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.)
- Inifinitely many online tutorials on everything (to build intuition)

Suggested: deeplearning.ai courses by Andrew Ng

Course organization

Evaluation:

- Project
  - teams of two students
  - implementation of a selected model + analysis of given data
  - implementation either in C, C++ **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++ **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! You will get a detailed manual specifying the mandatory knowledge.

FAQ

Q: Why we cannot use specialized libraries in projects?

FAQ

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.

FAQ

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.

Q: Why should you attend this course when there are infinitely many great reasources elsewhere?

A: There are at least two reasons:

- You may discuss issues with me, my colleagues and other students.
- I will make you truly learn fundamentals by heart.

Notable features of the course

- Use of mathematical notation and reasoning (contains several proofs that are mandatory for the exam)
- Sometimes goes deeper into statistical underpinnings of neural networks learning
- The project demands a complete working solution which must satisfy a prescribed performance specification

Notable features of the course

- Use of mathematical notation and reasoning (contains several proofs that are mandatory for the exam)
- Sometimes goes deeper into statistical underpinnings of neural networks learning
- The project demands a complete working solution which must satisfy a prescribed performance specification

An unusual exam system! You can repeat the oral exam as many times as needed (only the best grade goes into IS).

Notable features of the course

- Use of mathematical notation and reasoning (contains several proofs that are mandatory for the exam)
- Sometimes goes deeper into statistical underpinnings of neural networks learning
- The project demands a complete working solution which must satisfy a prescribed performance specification

An unusual exam system! You can repeat the oral exam as many times as needed (only the best grade goes into IS).

An example of an instruction email (from another course with the same system):

```
It is typically not sufficient to devote a single
afternoon to the preparation for the exam.
You have to know _everything_ (which means every
single thing) starting with the slide 42
and ending with the slide 245 with notable exceptions
of slides: 121 - 123, 137 - 140, 165, 167.
Proofs presented on the whiteboard are also mandatory.
```

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

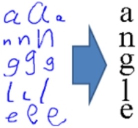
- Machine learning = construction of systems that may learn their functionality from data

  (... 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

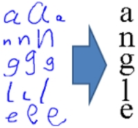
  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labelled" emails
    - consequently is able to distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labelled by their correct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

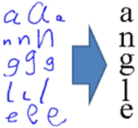


  - ...
  - and lots of much much more sophisticated applications ...

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.

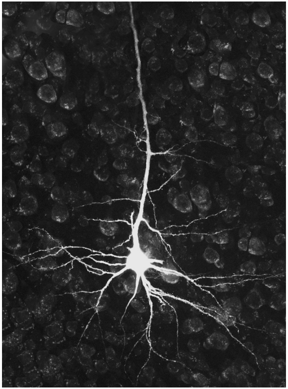
There are many types of models:

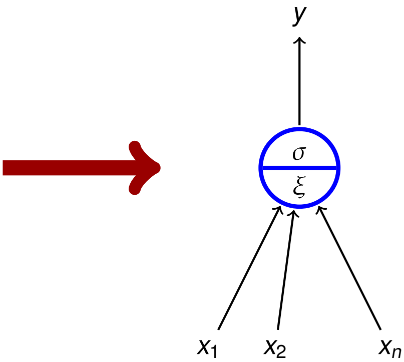
- decision trees
- support vector machines
- hidden Markov models
- Bayes networks and other graphical models
- **neural networks**
- ...

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, Starcraft, ...
  - finance - stock prices, risk analysis
  - medicine - diagnosis, signal processing (EKG, EEG, ...), image processing (MRI, CT, WSI ...)
  - text and speech processing - machine translation, text generation, speech recognition
  - other signal processing - filtering, radar tracking, noise reduction
  - art - music and painting generation, deepfakes
  - ...

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 an image 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 an image 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, recurrent networks, transformers, autoencoders and generative adversarial networks)

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, recurrent networks, transformers, autoencoders and generative adversarial networks)
  - Simple applications of these models
    (image processing, a little bit of 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, recurrent networks, transformers, autoencoders and generative adversarial networks)
  - Simple applications of these models
    (image processing, a little bit of speech and text processing)
  - Basic learning algorithms
    (gradient descent with backpropagation)

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, recurrent networks, transformers, autoencoders and generative adversarial networks)
  - Simple applications of these models
    (image processing, a little bit of speech and text processing)
  - Basic learning algorithms
    (gradient descent with backpropagation)
  - Basic practical training techniques
    (data preparation, setting various hyper-parameters, control of learning, improving generalization)

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, recurrent networks, transformers, autoencoders and generative adversarial networks)
  - Simple applications of these models
    (image processing, a little bit of speech and text processing)
  - Basic learning algorithms
    (gradient descent with backpropagation)
  - Basic practical training techniques
    (data preparation, setting various hyper-parameters, control of learning, improving generalization)
  - Basic information about current implementations
    (TensorFlow-Keras, Pytorch)

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.

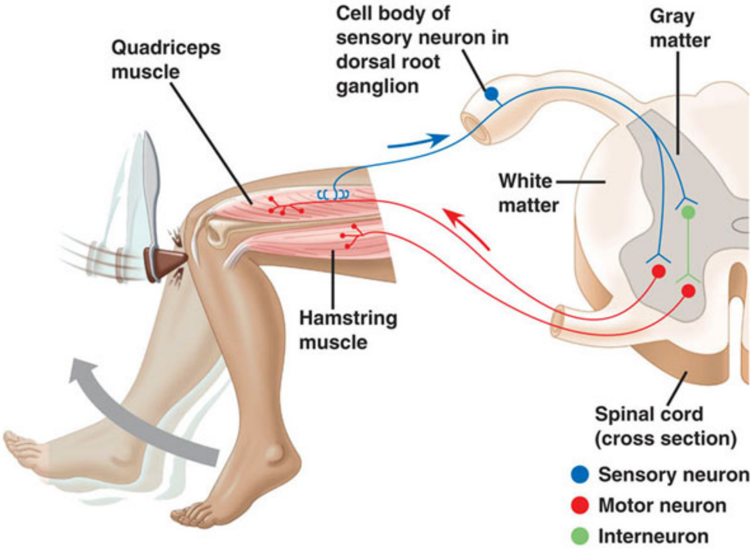
Biological neural network

- 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

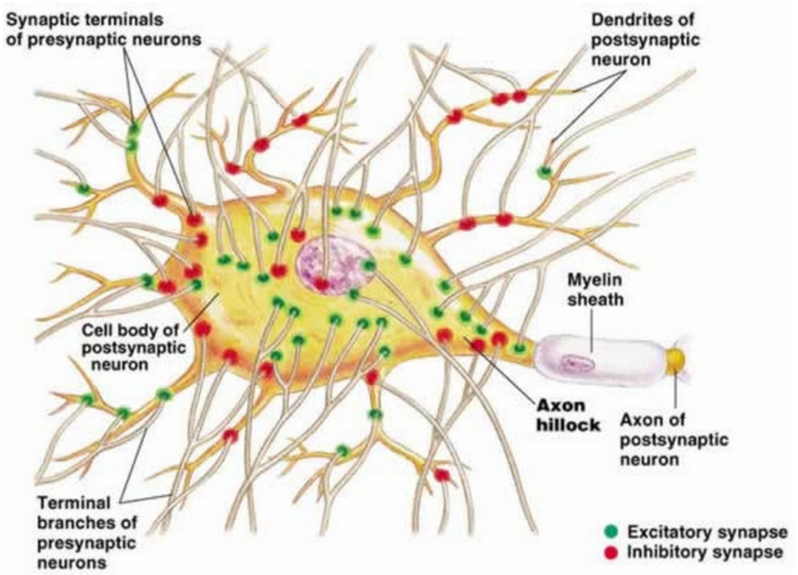
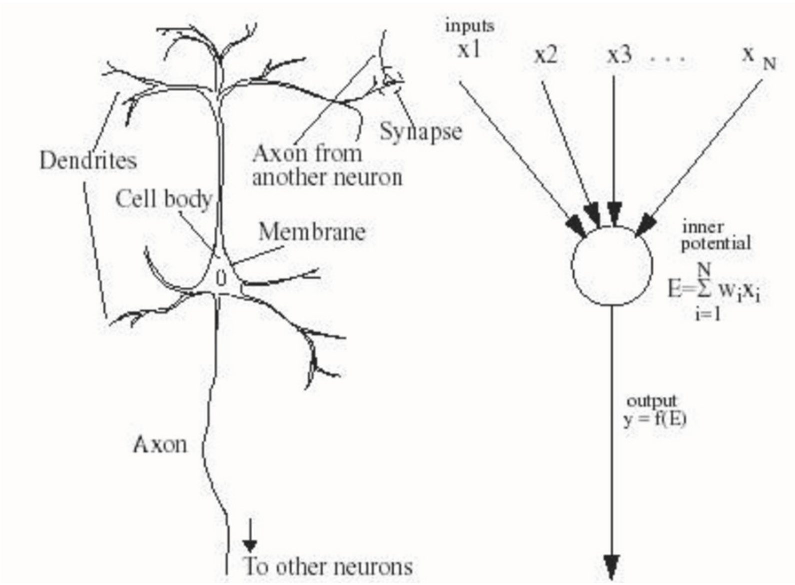
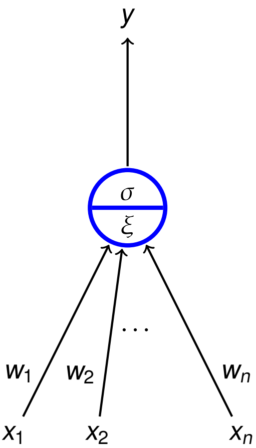


Figure 48.11(a), page 972, Campbell's *Biology, 5th Edition*

Biological and Mathematical neurons

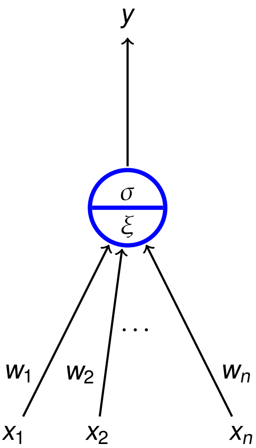


Formal neuron (without bias)



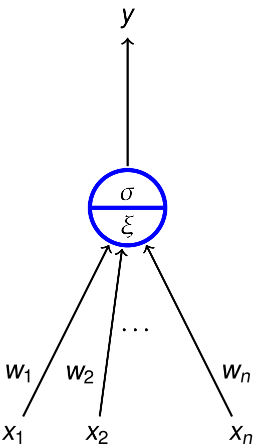
- $x_1, \dots, x_n \in \mathbb{R}$ are **inputs**

Formal neuron (without bias)



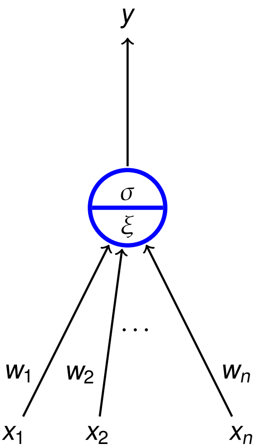
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

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**;

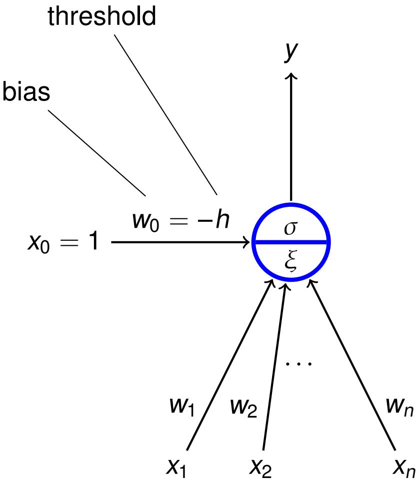
  e.g. a *unit step function*

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

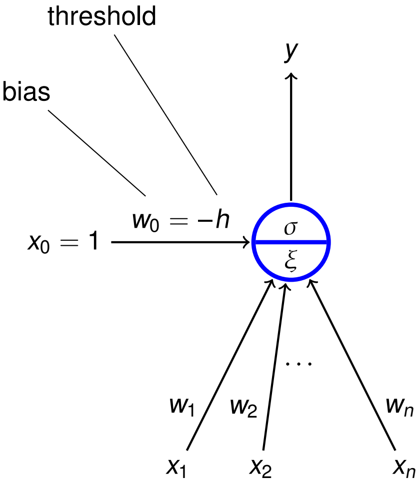
where $h \in \mathbb{R}$ is a *threshold*.

