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

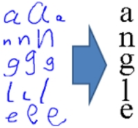
- Machine learning = construction of systems that may learn their functionality from data

  (... 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

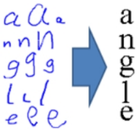
  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labelled" emails
    - consequently is able to distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labelled by their correct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

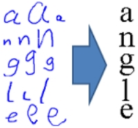


  - . . .
  - and lots of much much more sophisticated applications ...

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.

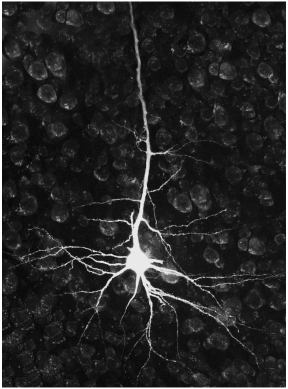
There are many types of models:

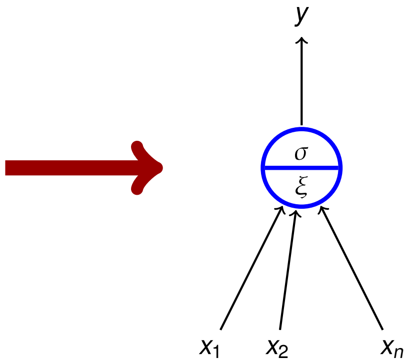
- decision trees
- support vector machines
- hidden Markov models
- Bayes networks and other graphical models
- **neural networks**
- ...

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.

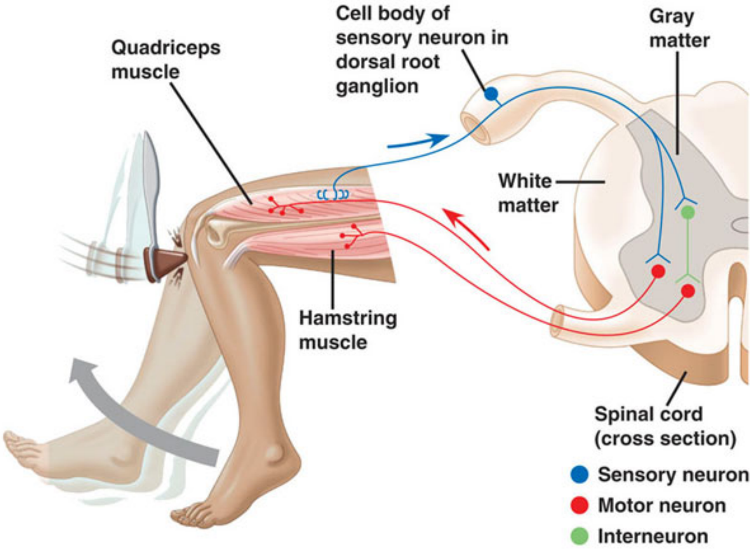
Biological neural network

- 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

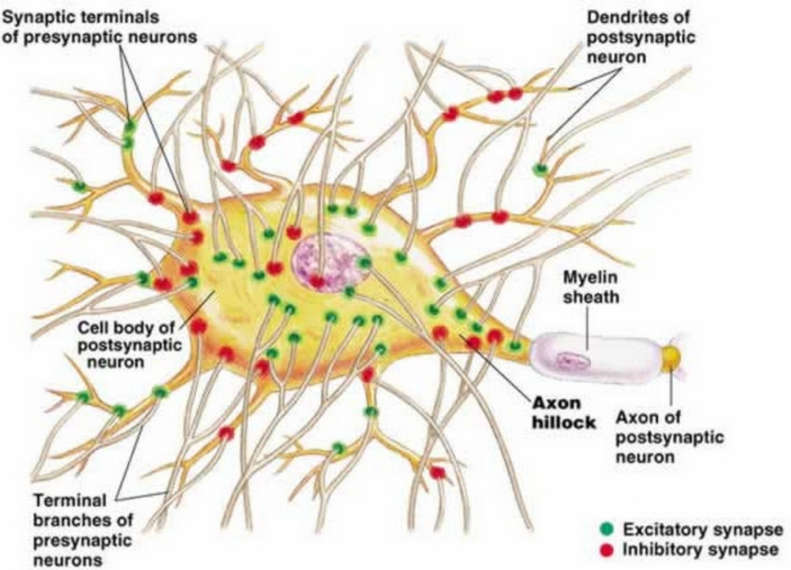
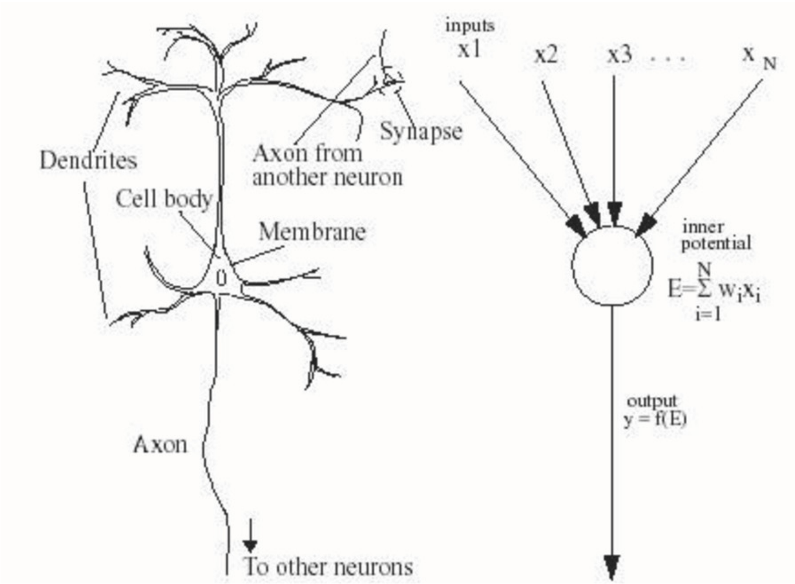
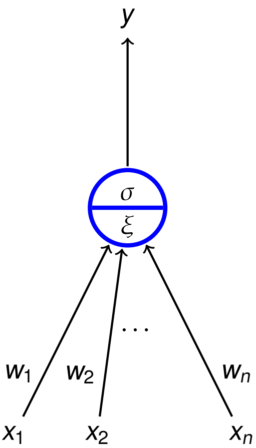


Figure 48.11(a), page 972, Campbell's *Biology, 5th Edition*

Biological and Mathematical neurons

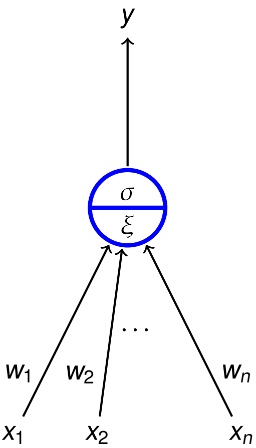


Formal neuron (without bias)



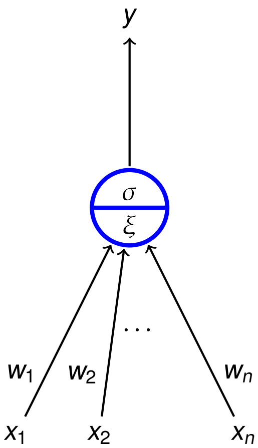
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**

Formal neuron (without bias)



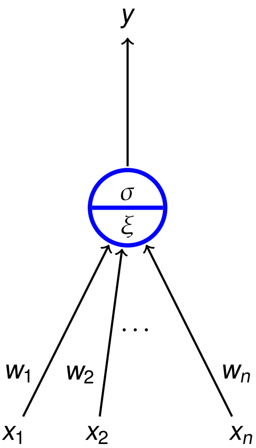
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

Formal neuron (without bias)



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

Formal neuron (without bias)



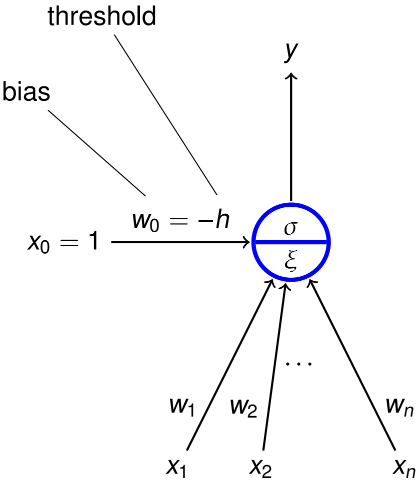
- $x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$
  where σ is an **activation function**;
  e.g. a *unit step function*

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

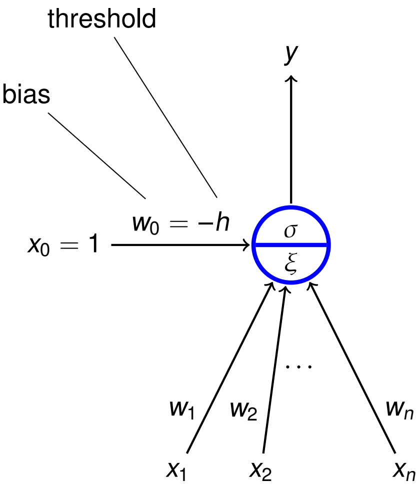
where $h \in \mathbb{R}$ is a *threshold*.

