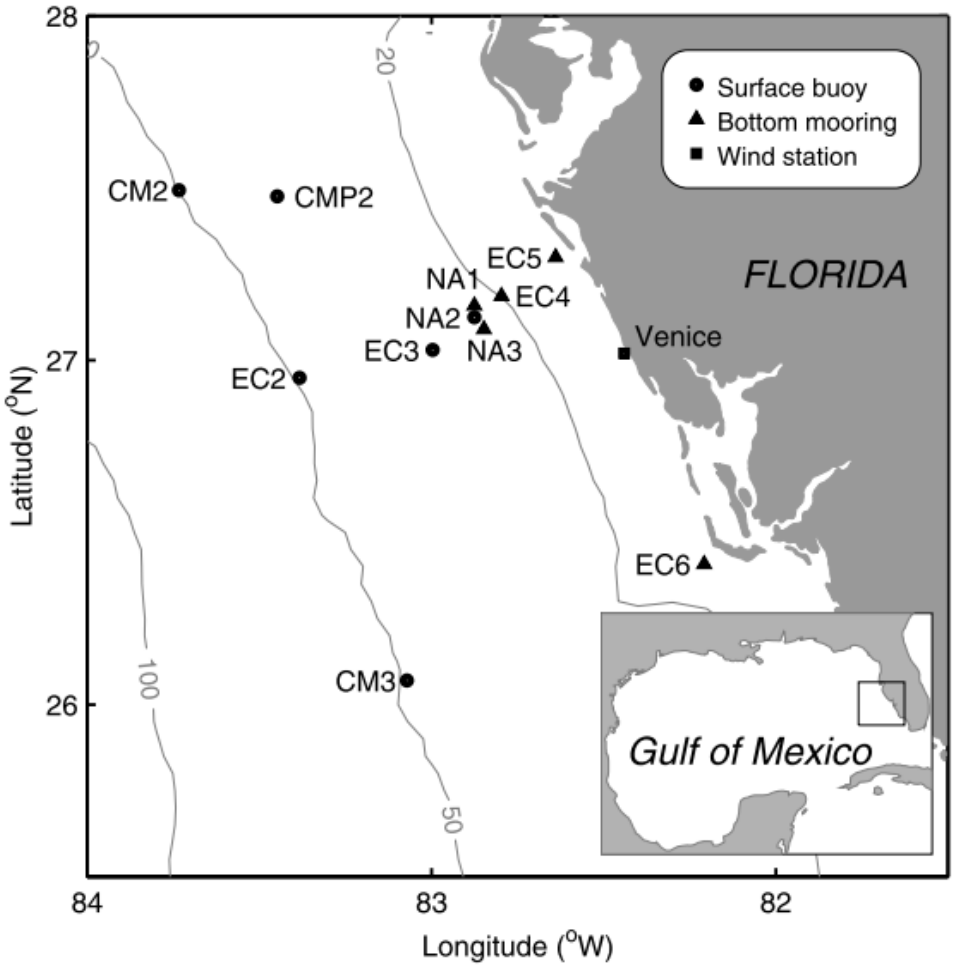


Zdroj obrázku: The Self-Organizing Map, Teuvo Kohonen, IEEE, 1990

Oceanographic data

Source: Patterns of ocean current variability on the West Florida Shelf using the self-organizing map. Y. Liu a R. H. Weisberg, JOURNAL OF GEOPHYSICAL RESEARCH, 2005

Investigates currents in the ocean around Florida.



Oceanographic data

- 11 measuring stations, 3 depths (surface, bottom, in between).
- data: 2D velocity vectors of the current
- measured by every hour, for 25585 hours

Oceanographic data

- 11 measuring stations, 3 depths (surface, bottom, in between).
- data: 2D velocity vectors of the current
- measured by every hour, for 25585 hours

Thus we have 25585 data samples, 66 dimensions.