Formal neuron (with bias)

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

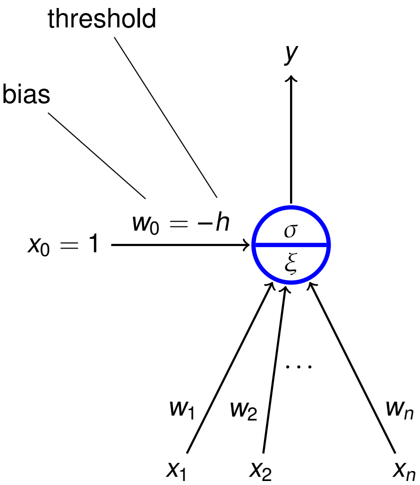


Formal neuron (with bias)



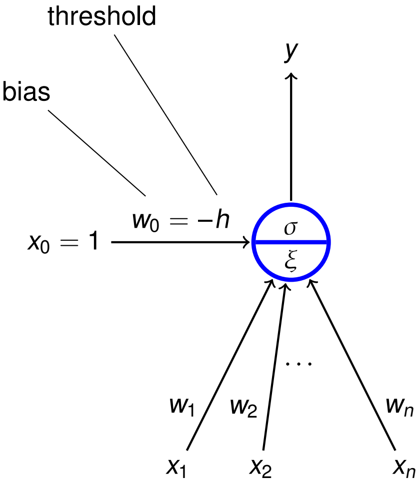
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

Formal neuron (with bias)



- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$

Formal neuron (with bias)



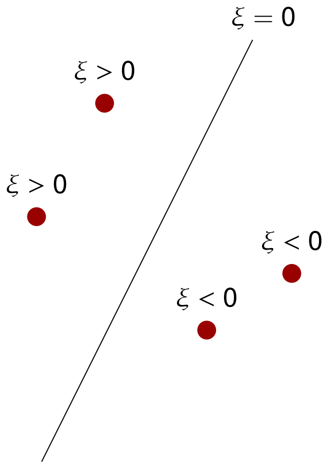
- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$
  where σ is an **activation function**;

  e.g. a *unit step function*

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

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

Neuron and linear separation



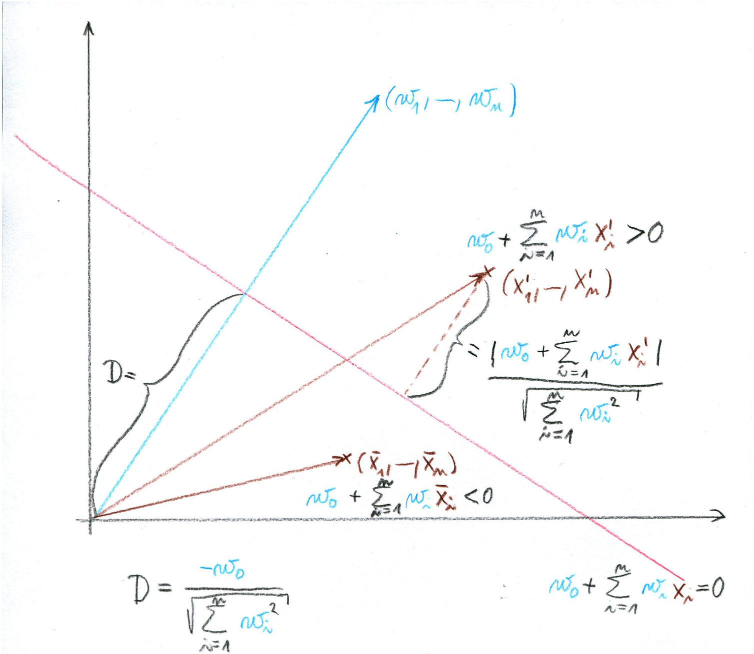
- inner potential

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

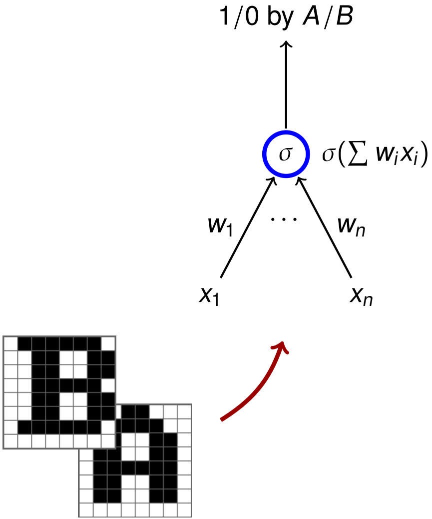
determines a separation hyperplane in the n-dimensional **input space**

  - in 2d line
  - in 3d plane
  - $\cdots$

Neuron geometry

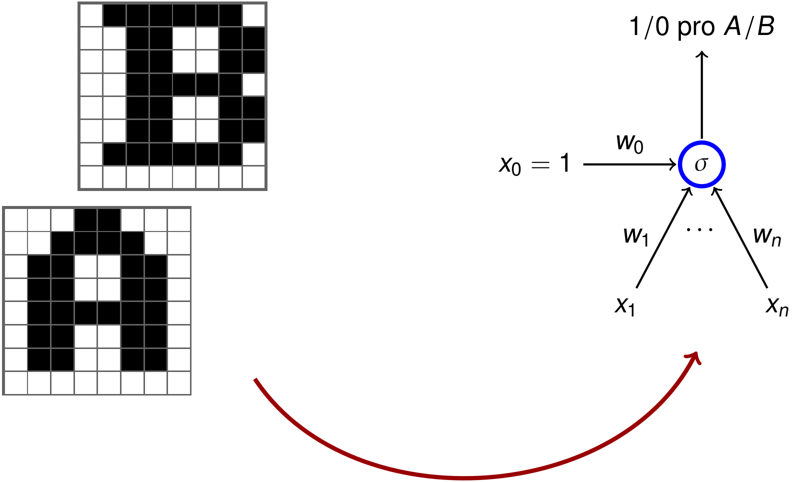


Neuron and linear separation



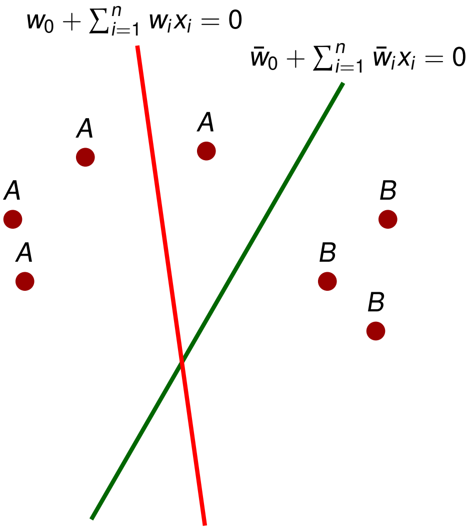
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

Neuron and linear separation



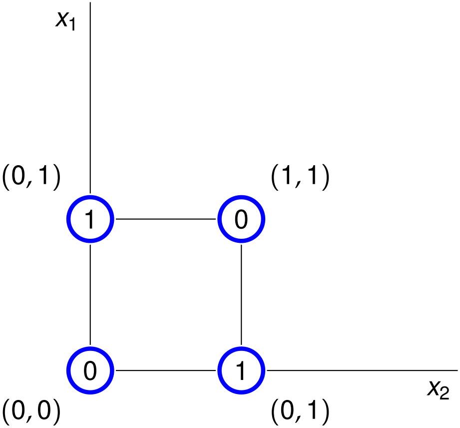
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

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.

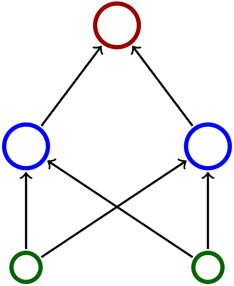
Architecture

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

We distinguish several categories of neurons:

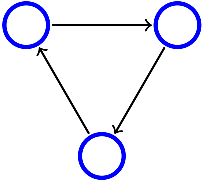
- Output neurons
- Hidden neurons
- Input neurons

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



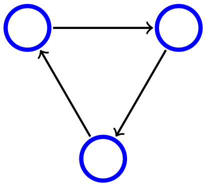
Architecture – Cycles

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

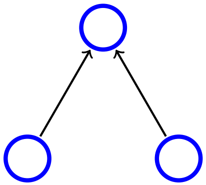


Architecture – Cycles

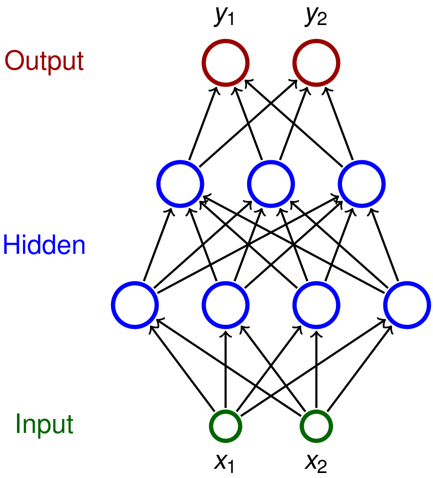
- A network is **cyclic** (recurrent) if its architecture contains a directed cycle.



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

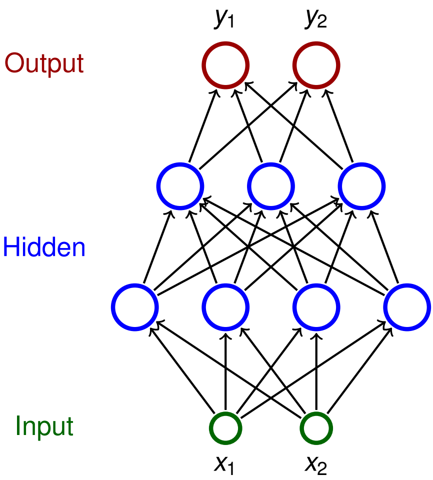


Architecture – Multilayer Perceptron (MLP)



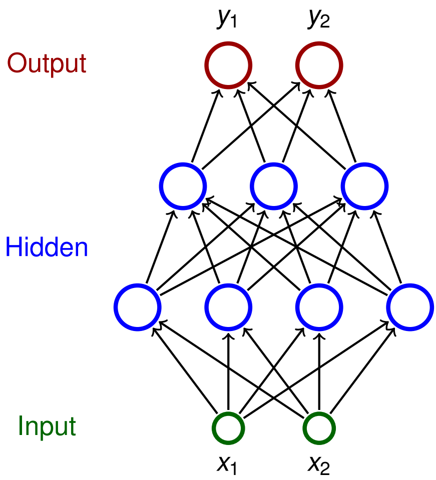
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers

Architecture – Multilayer Perceptron (MLP)



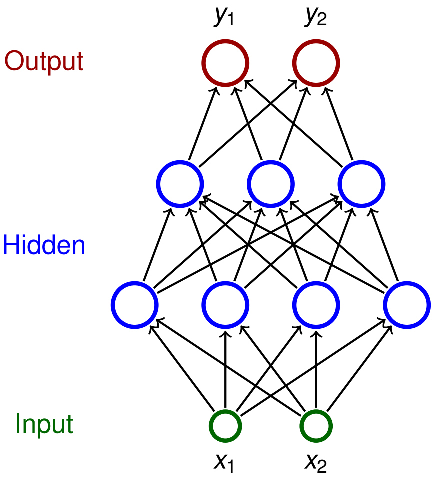
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer

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.*

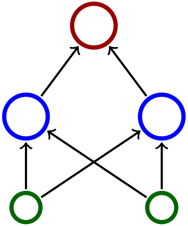
Activity – semantics of a network

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 networks 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.

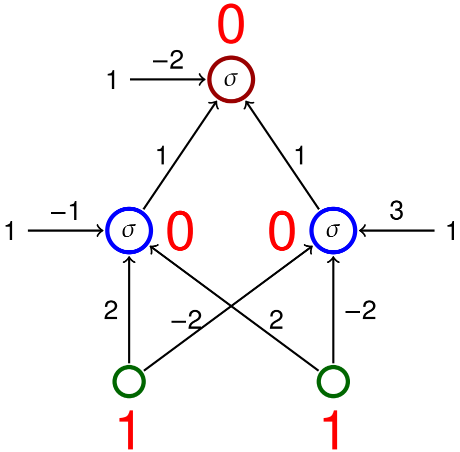
- Hyperbolic tangens

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

- ReLU

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

Activity – XOR



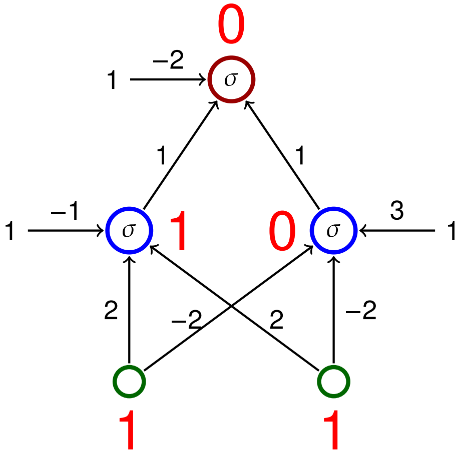
- Activation function is a unit step function

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

- 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



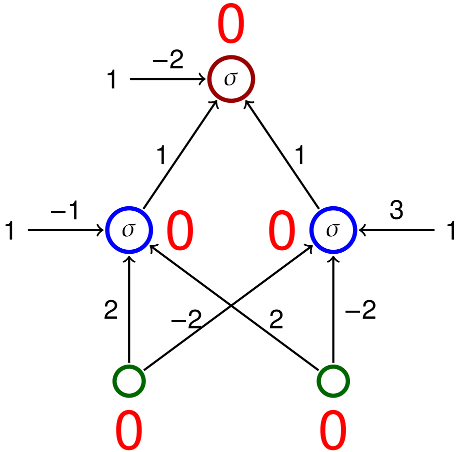
- Activation function is a unit step function

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

- 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



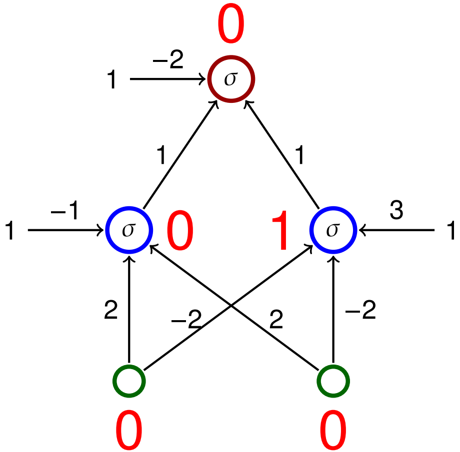
- Activation function is a unit step function