Formal neuron (with bias)

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

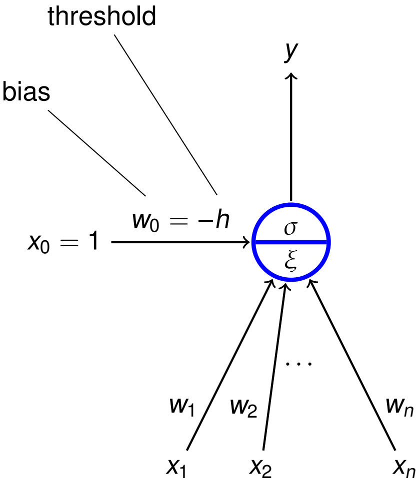


Formal neuron (with bias)



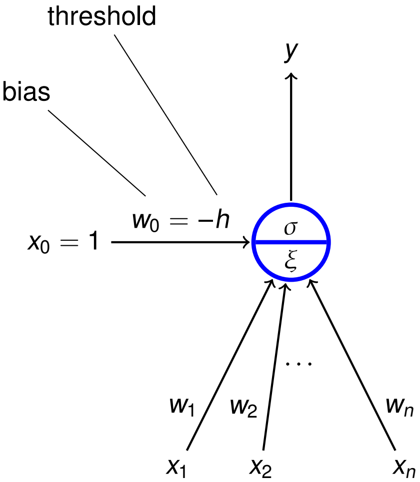
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

Formal neuron (with bias)



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

Formal neuron (with bias)



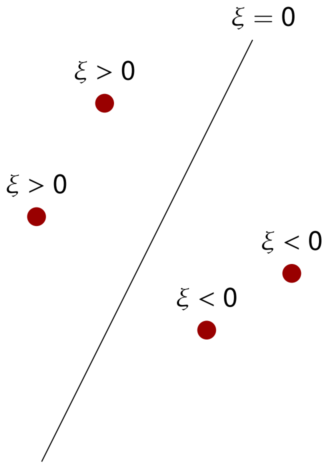
- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$
  where σ is an **activation function**;

  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



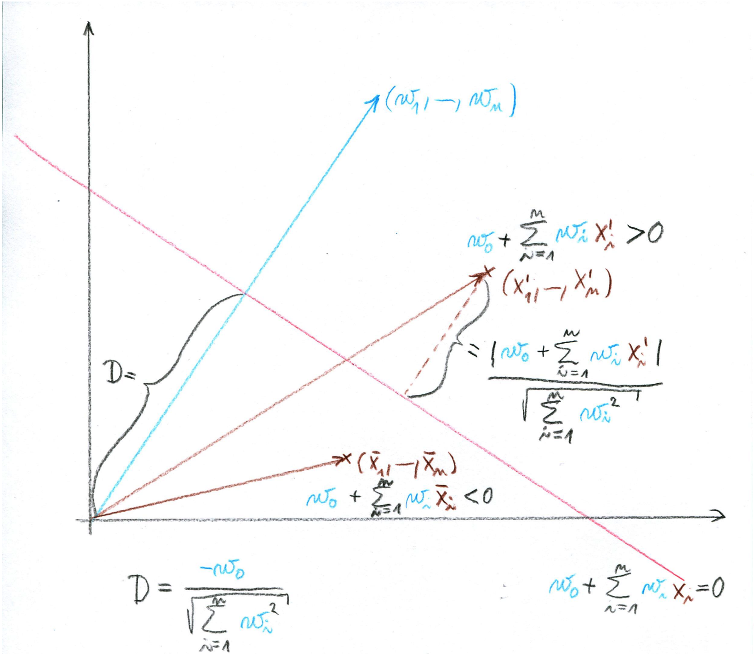
- inner potential

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

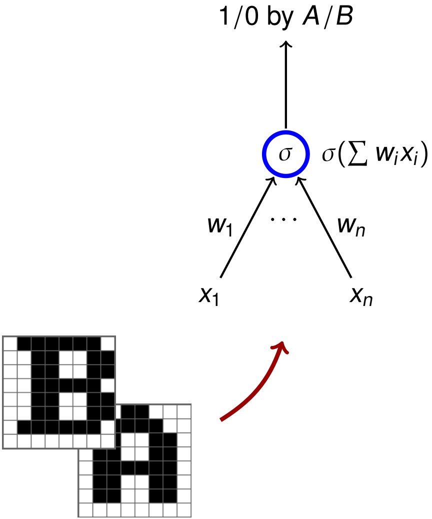
determines a separation hyperplane in the n-dimensional **input space**

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

Neuron geometry

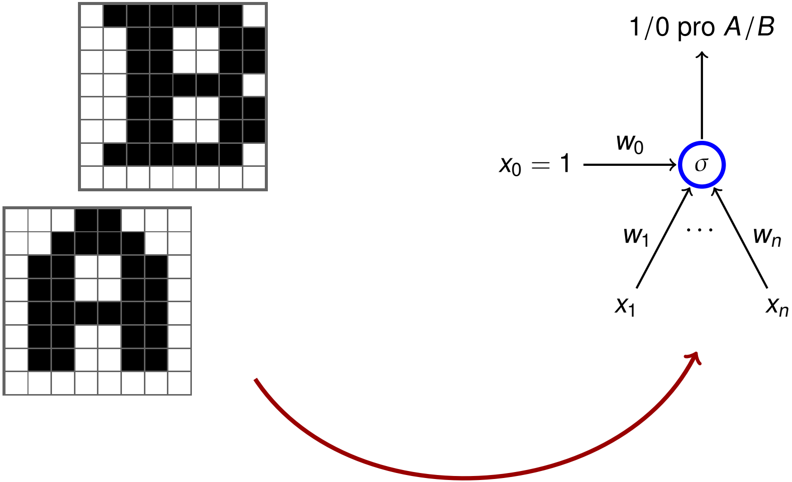


Neuron and linear separation



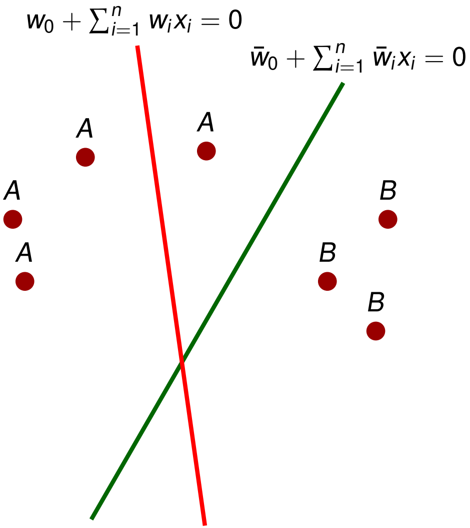
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

Neuron and linear separation



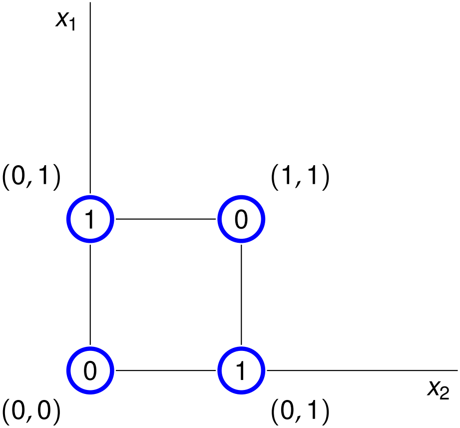
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

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.

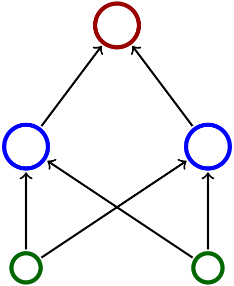
Architecture

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

We distinguish several categories of neurons:

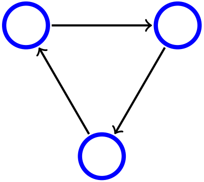
- Output neurons
- Hidden neurons
- Input neurons

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



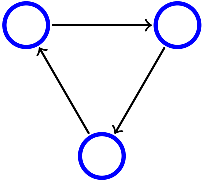
Architecture – Cycles

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

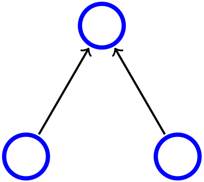


Architecture – Cycles

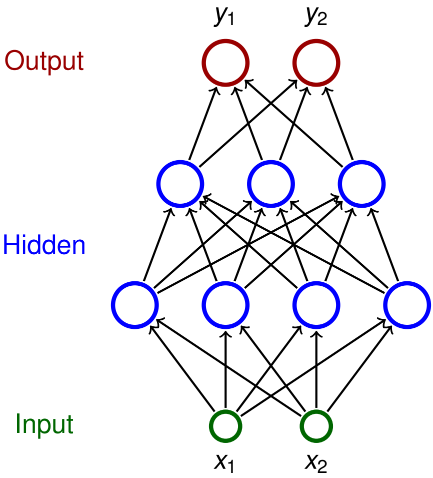
- A network is **cyclic** (recurrent) if its architecture contains a directed cycle.



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

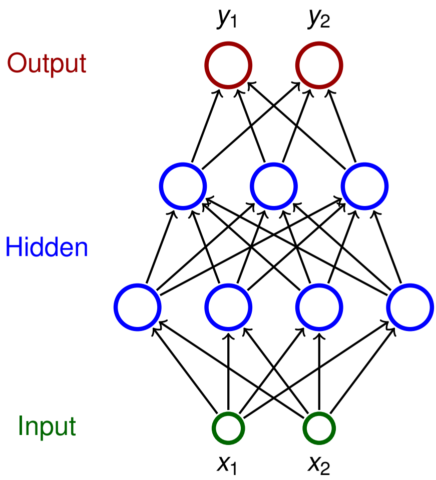


Architecture – Multilayer Perceptron (MLP)



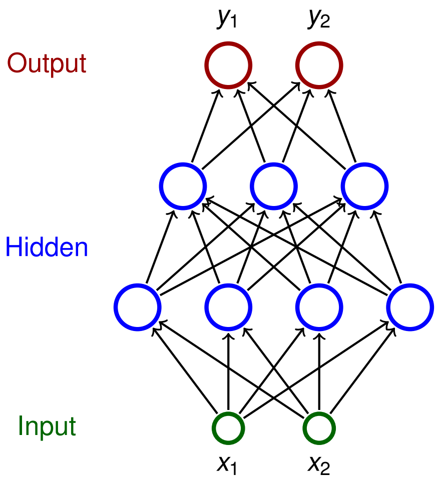
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers

Architecture – Multilayer Perceptron (MLP)



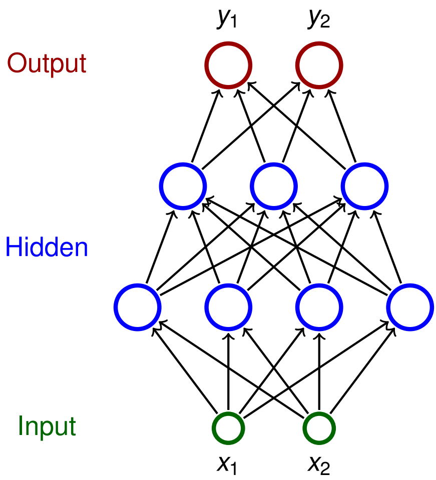
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer

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 $F : A \rightarrow B$ *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.*

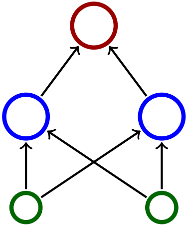
Activity – semantics of a network

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.