Kohonen's map:

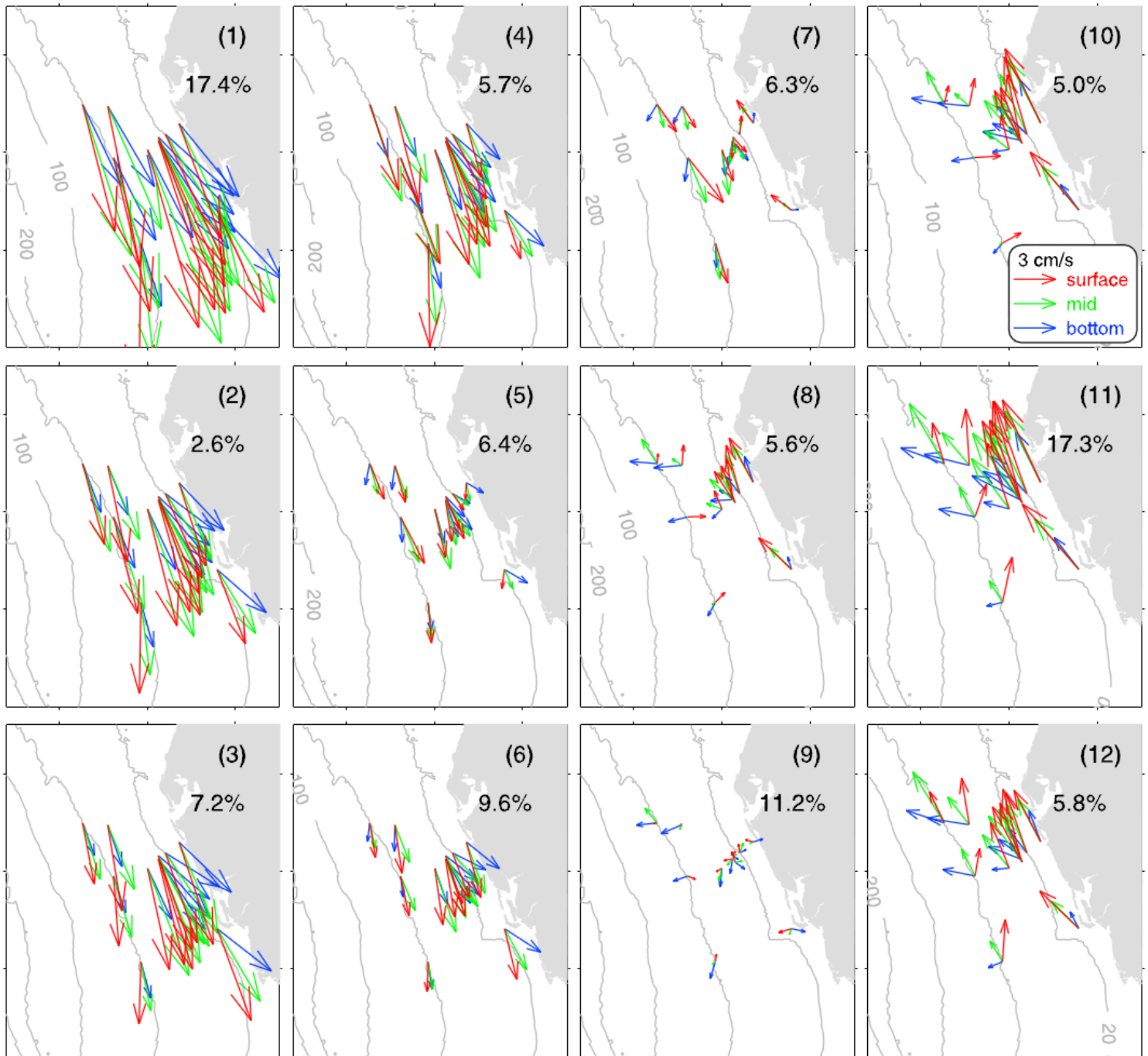
- grid 3×4
- neighbourhoods given by Gaussian functions