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

- 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



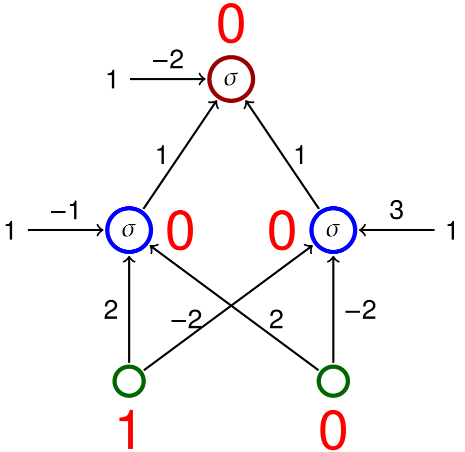
- Activation function is a unit step function

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

- 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



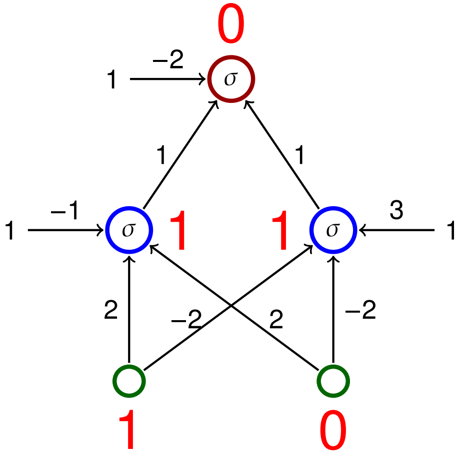
- Activation function is a unit step function

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

- 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



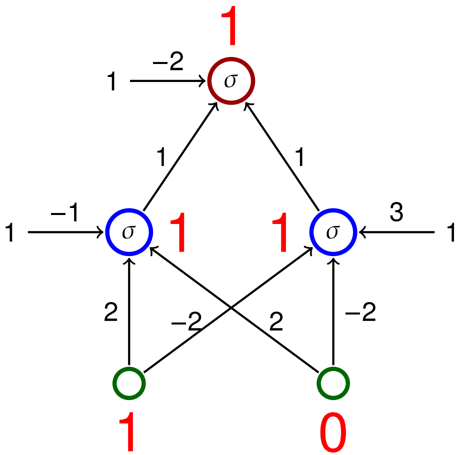
- Activation function is a unit step function

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

- 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



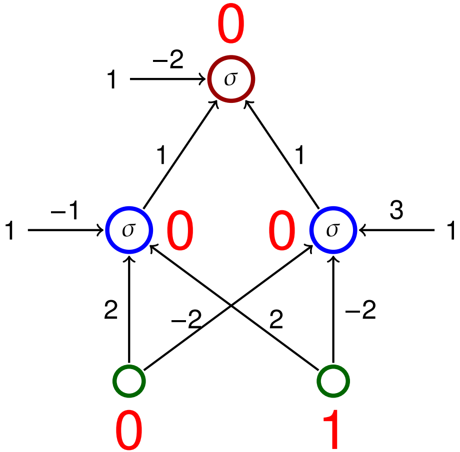
- Activation function is a unit step function

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

- 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



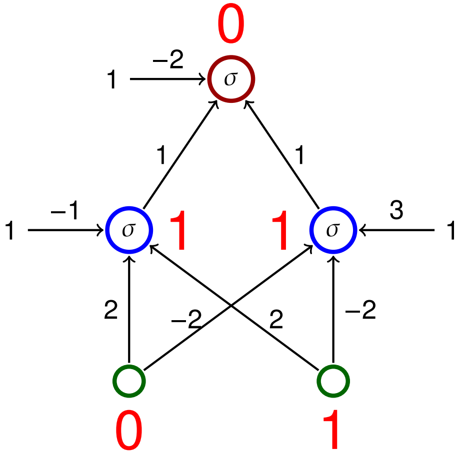
- Activation function is a unit step function

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

- 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



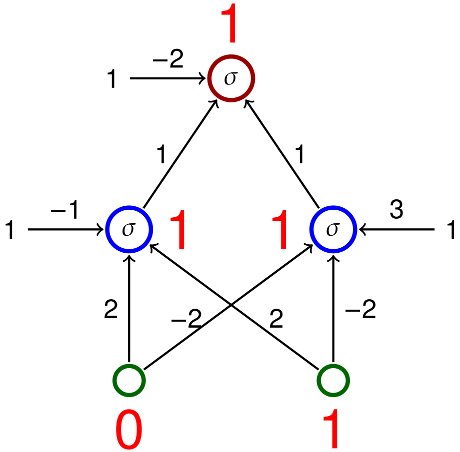
- Activation function is a unit step function

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

- 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 – XOR



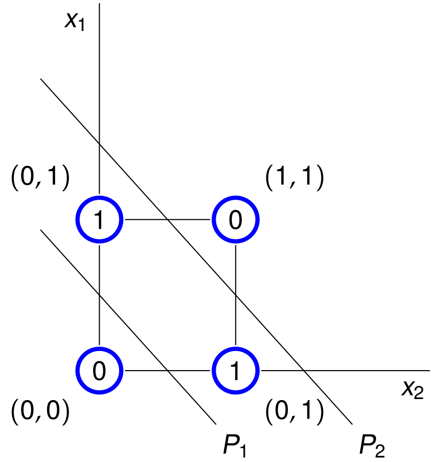
- Activation function is a unit step function

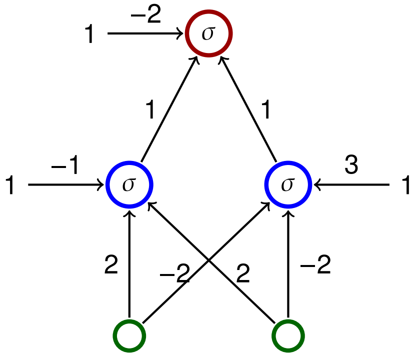
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

- 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

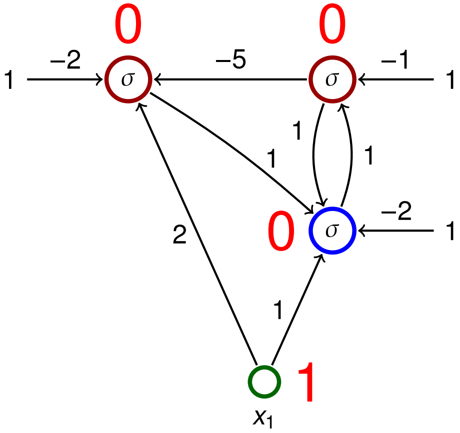
Activity – MLP and linear separation





- 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

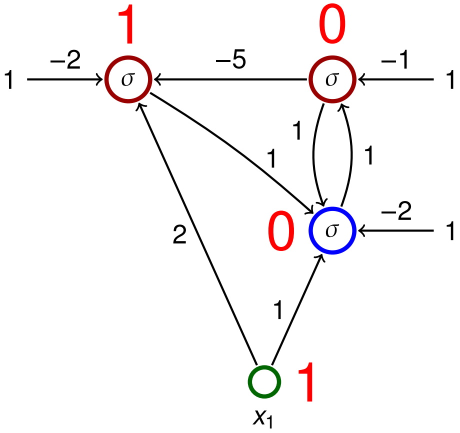


The activation function is the unit step function

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

The input is equal to 1

Activity – example

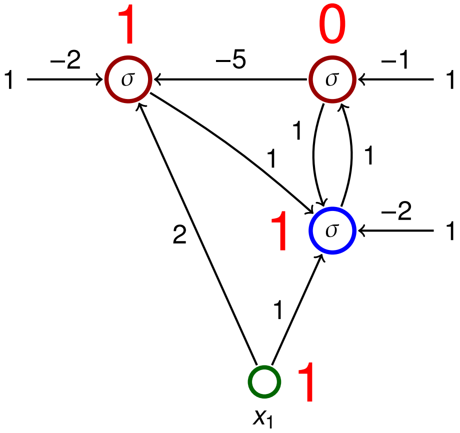


The activation function is the unit step function

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

The input is equal to 1

Activity – example

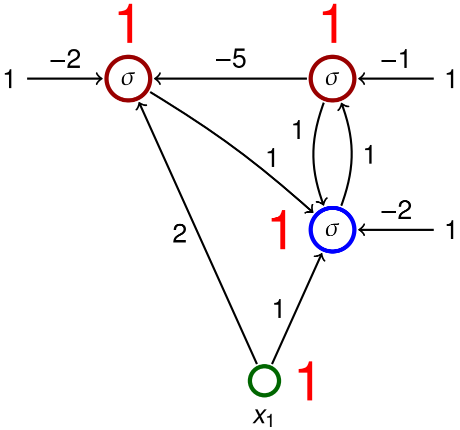


The activation function is the unit step function

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

The input is equal to 1

Activity – example

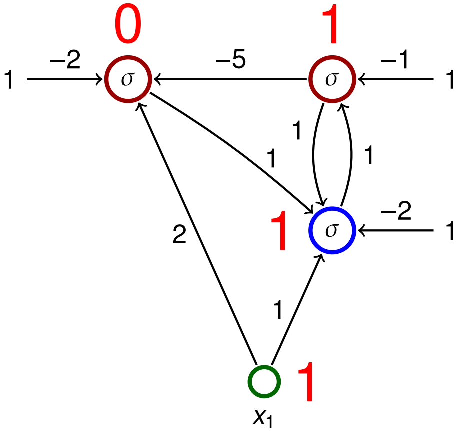


The activation function is the unit step function

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

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)

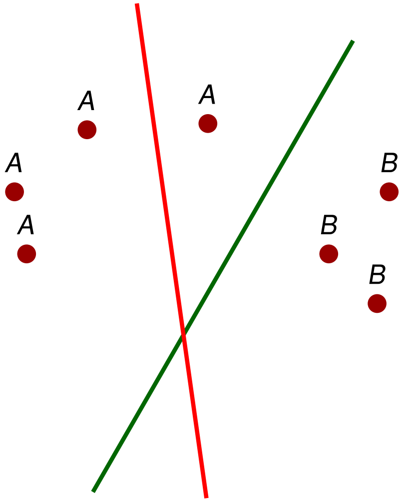
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.

Learning algorithms

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

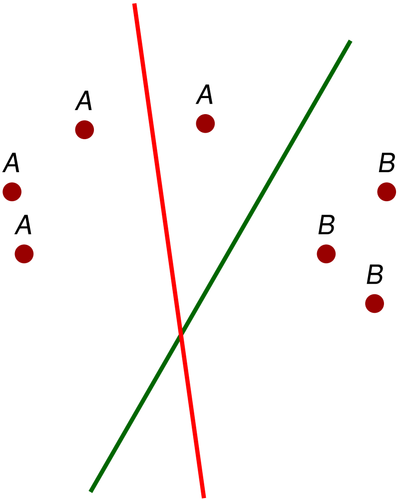
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.
- Unsupervised learning
  - The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

Supervised learning – illustration



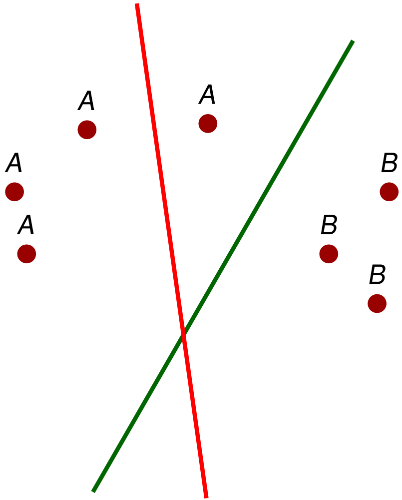
- classification in the plane using a single neuron

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

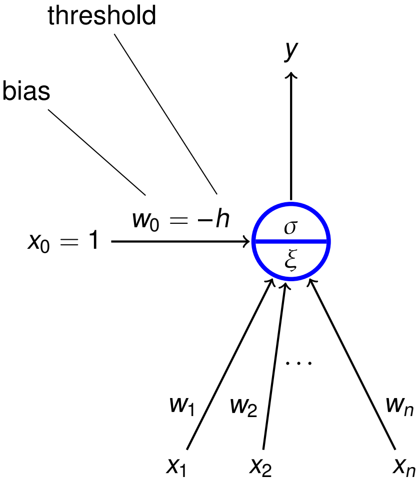
- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks
- generalization and robustness
  - information is encoded in a distributed manned in weights
  - "close" inputs typicaly get similar values

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*

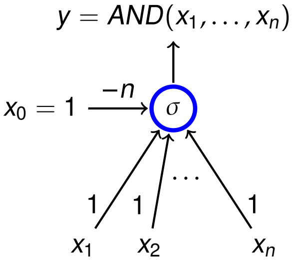
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

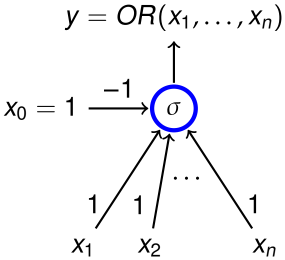
Boolean functions

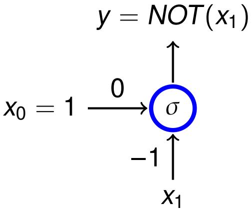
Activation function: *unit step function* $\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$