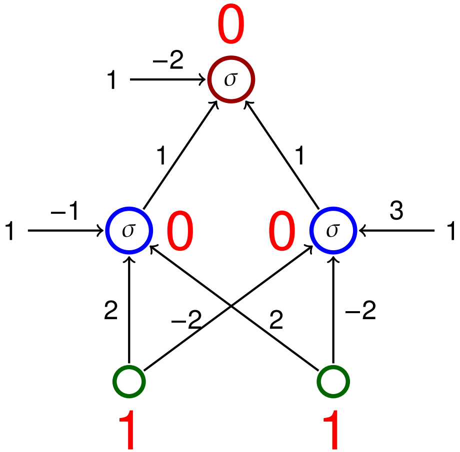
- Hyperbolic tangens

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

- ReLU

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

Activity – XOR



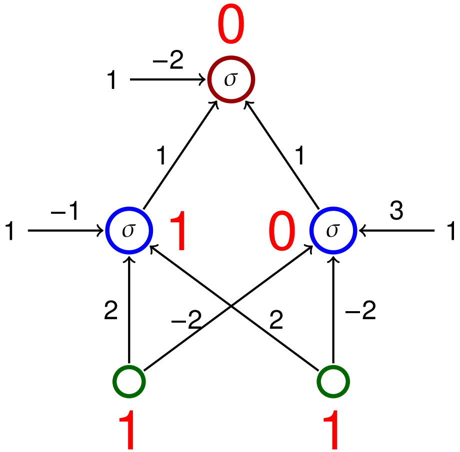
- Activation function is a unit step function

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

- 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



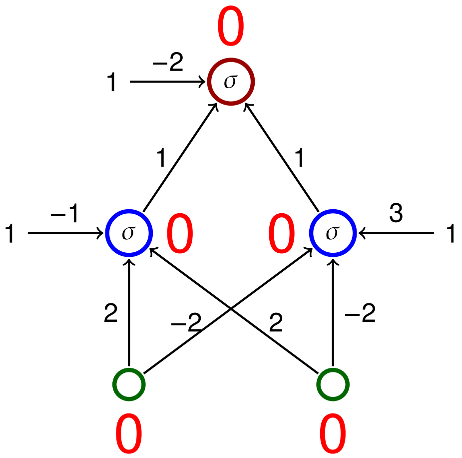
- Activation function is a unit step function

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

- 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



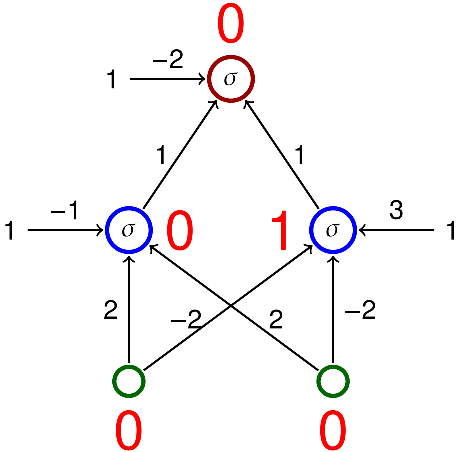
- Activation function is a unit step function

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

- 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



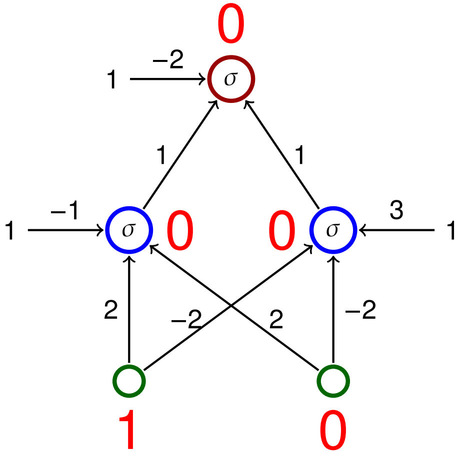
- Activation function is a unit step function

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

- 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



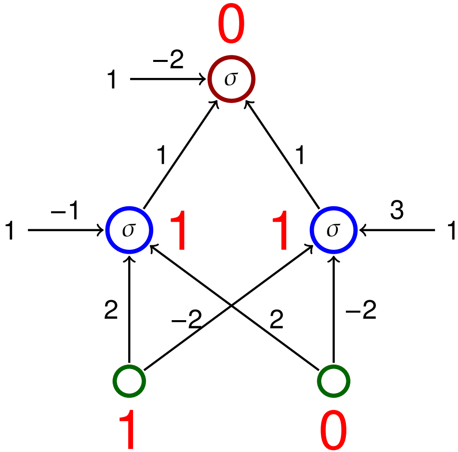
- Activation function is a unit step function

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

- 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



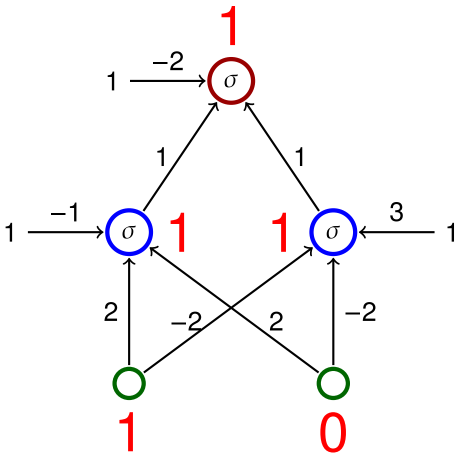
- Activation function is a unit step function

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

- 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



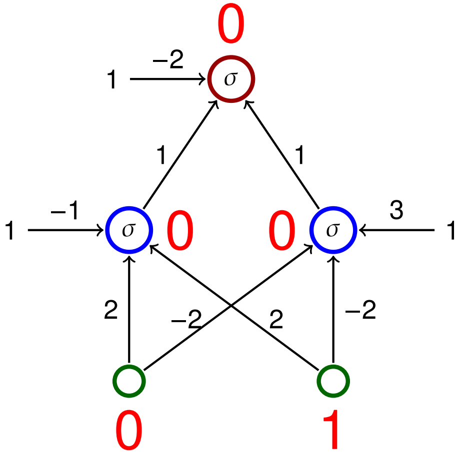
- Activation function is a unit step function

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

- 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



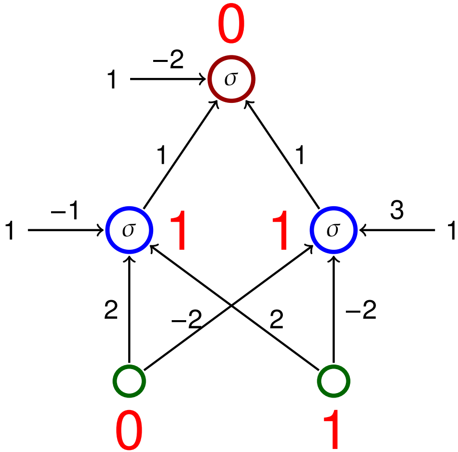
- Activation function is a unit step function

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

- 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



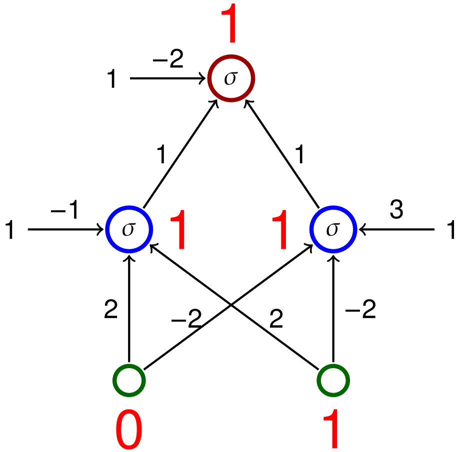
- Activation function is a unit step function

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

- 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



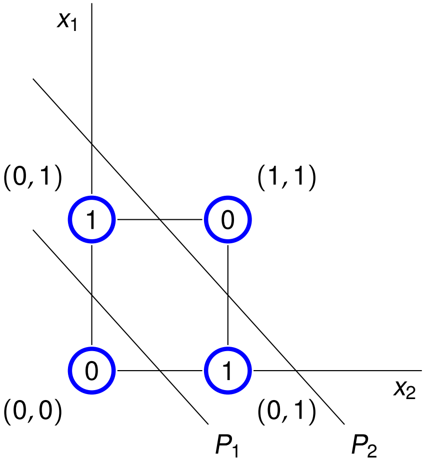
- Activation function is a unit step function

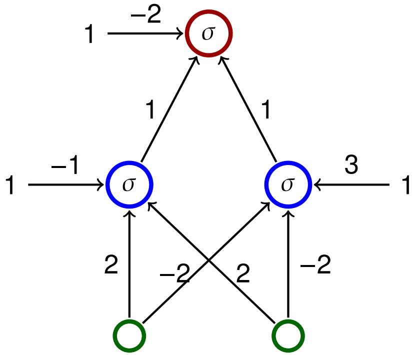
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

- 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

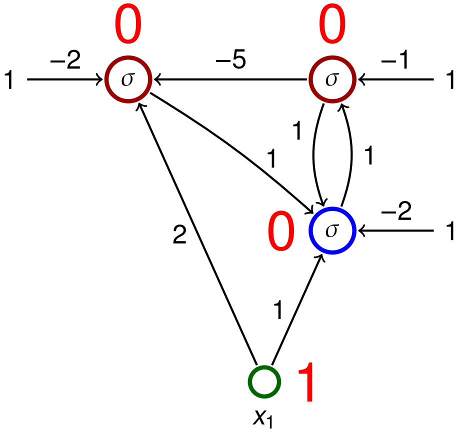
Activity – MLP and linear separation





- 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

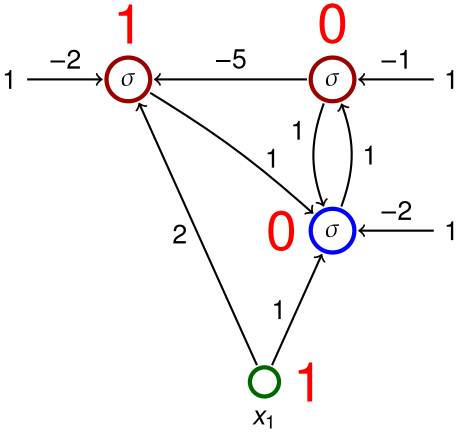


The activation function is the unit step function

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

The input is equal to 1

Activity – example

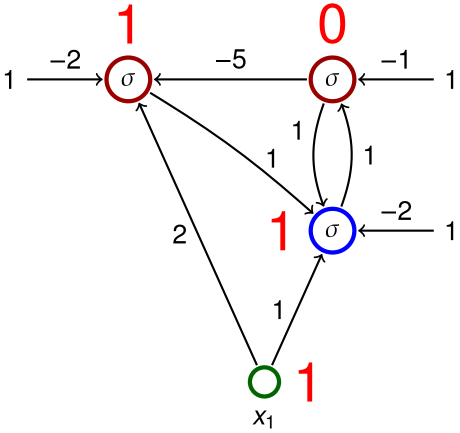


The activation function is the unit step function

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

The input is equal to 1

Activity – example

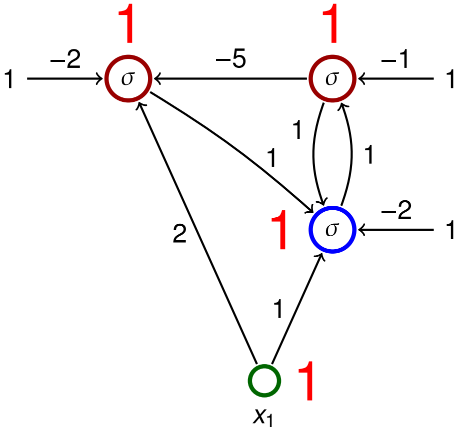


The activation function is the unit step function

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

The input is equal to 1

Activity – example

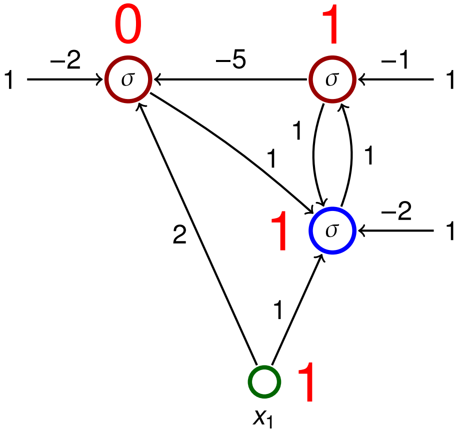


The activation function is the unit step function

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

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)