$$\Theta(c,k) = \theta_0 \cdot \exp\left(\frac{-d(c,k)^2}{\sigma^2}\right)$$

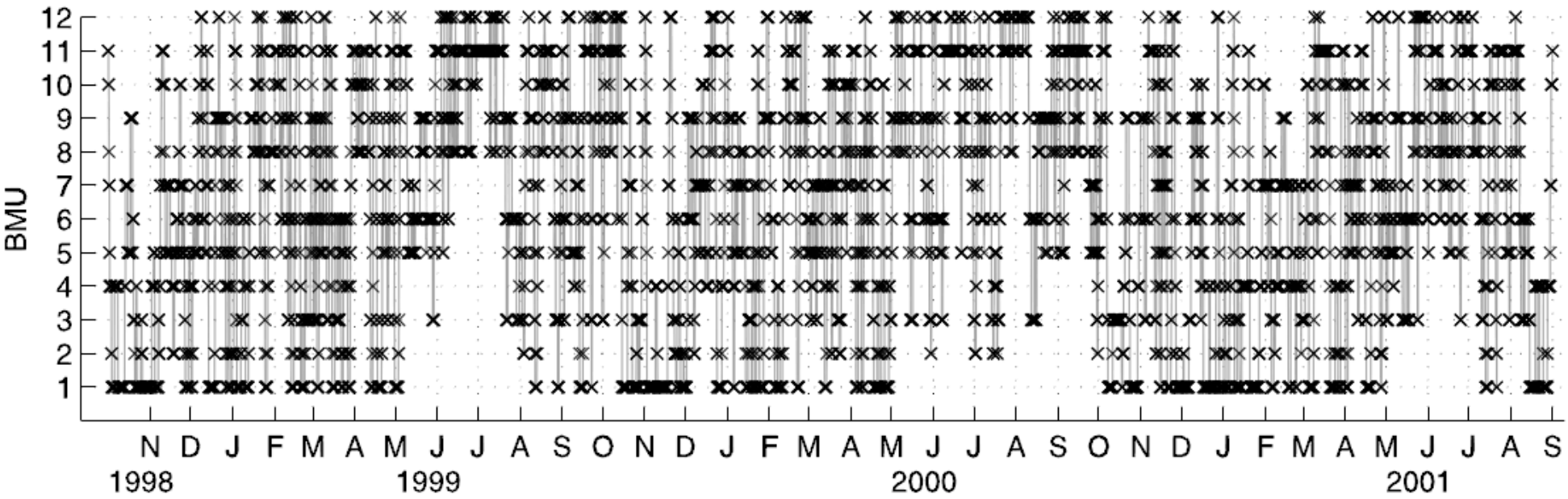
shrinking width

(linearly decreasing learning rate)

Oceanographic data



Oceanographic data



- crosses are winning neurons)
- influenced by local fluctuations
- observable trend:
  - winter: neurons 1-6 (south-east)
  - summer: neurons 10-12 (north-west)

Grimm's fairy tales

Zdroj: Contextual Relations of Words in Grimm Tales, Analyzed by Self-Organizing Map. T. Kohonen, T. Honkela a V. Pulkki, ICANN, 1995

Our goal is to visualize syntactic and semantic categories of words in fairy tales (depending on context).

Grimm's fairy tales

Zdroj: Contextual Relations of Words in Grimm Tales, Analyzed by Self-Organizing Map. T. Kohonen, T. Honkela a V. Pulkki, ICANN, 1995

Our goal is to visualize syntactic and semantic categories of words in fairy tales (depending on context).

Input: Grimm's fairy tales (understandably encoded using a stream of 270-dimensional vectors)

- triples of words (predecessor, key, successor)
- every component in the triple encoded using a randomly generated 90 dimensional real vector

Network: Kohonen's map, 42×36 neurons, weights of the form $w = (w_p, w_k, w_n)$ where $w_p, w_k, w_n \in \mathbb{R}^{90}$.

Grimm's fairy tales

Learning:

Trained on triples of successive words in fairy tales

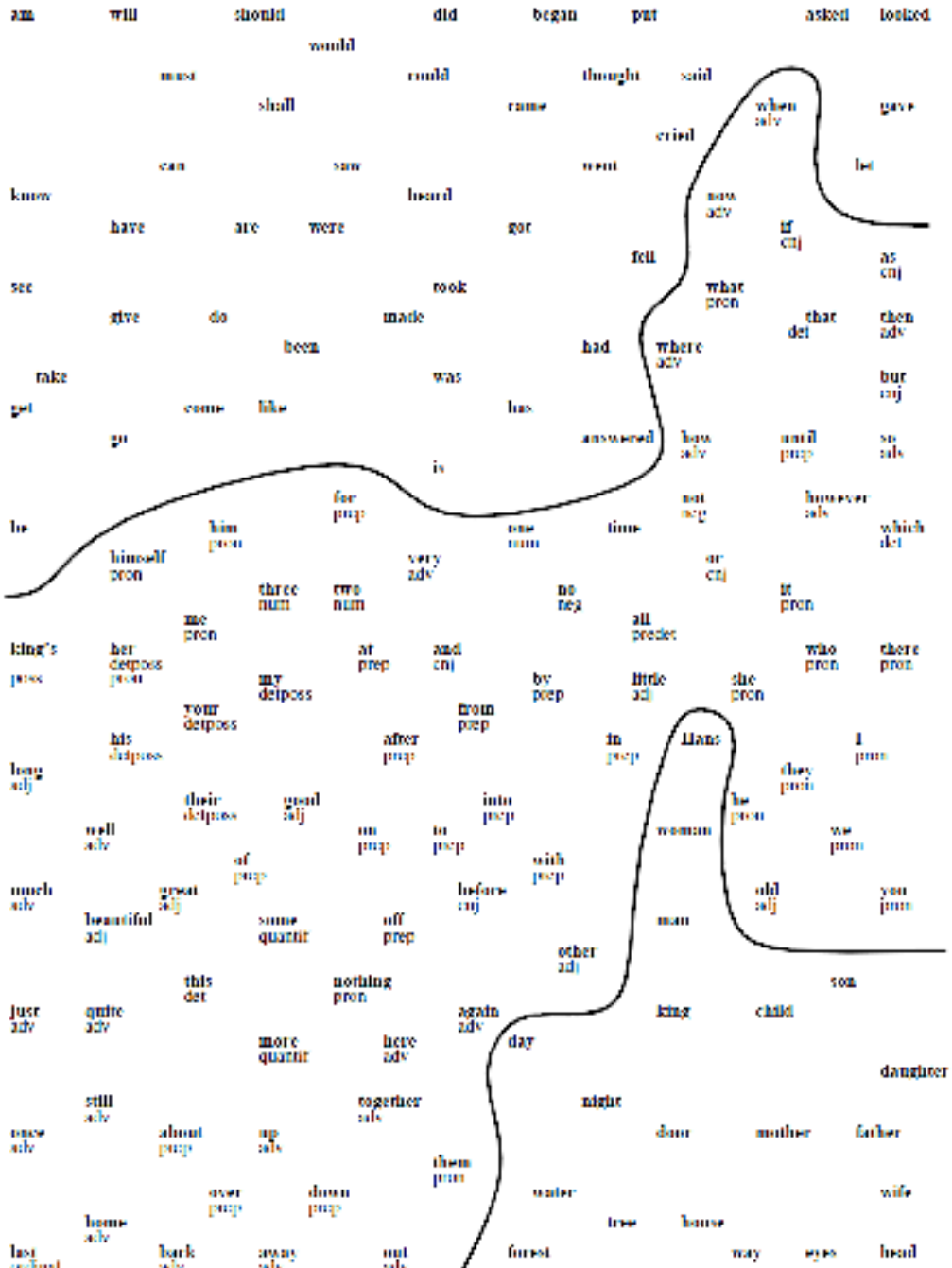
The training set consisted of 150 most common words, with "average" context.

Training: Approx. 1000 000 iterations.

In the end, 150 most common words labelled neurons:

A word u labels a neuron with weights $w = (w_p, w_k, w_n)$ when w_k is closest to the code of u.

Grimm's fairy tales



Course Summary

Great summary – models

We have considered several models of neural networks:

- ADALINE (aka linear regression)
- Multilayer Perceptron
- Hopfield Networks
- Convolutional Networks
- Recurrent Networks (LSTM)
- Restricted Boltzmann Machines and Deep Belief Networks
- Kohonen's Maps

Great summary – algorithms

Gradient descent!

The only exception were Kohonen's maps (Kohonen learning) and Hopfield (Hebb's learning).

The gradient computed using

- Backpropagation: MLP, Convolutional, Recurrent (LSTM)
- Simulations: RBM

Deeper thoughts

- Most neural network models are universal approximators (i.e. capable of approximating any reasonable function), but it is difficult to find the appropriate configuration → such configuration can be learned efficiently (without guarantees of course)
- Depth is stronger than size: deep networks are more succinct in their representation but are harder to train: Do not forget the vanishin/exploding gradient problem!
- Weight tying = single most effective trick in the history of neural networks!