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

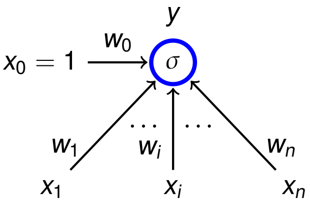
Boolean functions

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 $\vec{v}$:

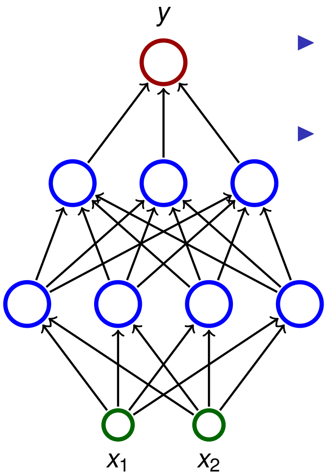


$$w_0 = -\sum_{i=1}^{n} v_i$$

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

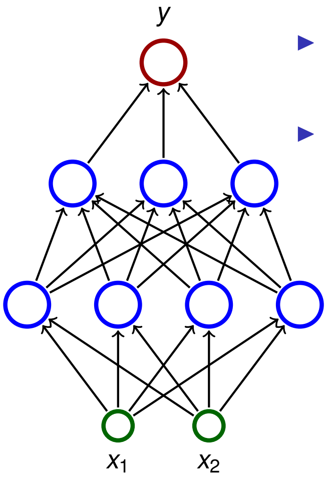
- Now let us connect all outputs of all neurons $N_{\vec{v}}$ satisfying $F(\vec{v}) = 1$ using a neuron implementing OR. □

Non-linear separation



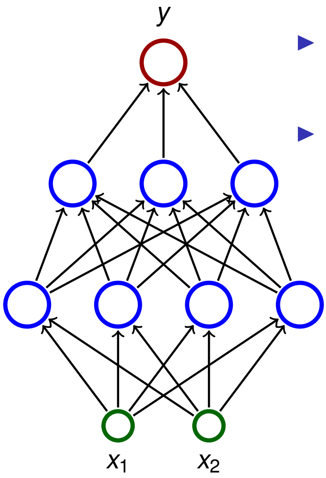
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).

Non-linear separation



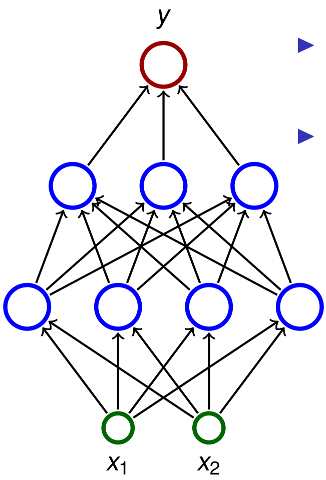
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.

Non-linear separation



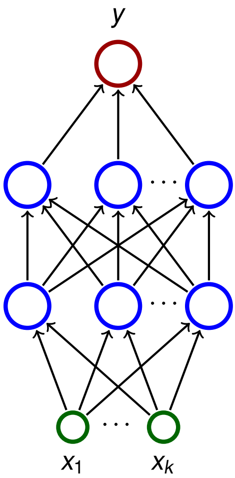
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation



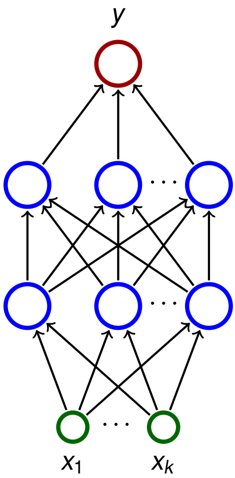
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation – illustration



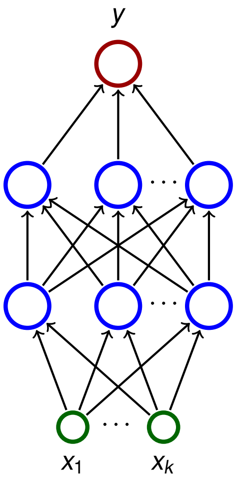
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.

Non-linear separation – illustration



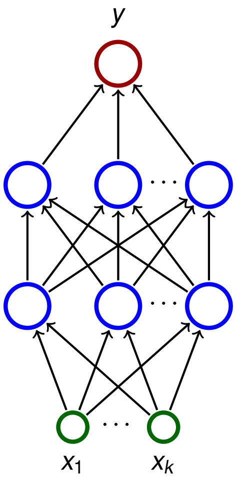
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)

Non-linear separation – illustration



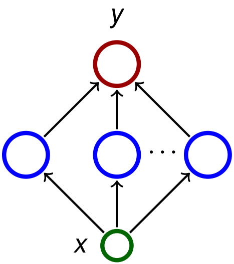
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)
  - Each hypercube K can be separated using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

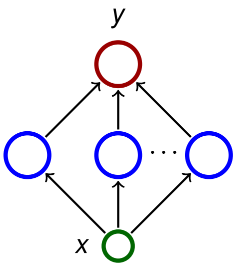
Power of ReLU



Consider a two layer network

- with a single input and single output;
- hidden neurons with the ReLU activation: $\sigma(\xi) = \max(\xi, 0)$;
- the output neuron with identity activation: $\sigma(\xi) = \xi$ (linear model)

Power of ReLU

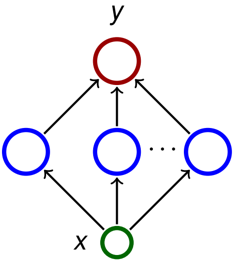


Consider a two layer network

- with a single input and single output;
- hidden neurons with the ReLU activation: $\sigma(\xi) = \max(\xi, 0)$;
- the output neuron with identity activation: $\sigma(\xi) = \xi$ (linear model)

For every continuous function $f : [0,1] \to [0,1]$ and $\varepsilon > 0$ there is a network of the above type computing a function $F : [0,1] \to \mathbb{R}$ such that $|f(x) - F(x)| \leq \varepsilon$ for all $x \in [0,1]$.

Power of ReLU



Consider a two layer network

- with a single input and single output;
- hidden neurons with the ReLU activation: $\sigma(\xi) = \max(\xi, 0)$;
- the output neuron with identity activation: $\sigma(\xi) = \xi$ (linear model)

For every continuous function $f : [0,1] \to [0,1]$ and $\varepsilon > 0$ there is a network of the above type computing a function $F : [0,1] \to \mathbb{R}$ such that $|f(x) - F(x)| \leq \varepsilon$ for all $x \in [0,1]$.

For every open subset $A \subseteq [0,1]$ there is a network of the above type such that for "most" $x \in [0,1]$ we have that $x \in A$ iff the network's output is > 0 for the input x.

Just consider a continuous function f where $f(x)$ is the minimum difference between x and a point on the boundary of A. Then uniformly approximate f using the networks.

Non-linear separation - sigmoid

Theorem (Cybenko 1989 - informal version)

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

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

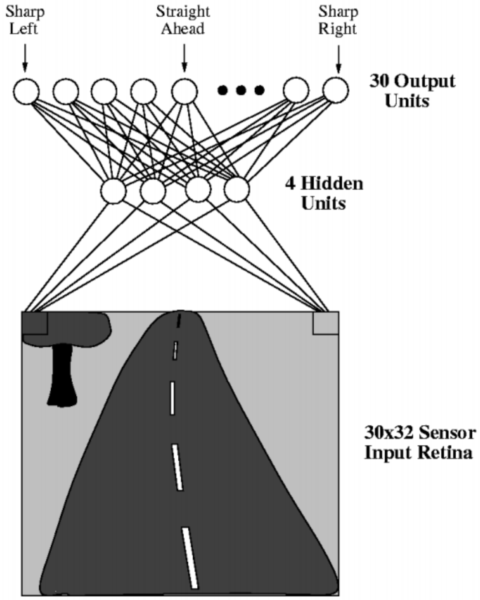
For every "reasonable" set $A \subseteq [0,1]^n$, there is a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following:*

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:

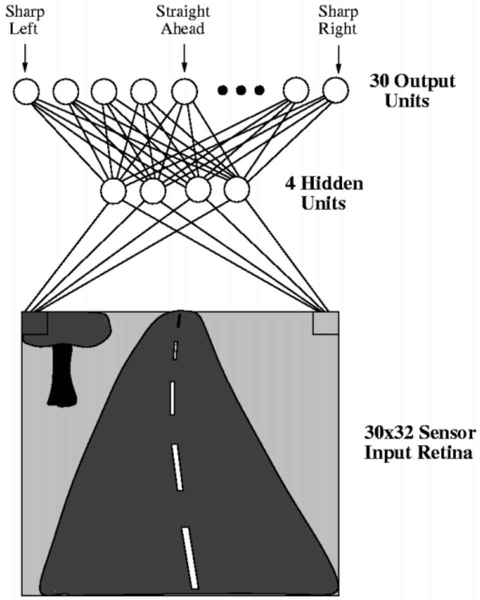
- "reasonable" means Lebesgue measurable
- "most" means that the set of incorrectly classified vectors has the Lebesgue measure smaller than a given $\varepsilon > 0$

Non-linear separation - practical illustration



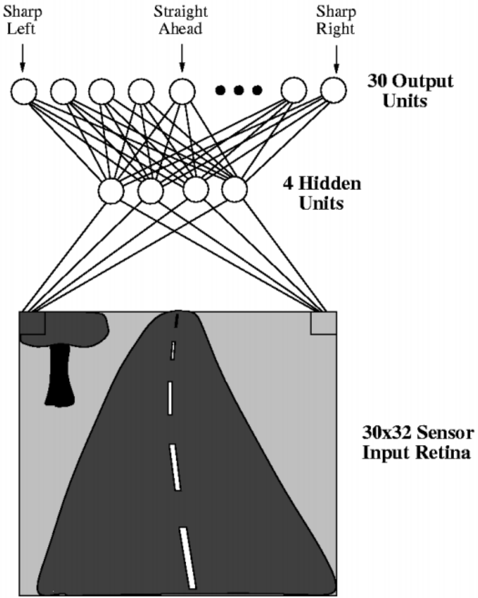
- ALVINN drives a car

Non-linear separation - practical illustration



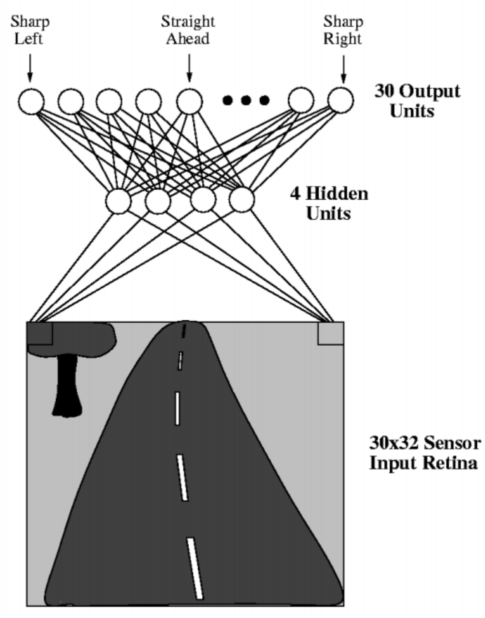
- ALVINN drives a car
- The net has $30 \times 32 = 960$ inputs (the input space is thus $\mathbb{R}^{960}$)

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 - two-layer networks

Theorem (Cybenko 1989)

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

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

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

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

Neural networks and computability

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

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.

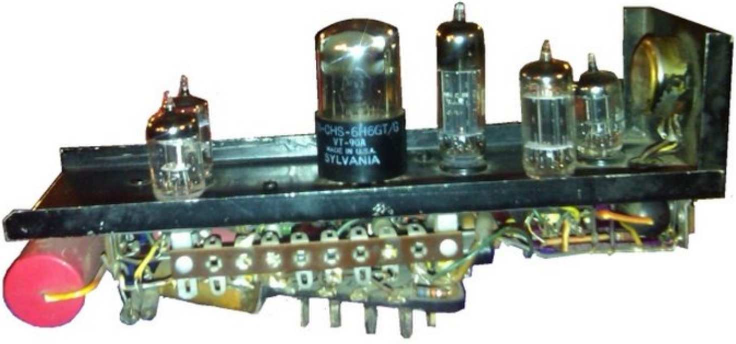
Neural networks vs classical computers

	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.)



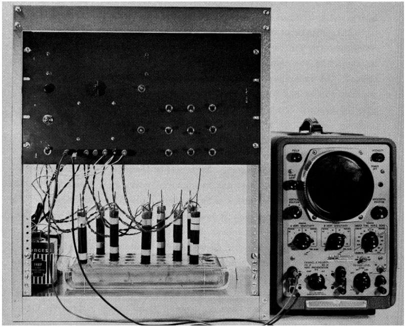
History of neurocomputers

- 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
  - ... specialized hw implementations (Google's TPU)

Some highlights

- Breakthrough in image recognition.
  Accuracy of image recognition improved by an order of magnitude in 5 years.
- Breakthrough in game playing.
  Superhuman results in Go and Chess almost without any human intervention. Master level in Starcraft, poker, etc.
- Breakthrough in machine translation.
  Switching to deep learning produced a 60% increase in translation accuracy compared to the phrase-based approach previously used in Google Translate (in human evaluation)
- Breakthrough in speech processing.
- Breakthrough in text generation.
  GPT-3 generates pretty realistic articles, short plays (for a theatre) have been successfully generated, etc.

History in waves ...