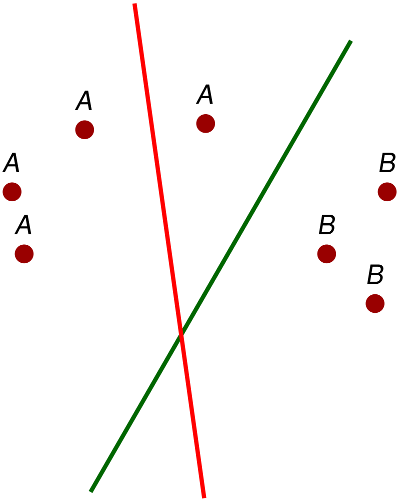
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.

Learning algorithms

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

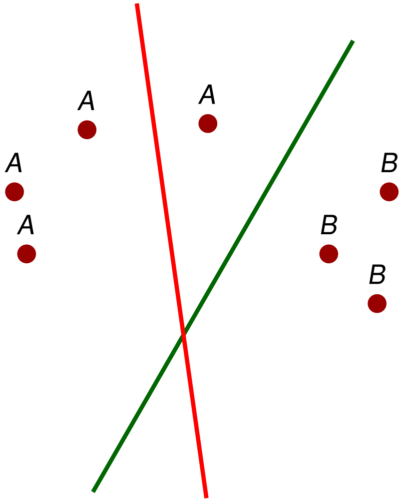
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.
- Unsupervised learning
  - The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

Supervised learning – illustration



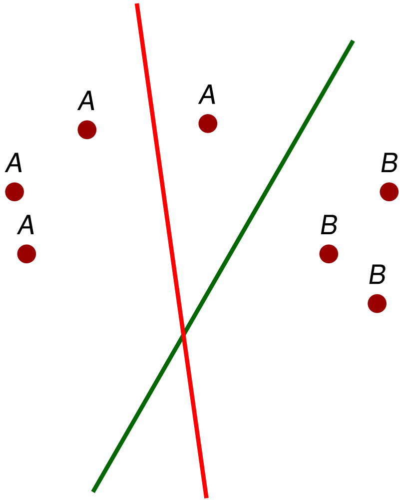
- classification in the plane using a single neuron

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

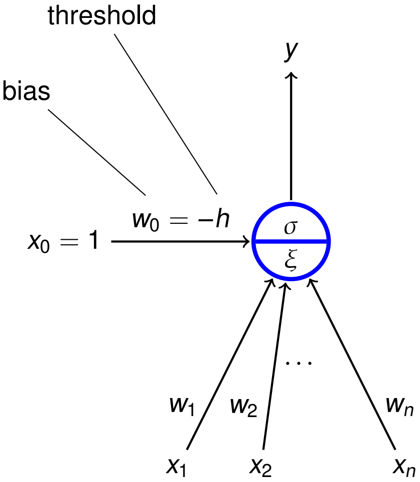
- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks
- generalization and robustness
  - information is encoded in a distributed manned in weights
  - "close" inputs typicaly get similar values

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)



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

  e.g. a *unit step function*

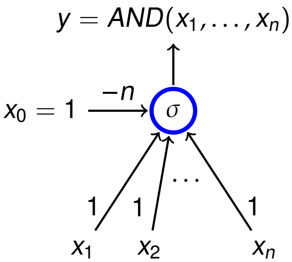
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

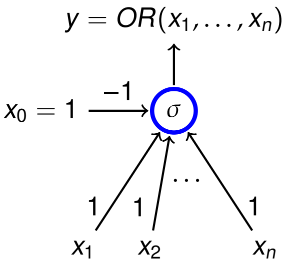
Boolean functions

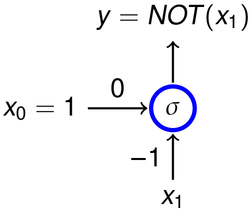
Activation function: *unit step function* $\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$

Boolean functions

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

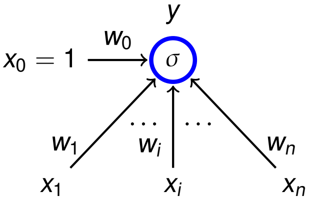
Boolean functions

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 $\vec{v}$:

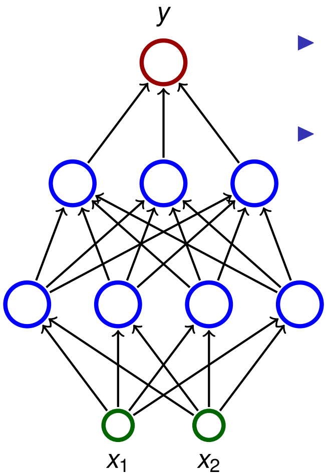


$$w_0 = -\sum_{i=1}^{n} v_i$$

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

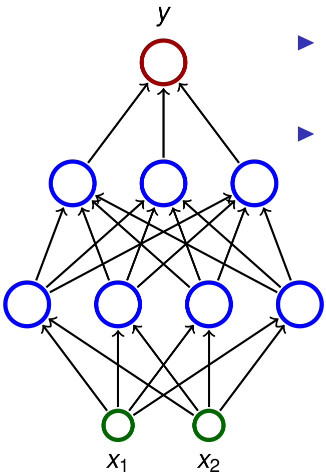
- Now let us connect all outputs of all neurons $N_{\vec{v}}$ satisfying $F(\vec{v}) = 1$ using a neuron implementing OR. $\square$

Non-linear separation



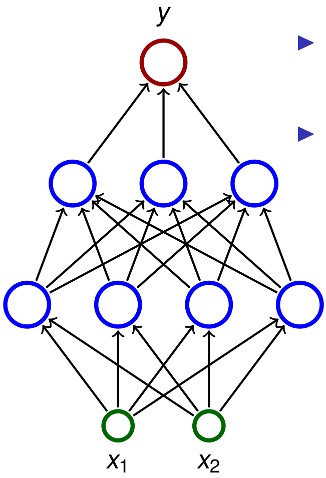
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).

Non-linear separation



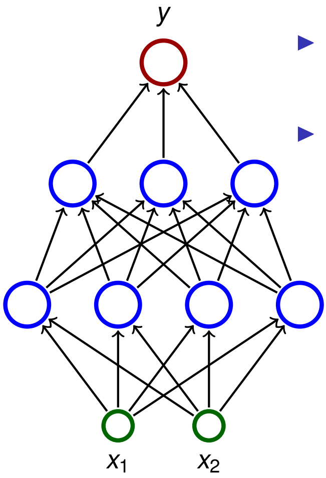
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.

Non-linear separation



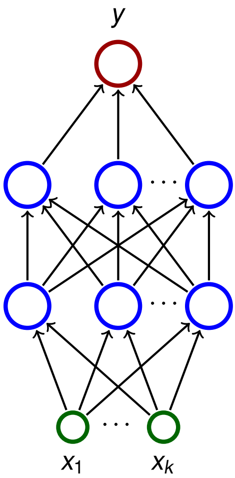
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation



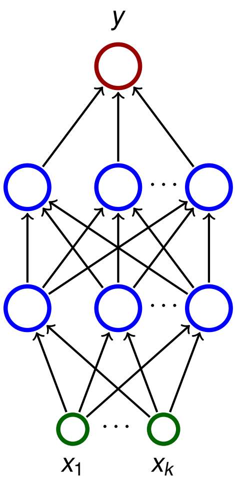
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation – illustration



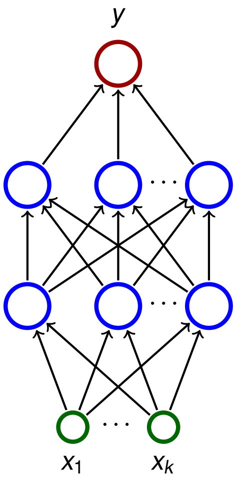
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.

Non-linear separation – illustration



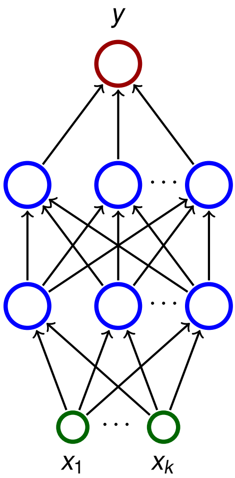
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)

Non-linear separation – illustration



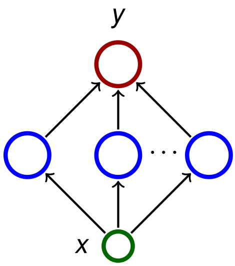
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)
  - Each hypercube K can be separated using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

Non-linear separation – illustration



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

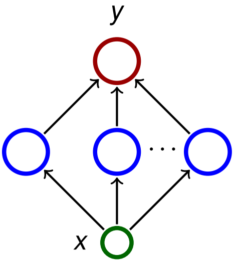
Power of ReLU



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

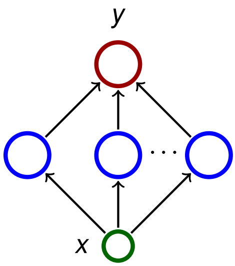


Consider a two layer network

- 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

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

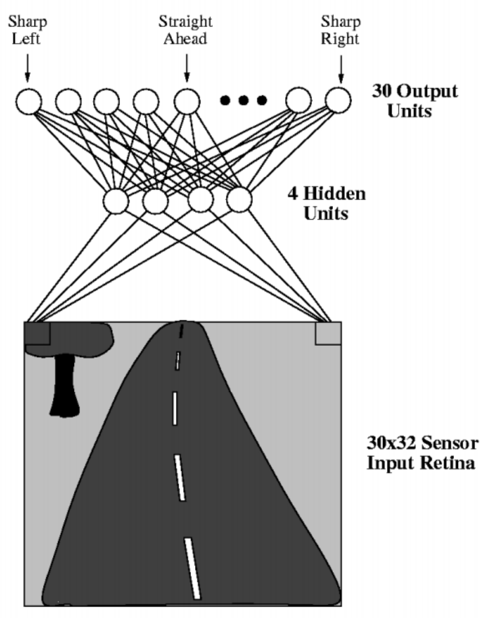
For every "reasonable" set $A \subseteq [0,1]^n$, there is a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following:*