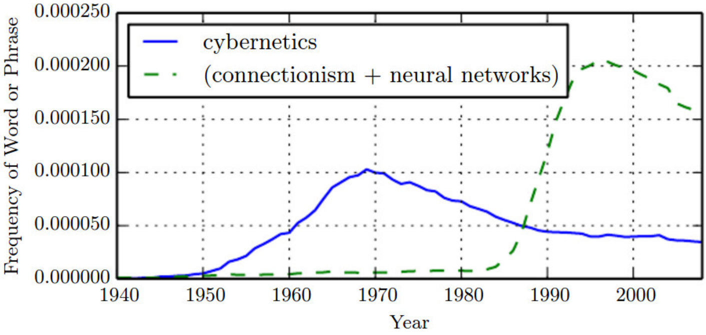
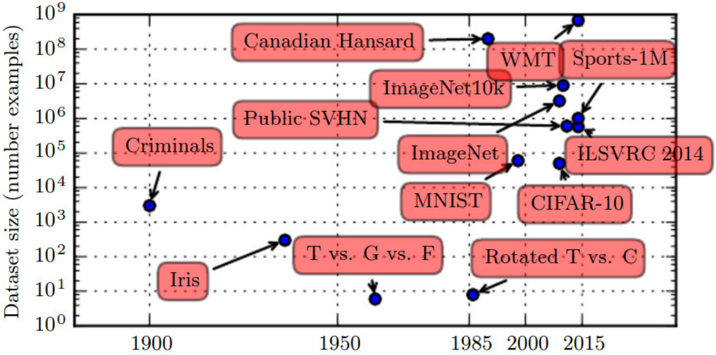


Figure: The figure shows two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks" according to Google Books (the third wave is too recent to appear).

Current hardware – What do we face?

Increasing dataset size ...



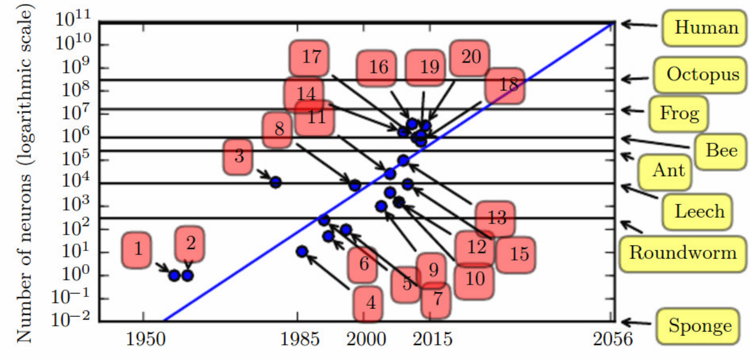
... weakly-supervised pre-training using hashtags from the Instagram uses $3.6 * 10^9$ images.

Revisiting Weakly Supervised Pre-Training of Visual Perception Models. Singh et al.

https://arxiv.org/pdf/2201.08371.pdf, 2022

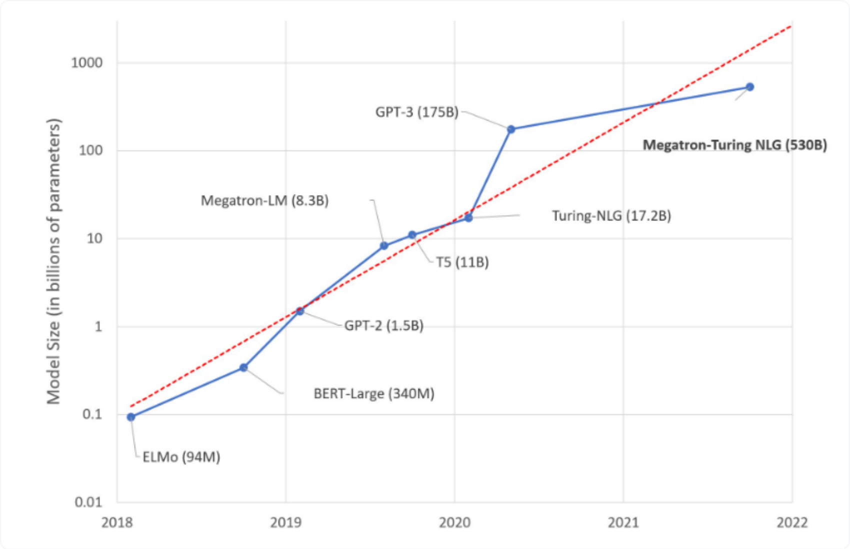
Current hardware – What do we face?

... 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?



Current hardware – What do we face?

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

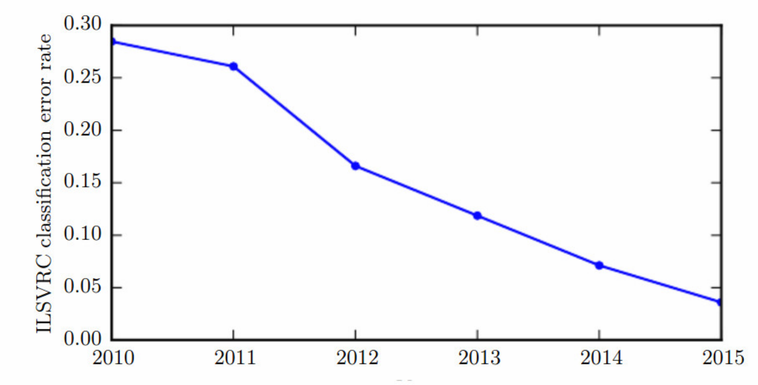


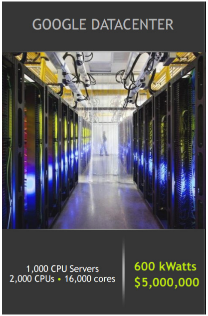
Figure: Since deep networks reached the scale necessary to compete in the ImageNetLarge Scale Visual Recognition Challenge, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky et al. (2014b) and He et al. (2015).

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

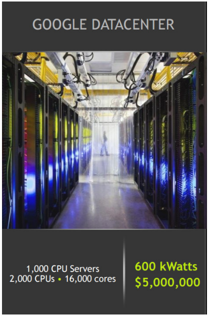
In 2012, Google trained a large network of 1.7 billion weights and 9 layers

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

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

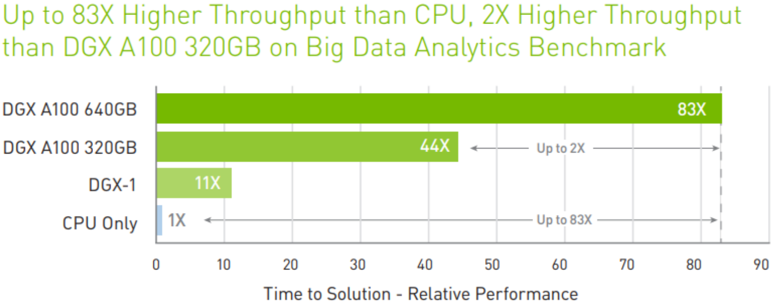
In 2014, similar task performed on Commodity Off-The-Shelf High Performance Computing (COTS HPC) technology: a cluster of GPU servers with Infiniband interconnects and MPI.

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

- 8x GPU (Nvidia A100 80GB Tensor Core)
- 5 petaFLOPS
- System memory: 2 TB
- Network: 200 Gb/s InfiniBand



Deep learning in clouds

Big companies offer cloud services for deep learning:

- Amazon Web Services
- Google Cloud
- Deep Cognition
- ...

Advantages:

- Do not have to care (too much) about technical problems.
- Do not have to buy and optimize highend hw/sw, networks etc.
- Scaling & virtually limitless storage.

Disadvatages:

- Do not have full control.
- Performance can vary, connectivity problems.
- Have to pay for services.
- Privacy issues.

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 part of TensorFlow that allows easy description of most modern neural networks
- **PyTorch** (Facebook)
  - similar to TensorFlow
  - object oriented
  - ... majority of new models in research papers implemented in PyTorch

    https://www.cioinsight.com/big-data/pytorch-vs-tensorflow/
- **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, 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)
```

Current software – Keras functional API

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

Current software – TensorFlow

```
# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Current software – TensorFlow

```
# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = multilayer_perceptron(X)
```