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:

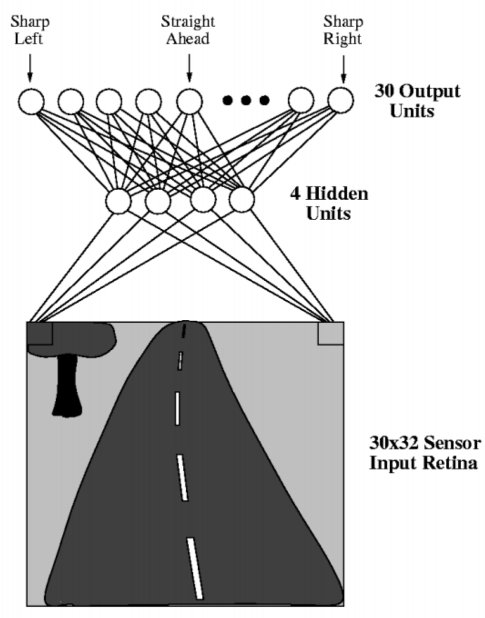
- "reasonable" means Lebesgue measurable
- "most" means that the set of incorrectly classified vectors has the Lebesgue measure smaller than a given $\varepsilon > 0$

Non-linear separation - practical illustration



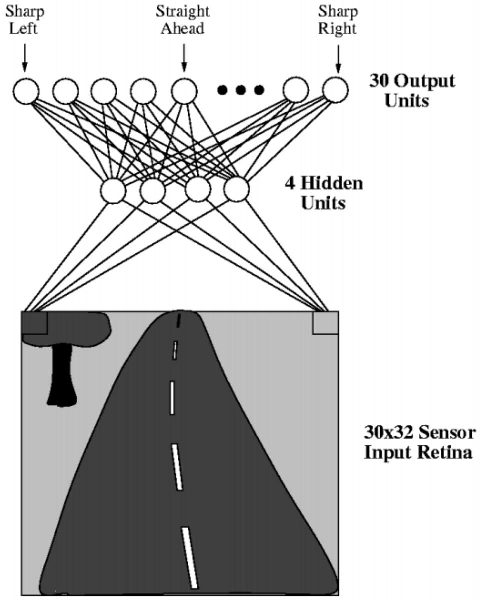
- ALVINN drives a car

Non-linear separation - practical illustration



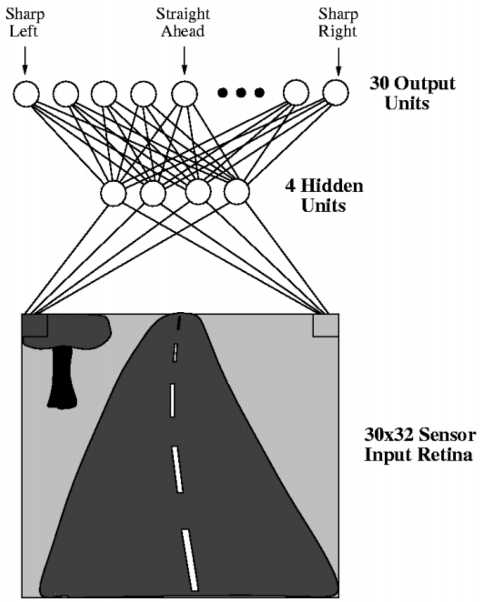
- ALVINN drives a car
- The net has $30 \times 32 = 960$ inputs (the input space is thus $\mathbb{R}^{960}$)

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

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

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

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

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 $F : A \rightarrow \mathbb{R}$ 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.

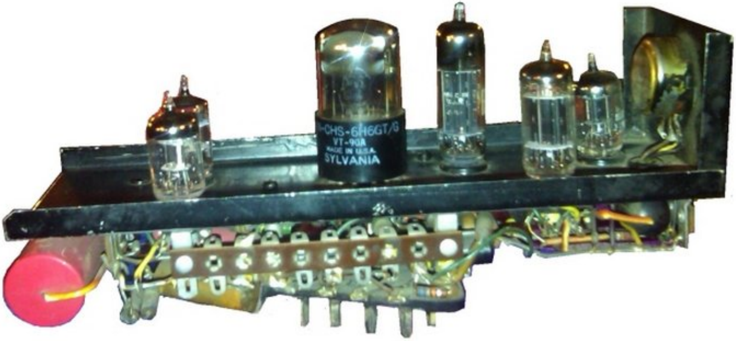
Neural networks vs classical computers

	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.)



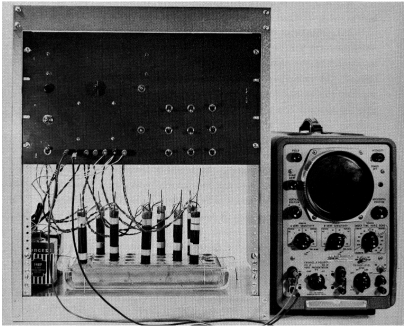
History of neurocomputers

- 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 ...

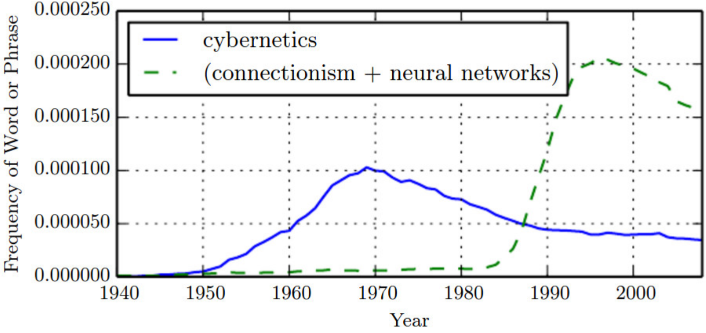
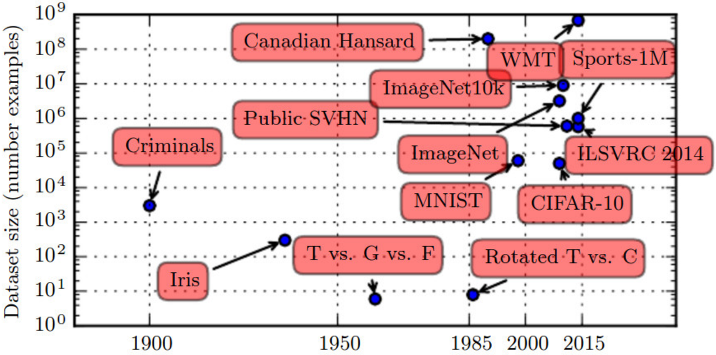


Figure: The figure shows two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks" according to Google Books (the third wave is too recent to appear).

Current hardware – What do we face?

Increasing dataset size ...



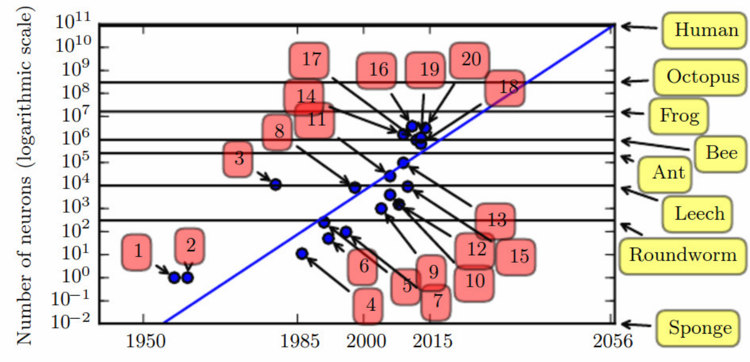
... weakly-supervised pre-training using hashtags from the Instagram uses $3.6 * 10^9$ images.

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

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

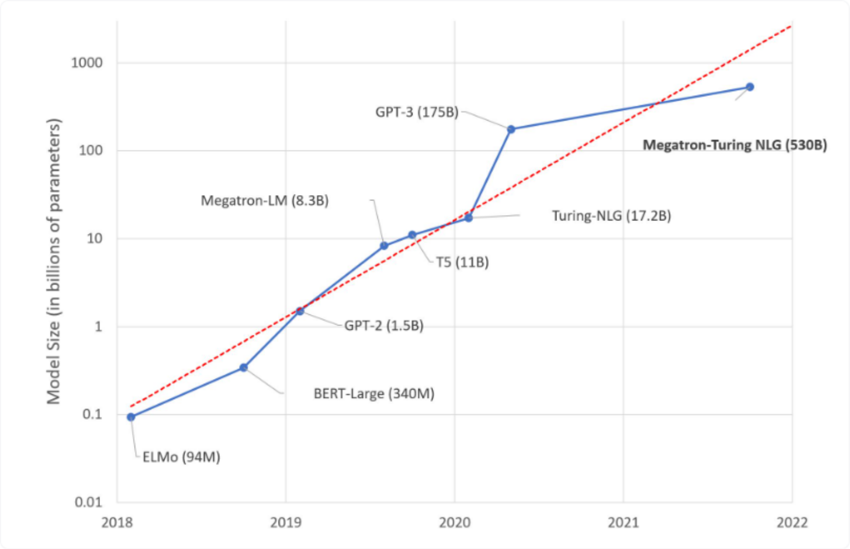
Current hardware – What do we face?

... 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 ...

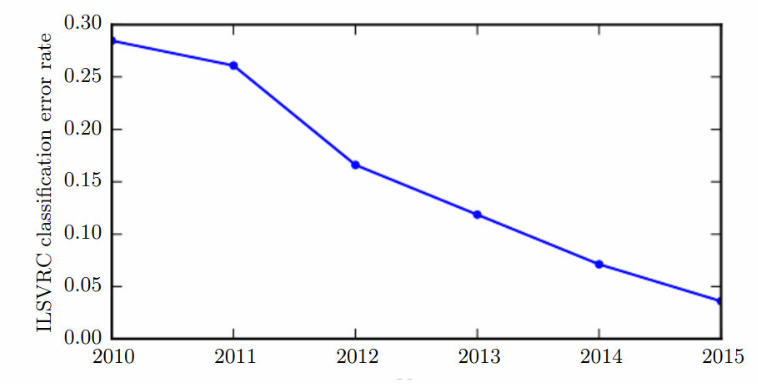


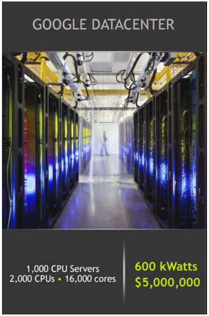
Figure: Since deep networks reached the scale necessary to compete in the ImageNetLarge Scale Visual Recognition Challenge, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky et al. (2014b) and He et al. (2015).

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

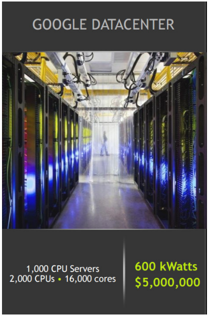
In 2012, Google trained a large network of 1.7 billion weights and 9 layers

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

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

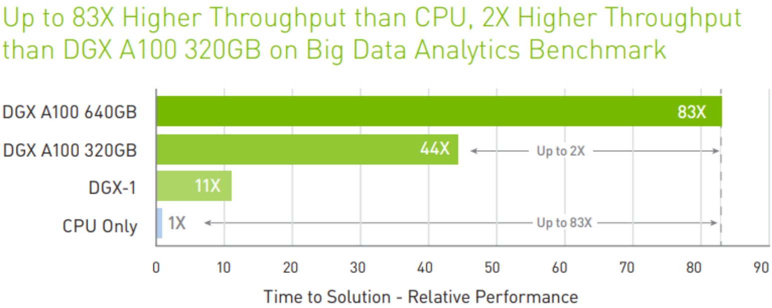
In 2014, similar task performed on Commodity Off-The-Shelf High Performance Computing (COTS HPC) technology: a cluster of GPU servers with Infiniband interconnects and MPI.

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

```python
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

```python
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:

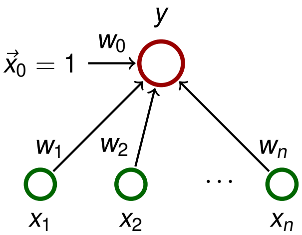
- MATLAB
- R
- STATISTICA
- Weka
- ...

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

Training linear models

Linear regression (ADALINE)

Architecture:



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

Activity:

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

Learning:

- Given a **training dataset**

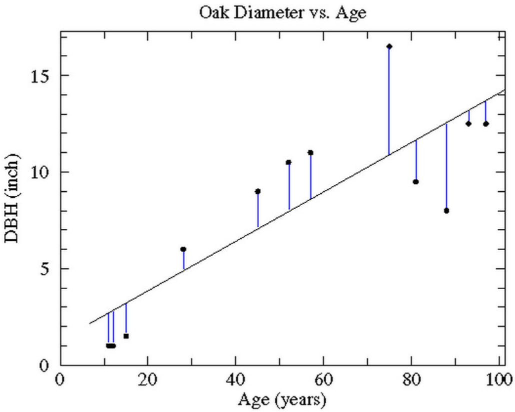
$$\mathcal{T} = \left\{ \left(\vec{x}_1, d_1\right), \left(\vec{x}_2, d_2\right), \ldots, \left(\vec{x}_p, d_p\right) \right\}$$

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.

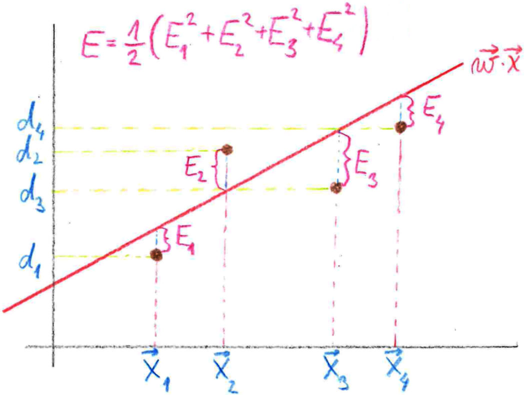
Oaks in Wisconsin

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



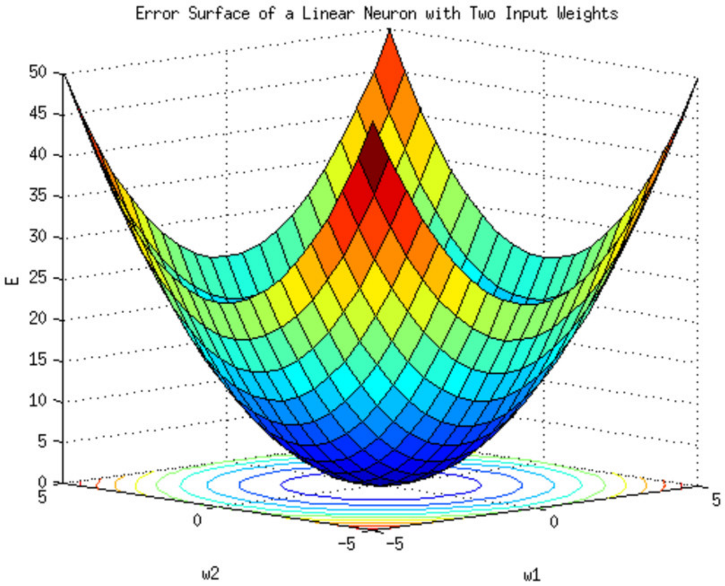
- **Error function:**

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



- 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:

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

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.

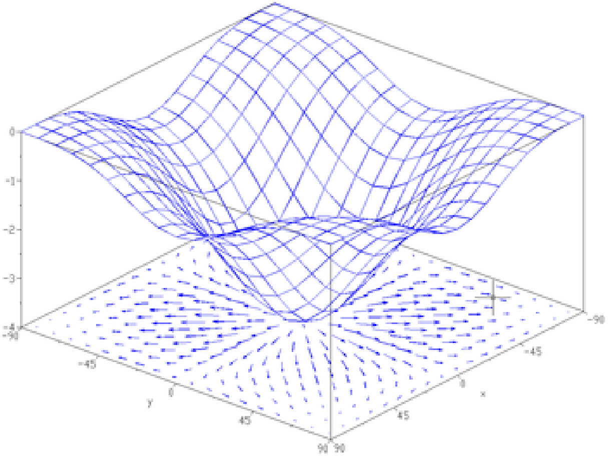
Note that the vectors $\vec{x}_k$ are just parameters of the function E, and are thus fixed!

Fact

If $\nabla E(\vec{w}) = \vec{0} = (0, \dots, 0)$, *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

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

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:

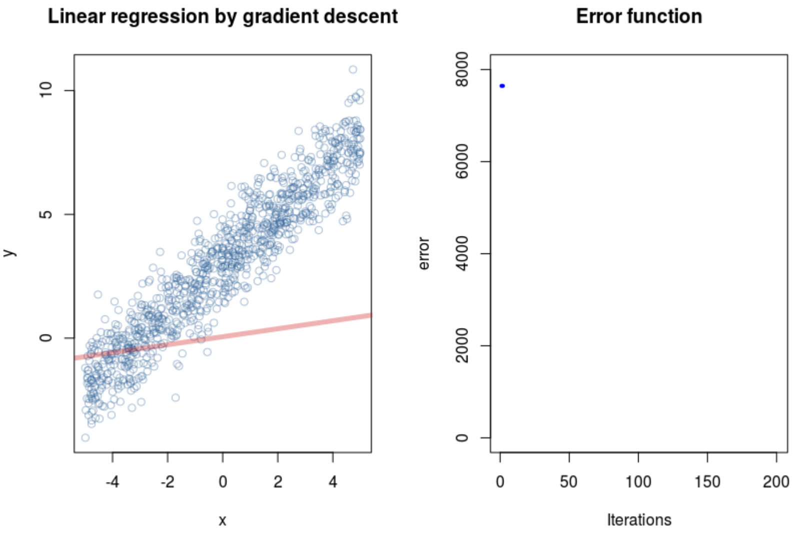
$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \vec{x}_k - d_k\right) \cdot \vec{x}_k\end{aligned}$$