Current software – PyTorch

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

net = Net(input_size, hidden_size, num_classes)
```

Other software implementations

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

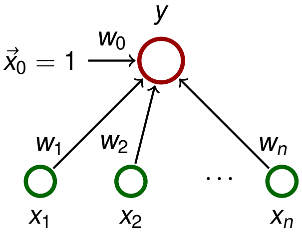
- MATLAB
- R
- STATISTICA
- Weka
- ...

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**

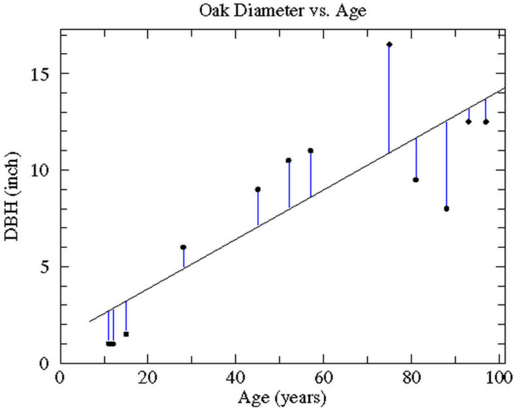
$$\mathcal{T} = \left\{\left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right)\right\}$$

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

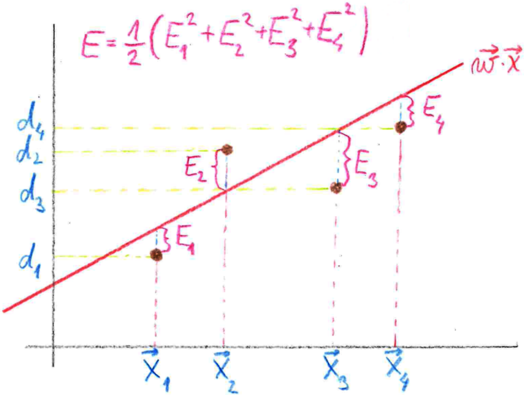
Age (years)	*DBH* (inch)
97	12.5
93	12.5
88	8.0
81	9.5
75	16.5
57	11.0
52	10.5
45	9.0
28	6.0
15	1.5
12	1.0
11	1.0



Linear regression (ADALINE)

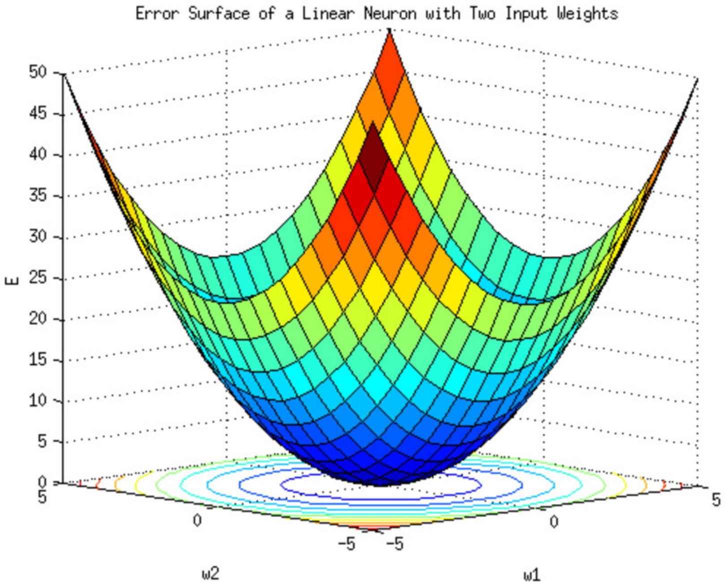
- **Error function:**

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



- 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.

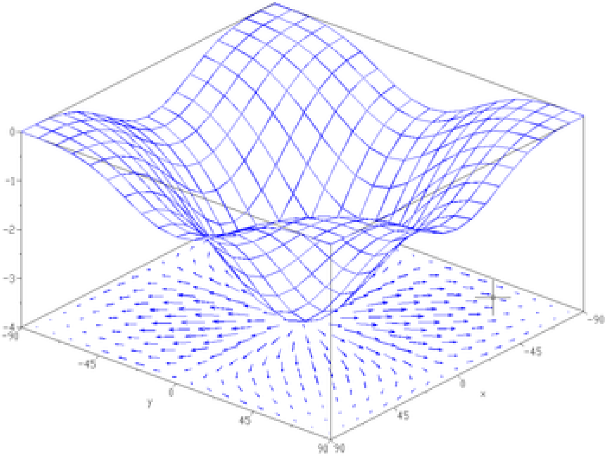
Note that the vectors $\vec{x}_k$ are just parameters of the function E, and are thus fixed!

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:

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

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

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:

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

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

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

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:

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

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

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

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:

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

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

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

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

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:

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

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

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

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

Gradient of the error function

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

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:

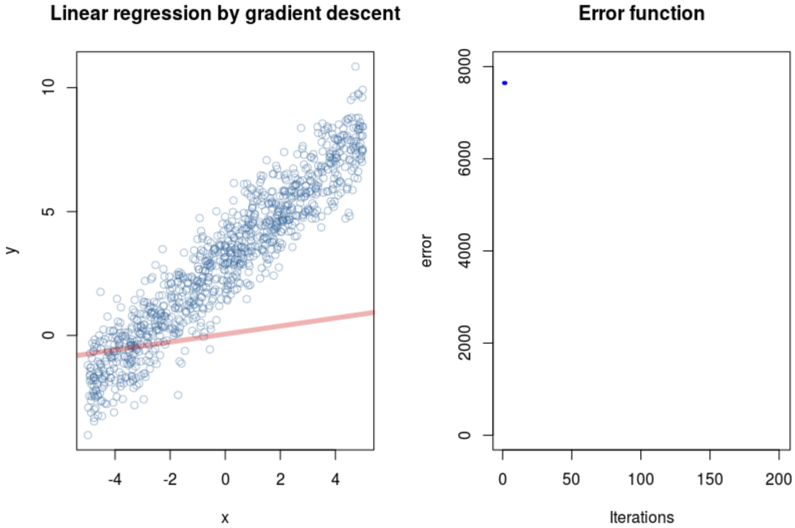
$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \vec{x}_k - d_k\right) \cdot \vec{x}_k\end{aligned}$$

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

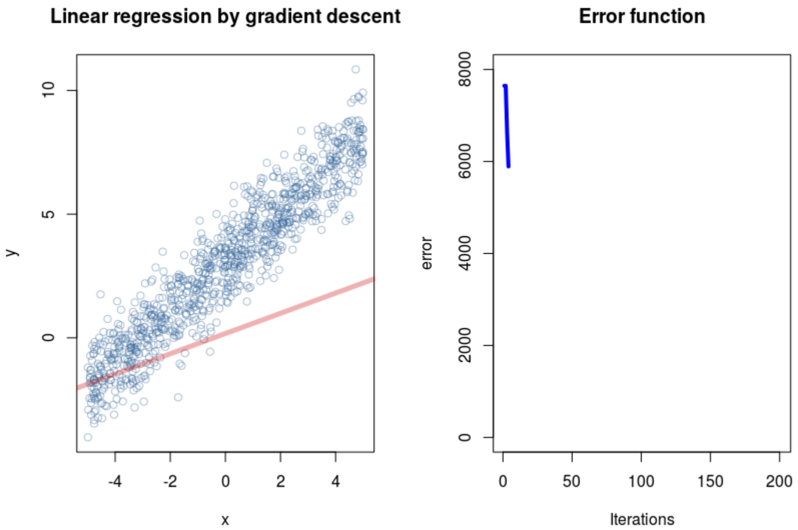
Proposition

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 $\nabla E(\vec{w}) = \vec{0}$).

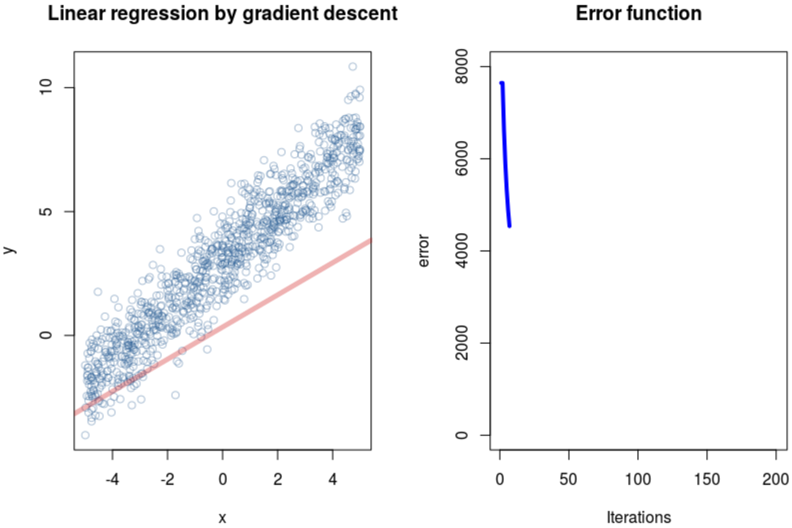
Linear regression - animation



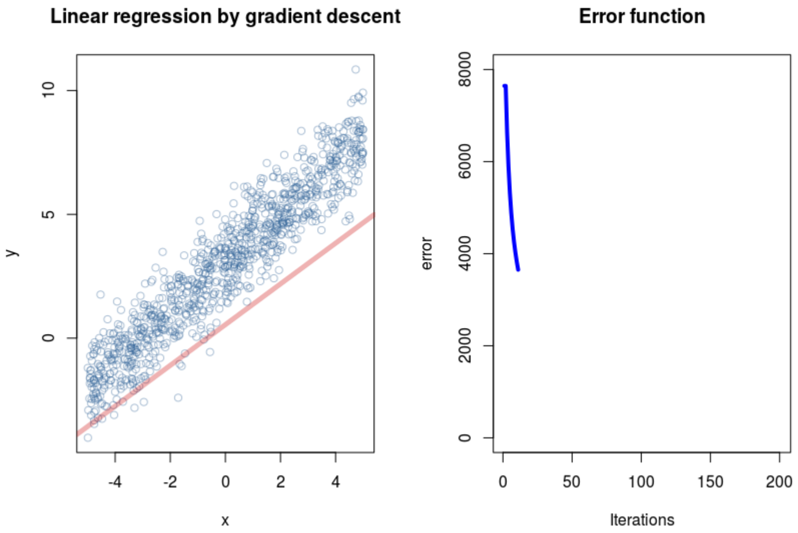
Linear regression - animation



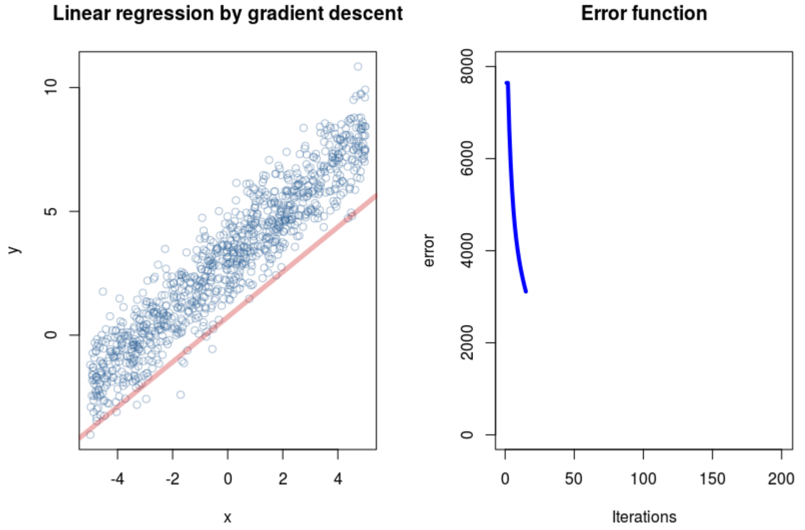
Linear regression - animation



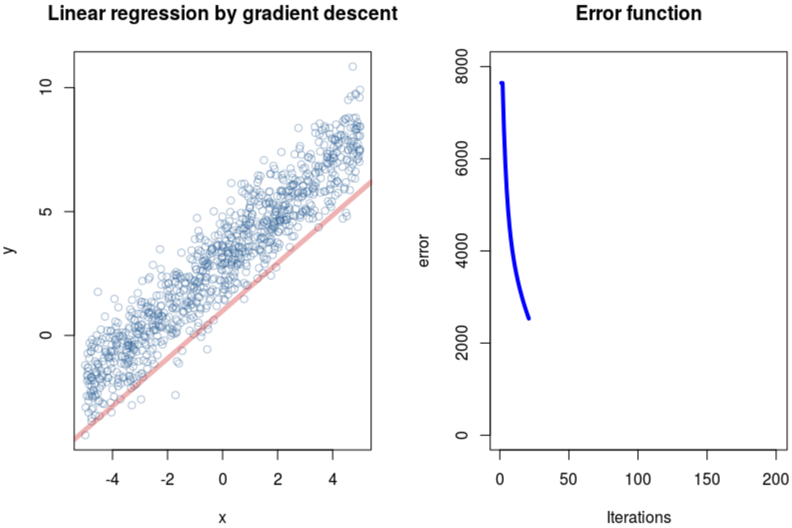
Linear regression - animation