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

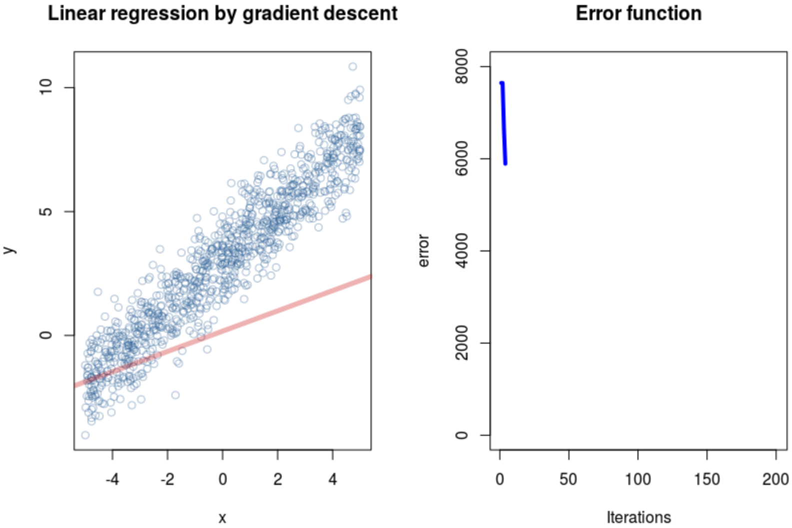
Proposition

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

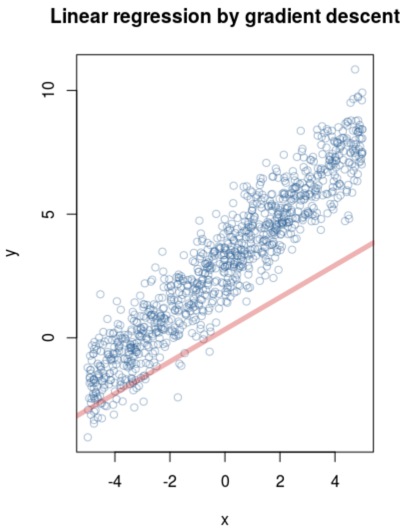
Linear regression - animation

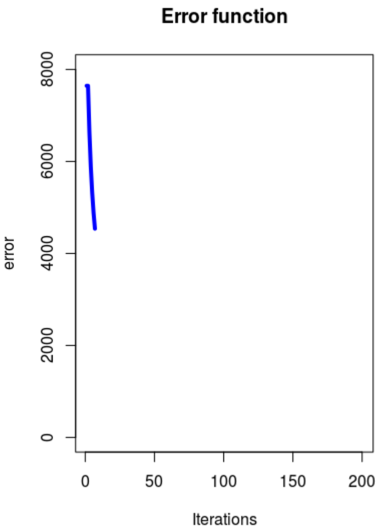


Linear regression - animation

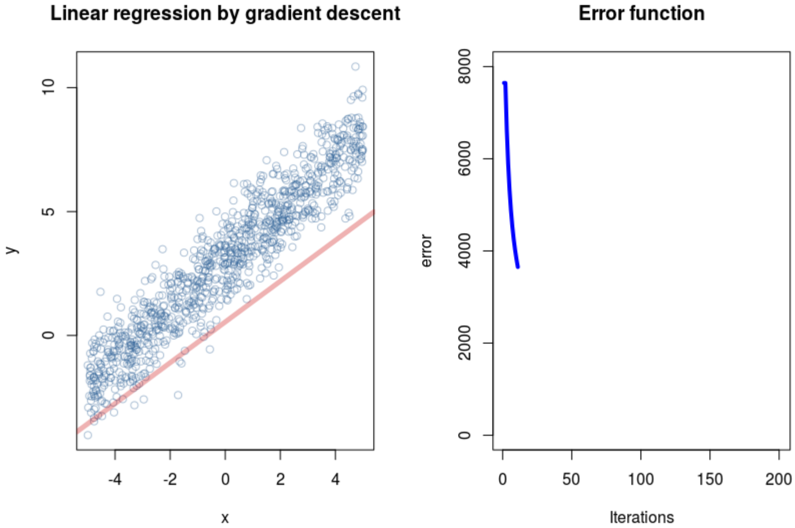


Linear regression - animation

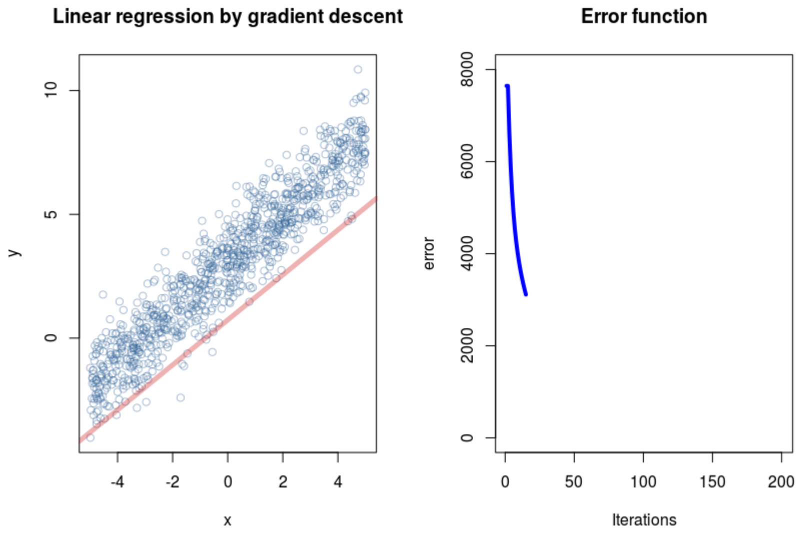




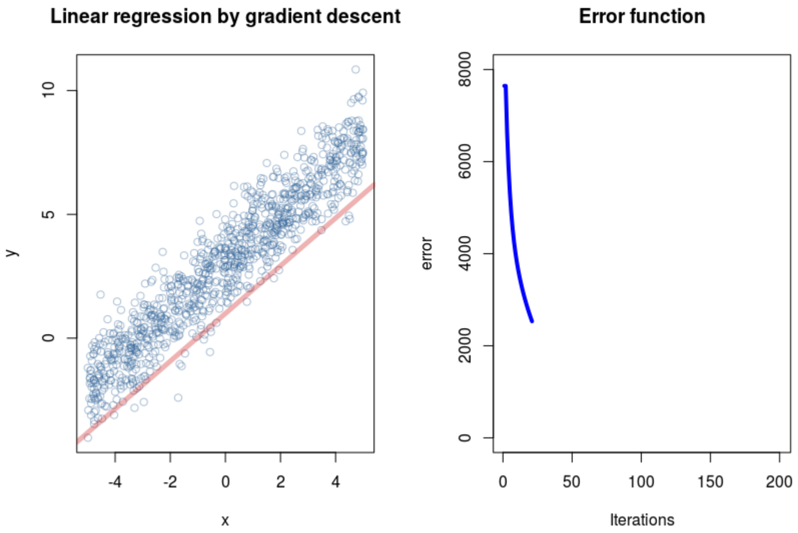
Linear regression - animation



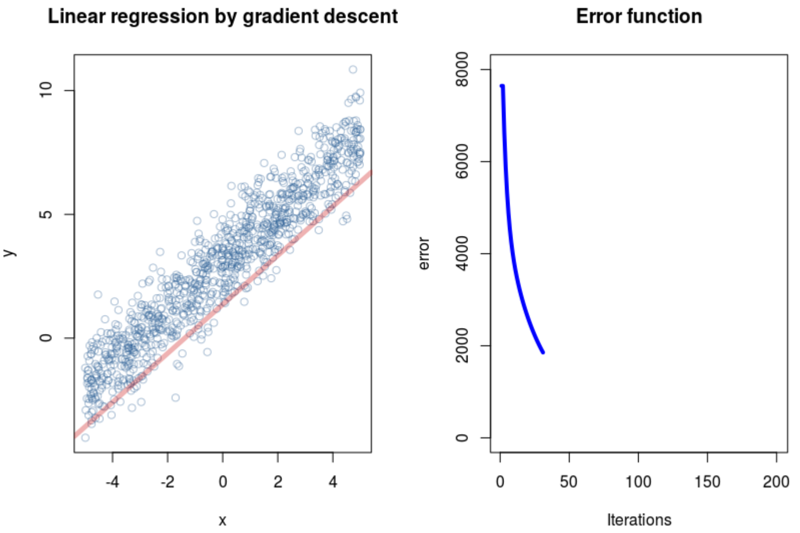
Linear regression - animation



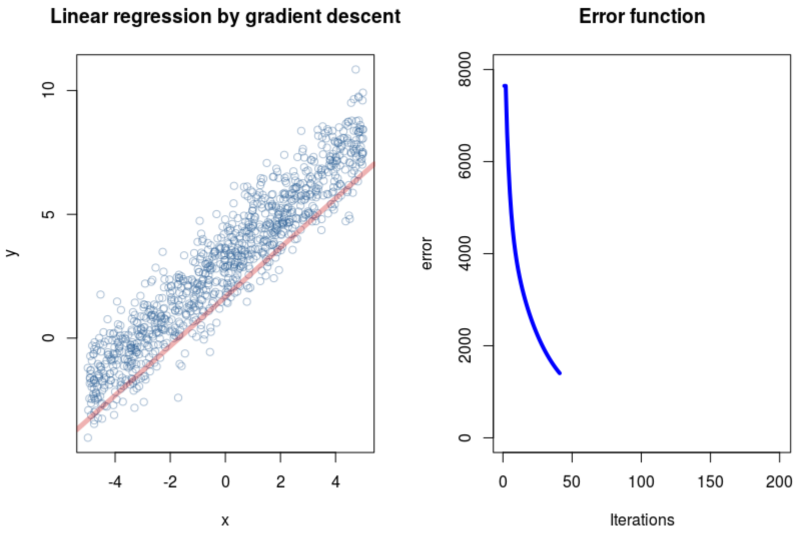
Linear regression - animation



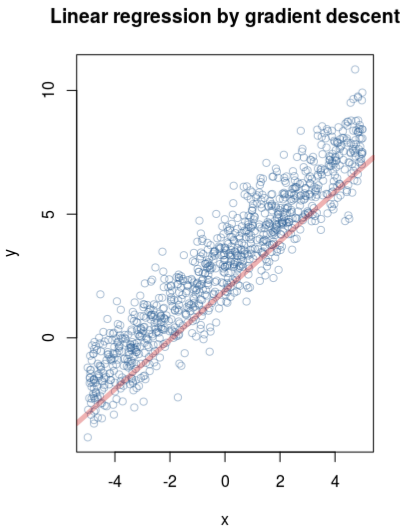
Linear regression - animation

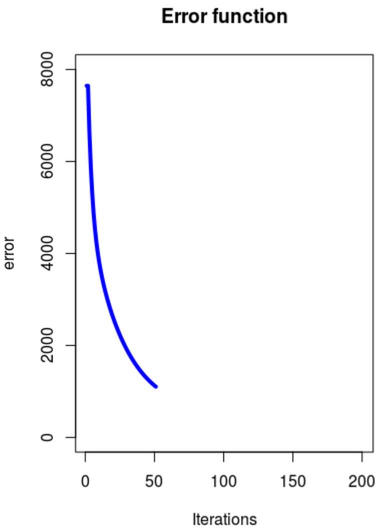


Linear regression - animation