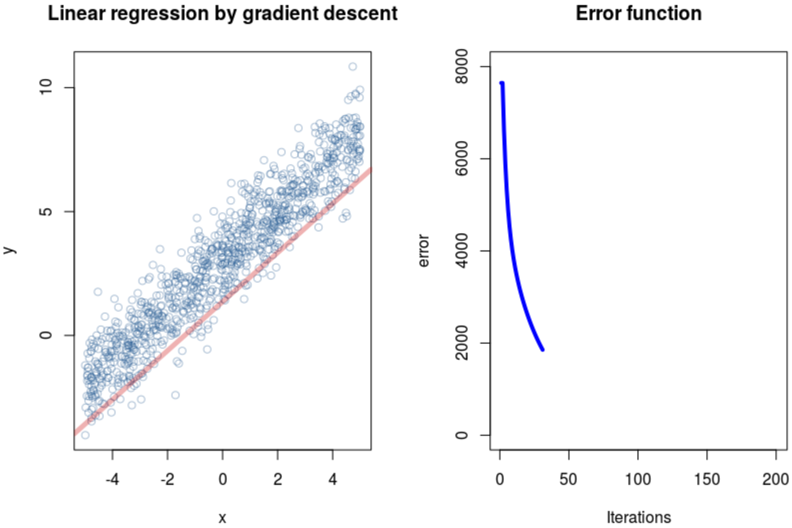
Linear regression - animation



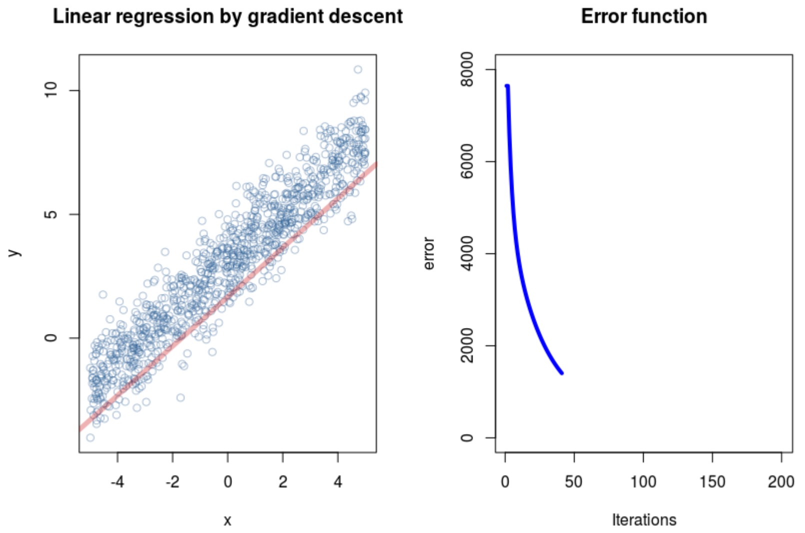
Linear regression - animation



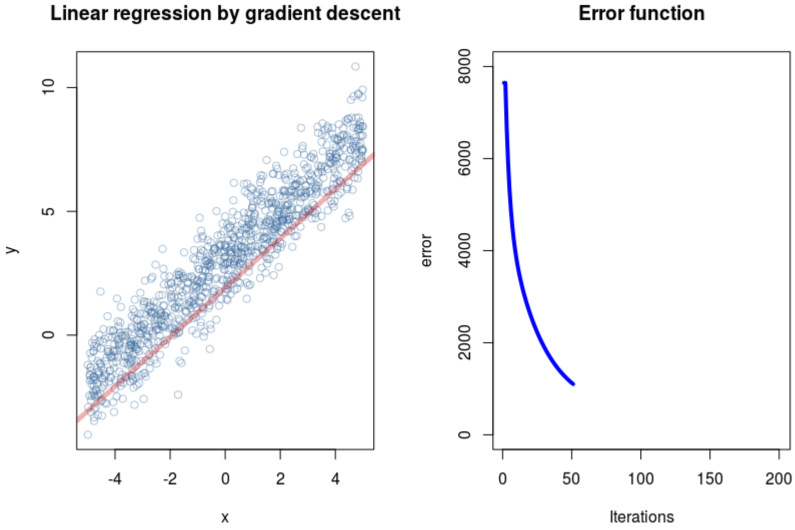
Linear regression - animation



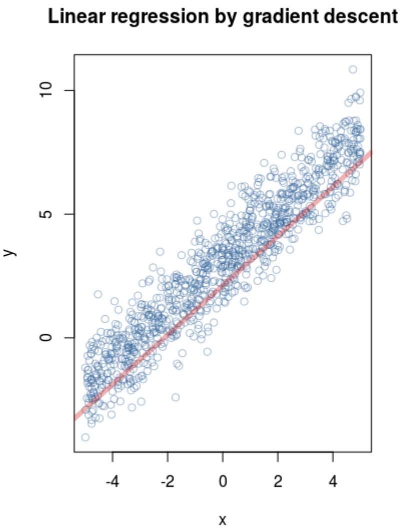
Linear regression - animation

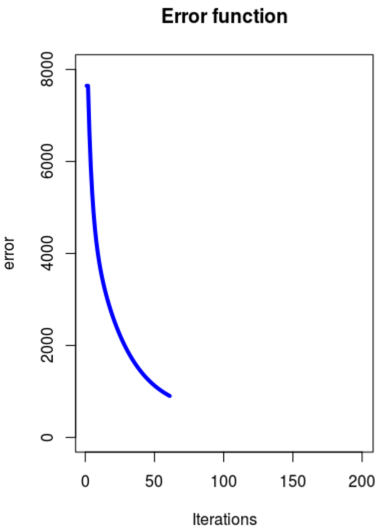


Linear regression - animation

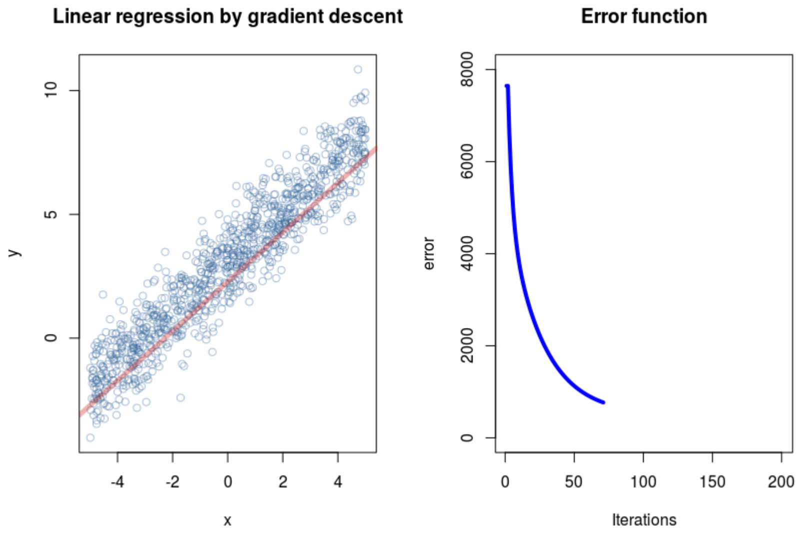


Linear regression - animation

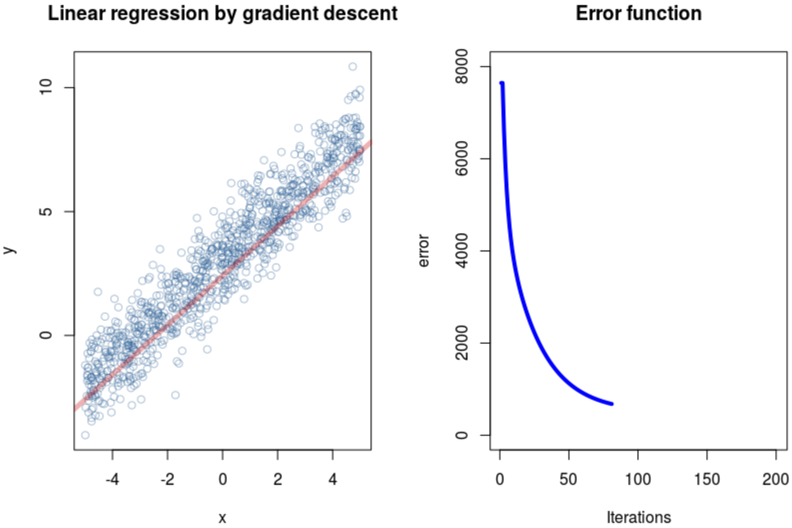




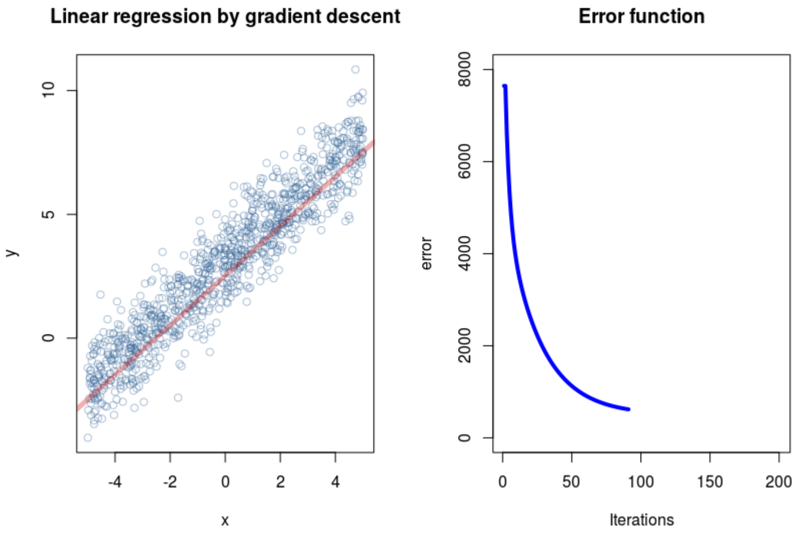
Linear regression - animation



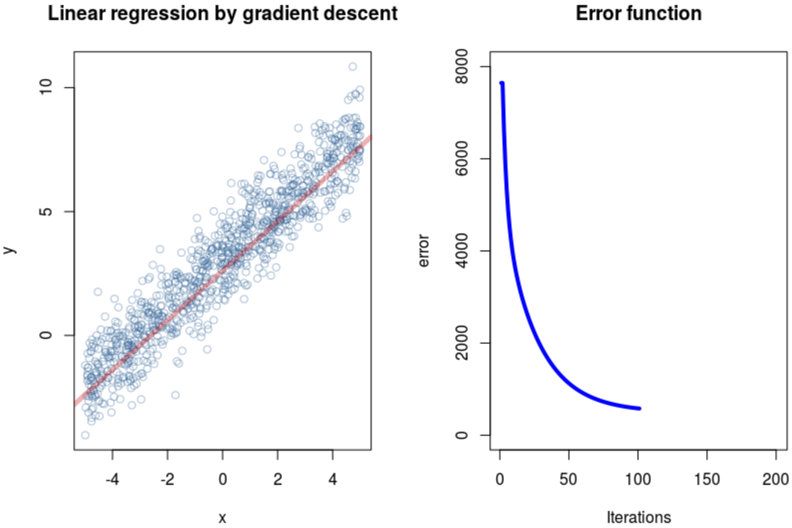
Linear regression - animation



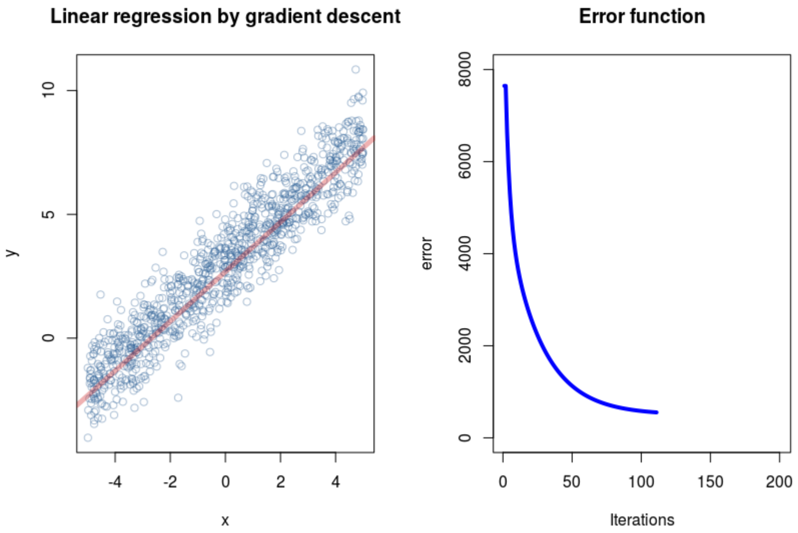
Linear regression - animation



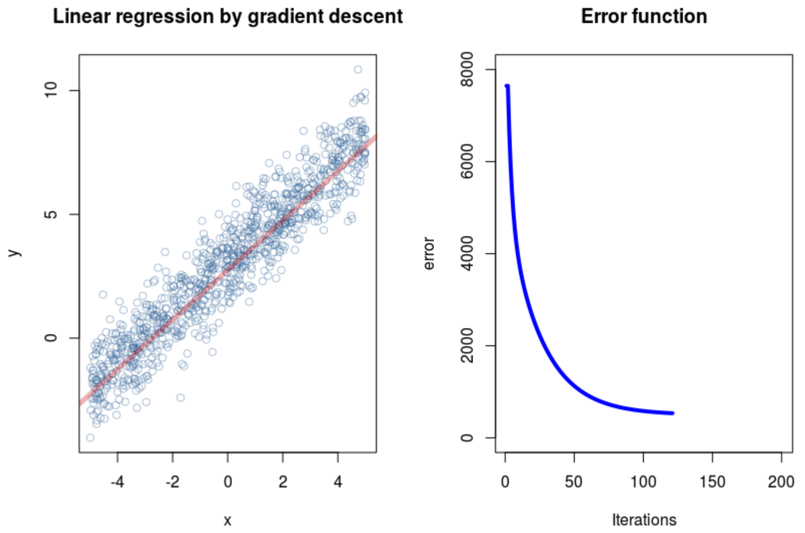
Linear regression - animation



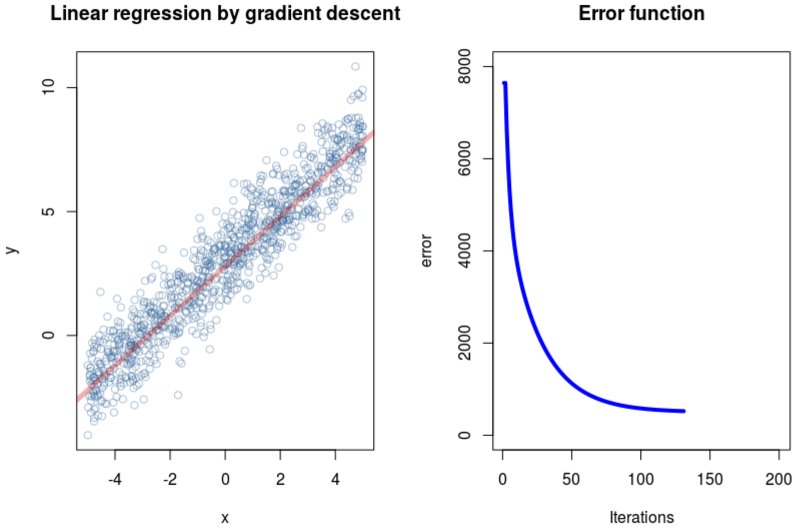
Linear regression - animation



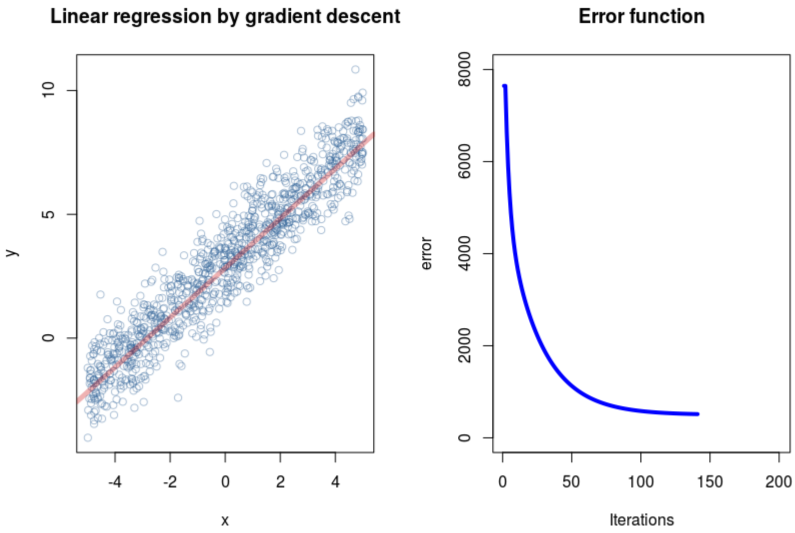
Linear regression - animation



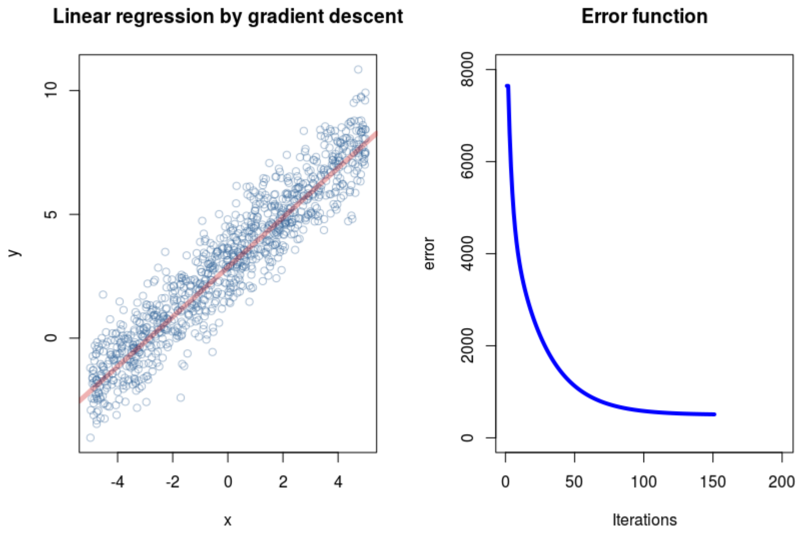
Linear regression - animation



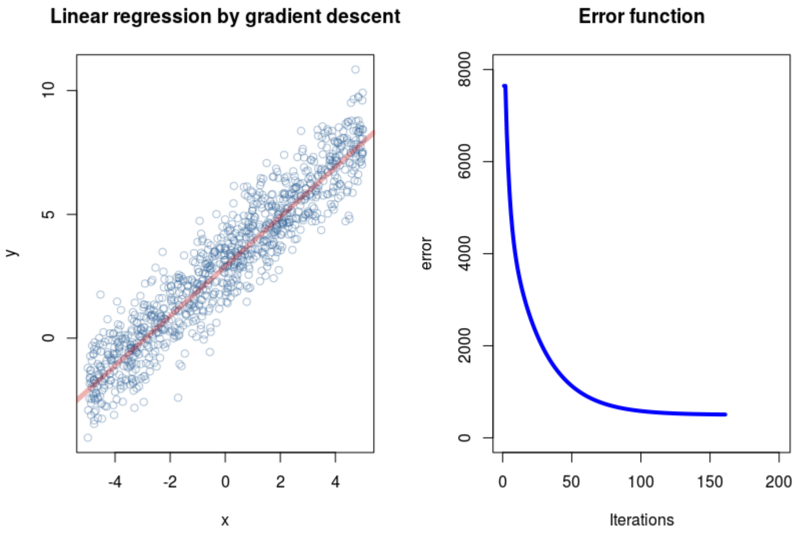
Linear regression - animation



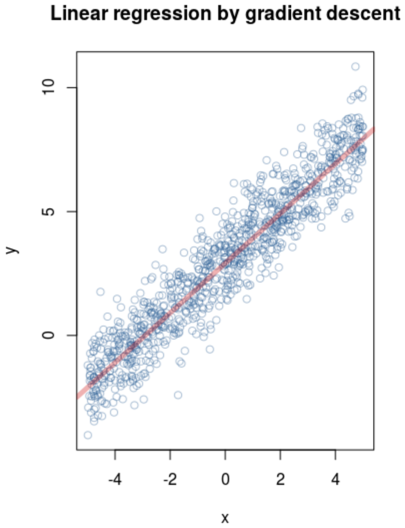
Linear regression - animation

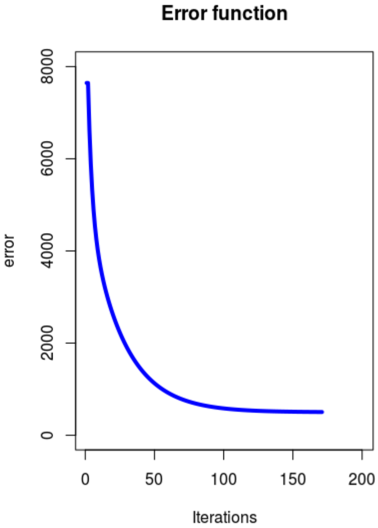


Linear regression - animation

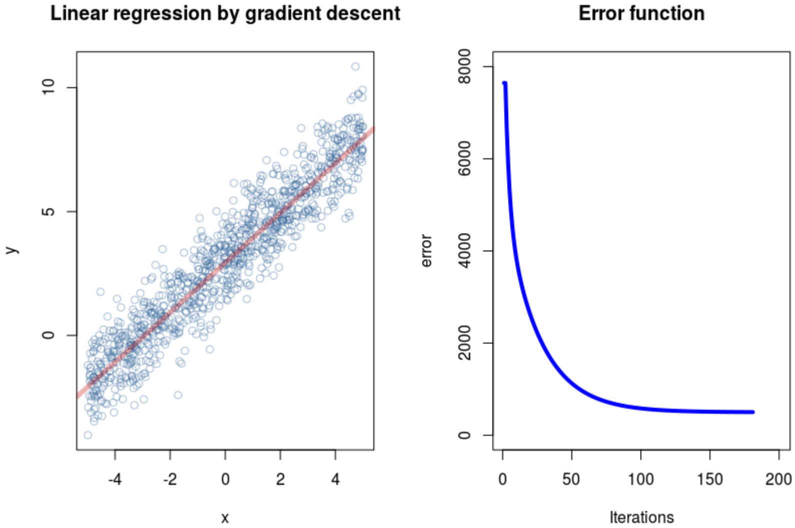


Linear regression - animation

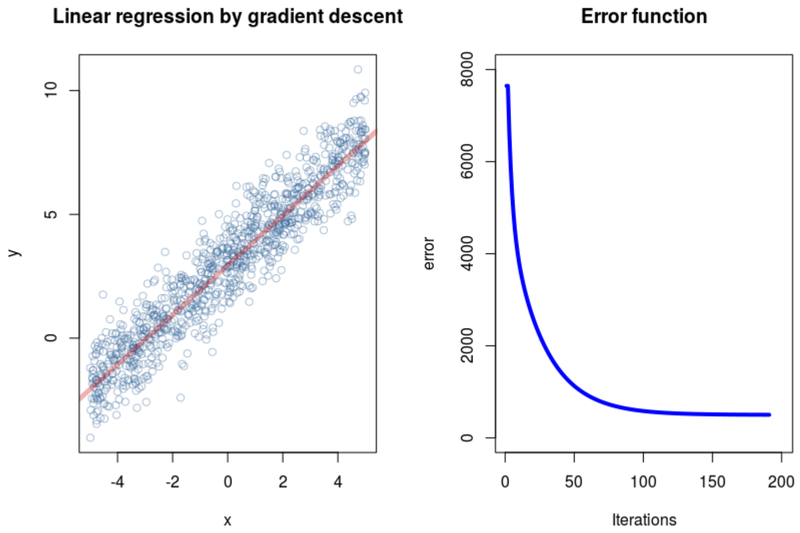




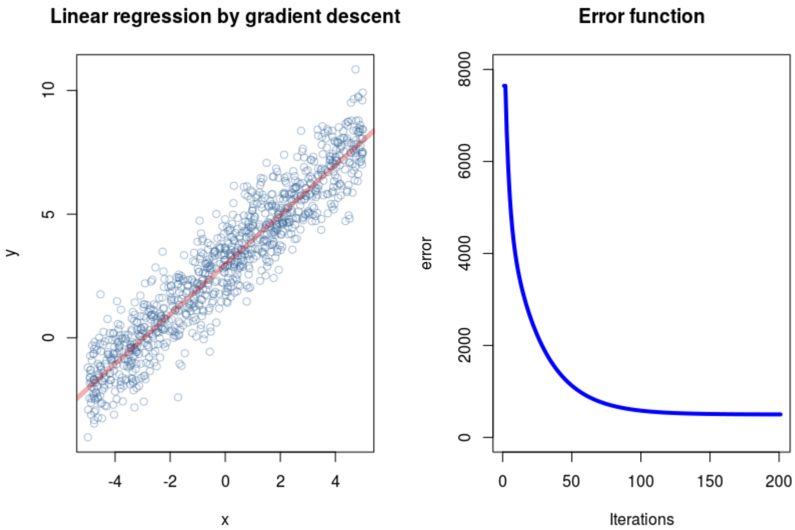
Linear regression - animation



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 $t+1$.

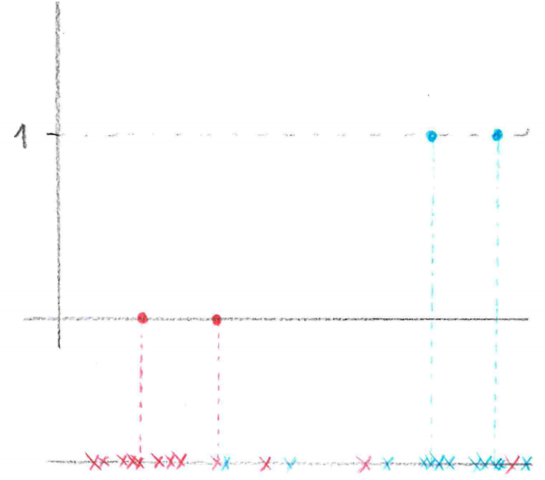
Note that the algorithm does not work with the complete gradient but only with its part determined by the currently considered training example.

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?

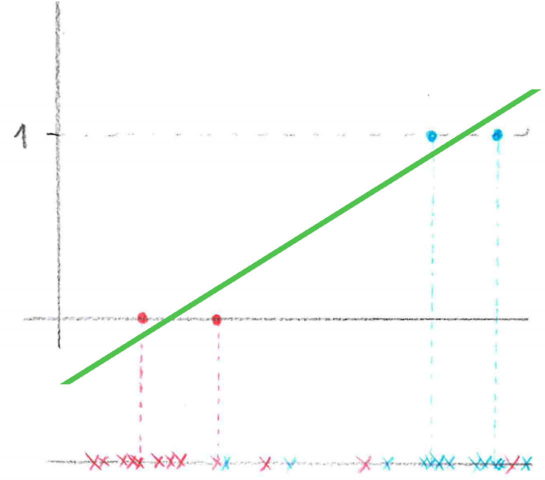
Binary classification: Desired outputs 0 and 1.



Ideally, capture the probability distribution of classes.

What about classification?

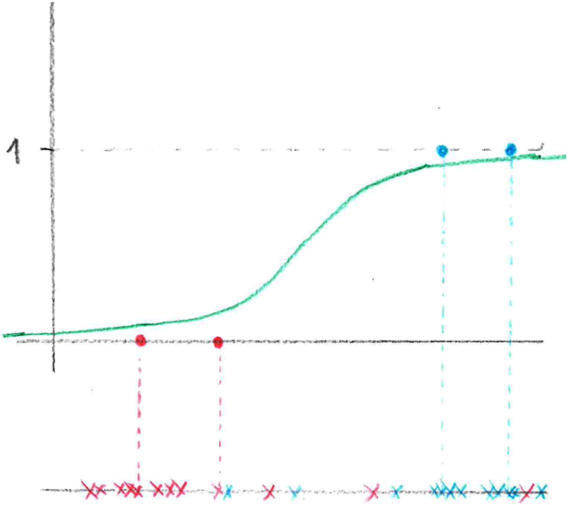
Binary classification: Desired outputs 0 and 1.



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

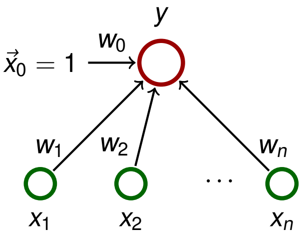
What about classification?

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 interpreted as the probability of the class 1 given the input $\vec{x}$.

But what is the meaning of the sigmoid?

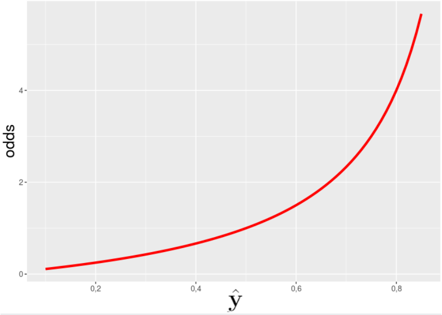
The model gives a probability y of the class 1 given an input $\vec{x}$.
But why we model such a probability using $1/(1+e^{-\vec{w}\cdot\vec{x}})$??

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 $1/(1+e^{-\vec{w}\cdot\vec{x}})$??

Let $\hat{y}$ be the "true" probability of the class 1 to be modeled.
What about odds of the class 1?

$$odds(\hat{y}) = \hat{y}/1-\hat{y}$$



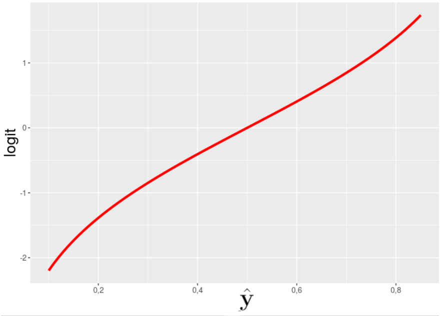
Resembles an exponential function ...

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 $1/(1 + e^{-\vec{w} \cdot \vec{x}})$??

Let $\hat{y}$ be the "true" probability of the class 1 to be modeled.
What about log odds (aka logit) of the class 1?

$$logit(\hat{y}) = \log(\hat{y}/(1 - \hat{y}))$$



Looks almost linear ...

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

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

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

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

Then

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

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

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

Then

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

and

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

But what is the meaning of the sigmoid?

Assume that $\hat{y}$ is the probability of the class 1. Put

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

Then

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

and

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

and

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

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 $\hat{y}$ and is unknown!

Log likelihood is your friend!

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

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

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

- What is the best model y of $\hat{y}$ based on the data?

Log likelihood is your friend!

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\hat{y}$ 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 $\hat{y}$ based on the data?

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

Log likelihood is your friend!

Keep in mind our dataset:

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

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 $\mathcal{T}$ from our model 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 $\mathcal{T}$ from our model 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 $\mathcal{T}$ from our model 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

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

i.e. $-LL$ 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)

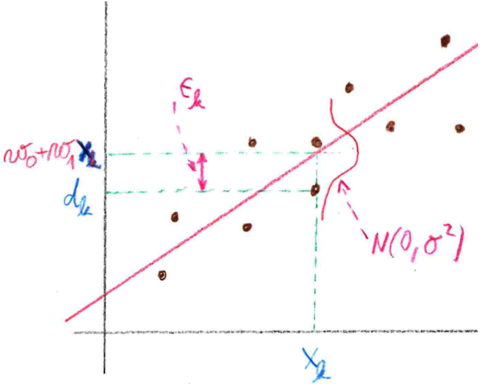
Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$

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 $\epsilon_1, \dots, \epsilon_p$ were generated **independently**.

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

The independence and normality imply

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

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 squared error* $E(w_0, w_1) = \sum_{k=1}^{p}(d_k - w_0 - w_1 x_k)^2$.

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 squared error* $E(w_0, w_1) = \sum_{k=1}^{p}(d_k - w_0 - w_1 x_k)^2$.

Note that maximizing $L(w_0, w_1, \sigma^2)$ w.r.t. (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 squared error* $E(w_0, w_1) = \sum_{k=1}^{p}(d_k - w_0 - w_1 x_k)^2$.

Note that maximizing $L(w_0, w_1, \sigma^2)$ w.r.t. (w_0, w_1) does not depend on σ^2.

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