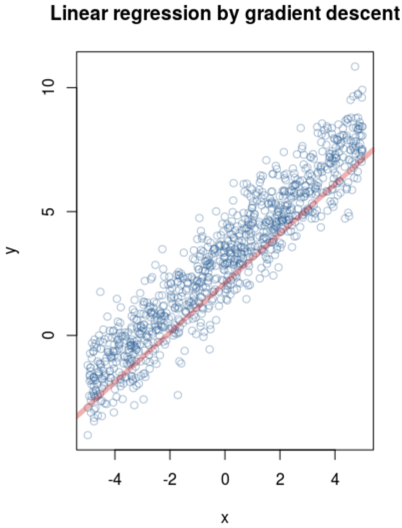


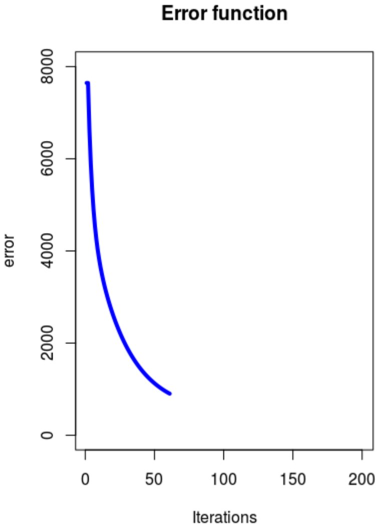
Linear regression - animation



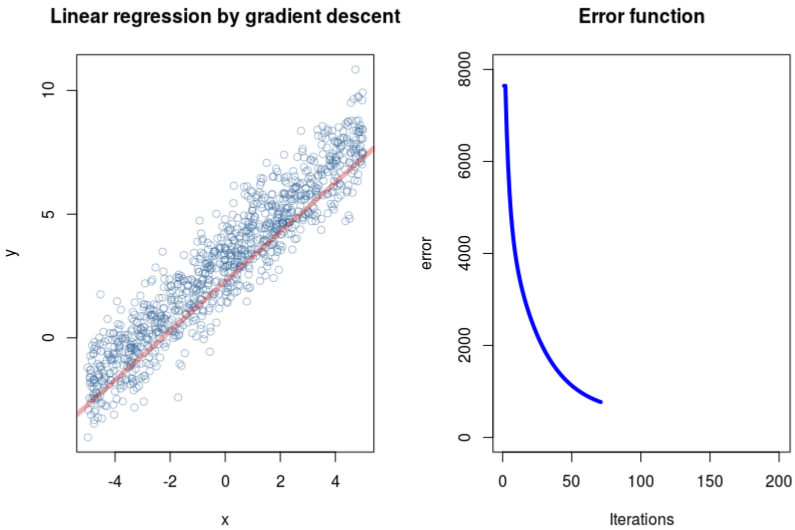


Linear regression - animation

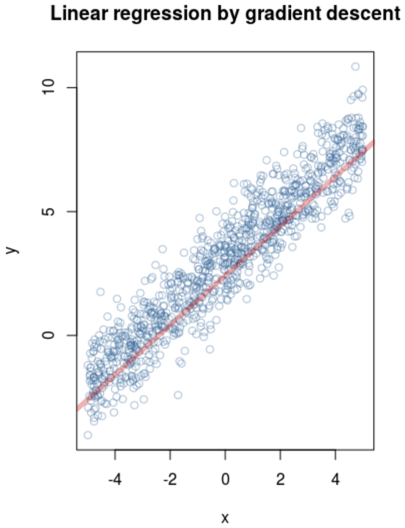


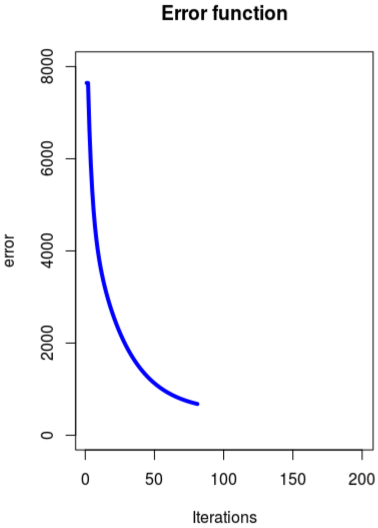


Linear regression - animation

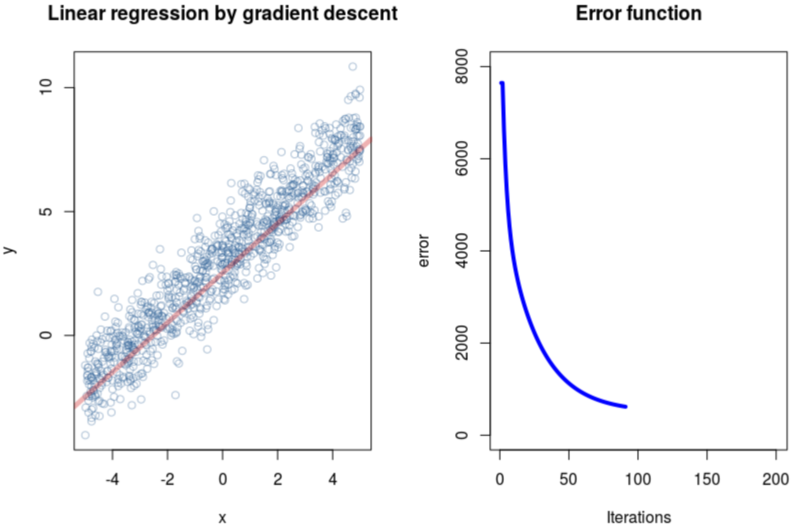


Linear regression - animation

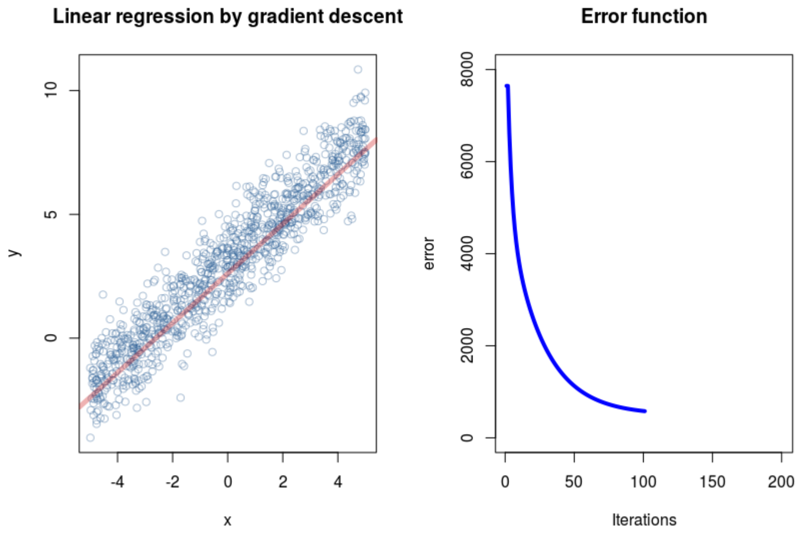




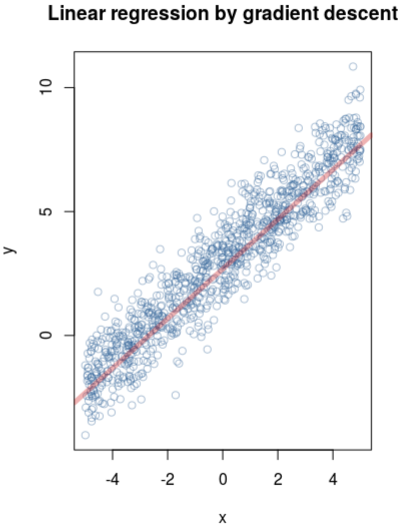
Linear regression - animation

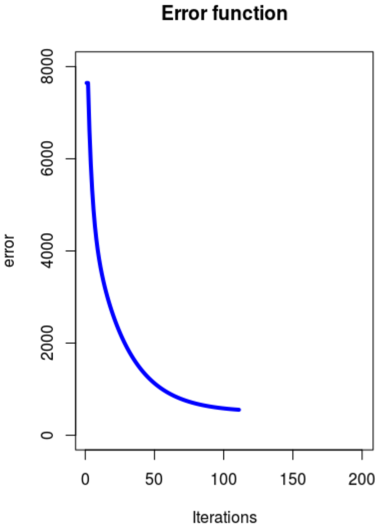


Linear regression - animation

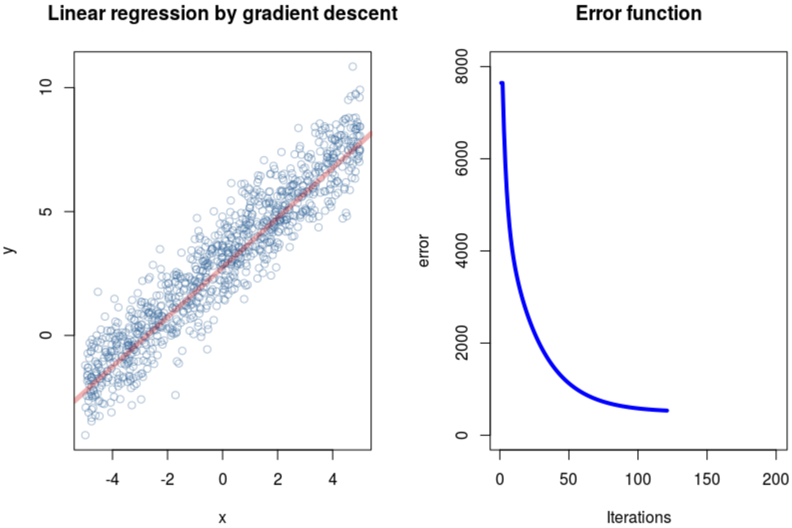


Linear regression - animation

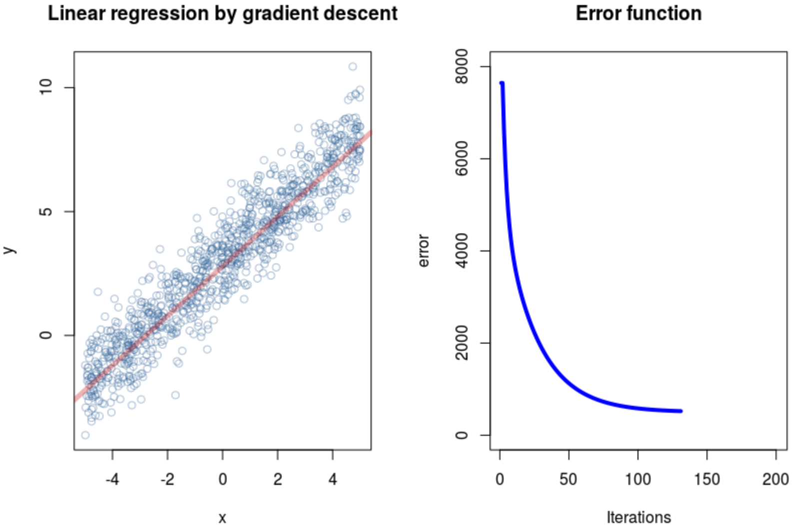




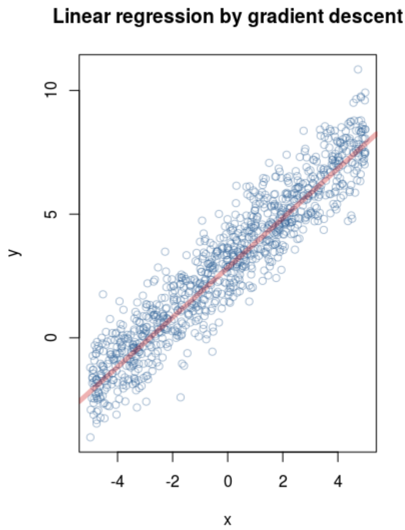
Linear regression - animation

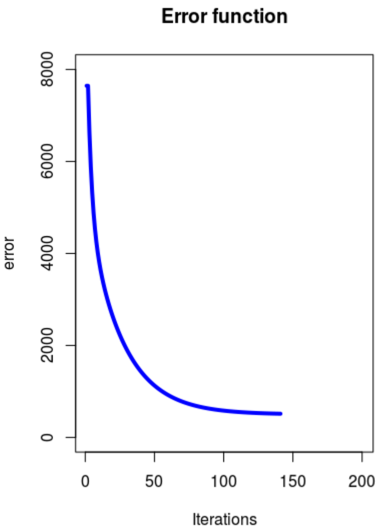


Linear regression - animation

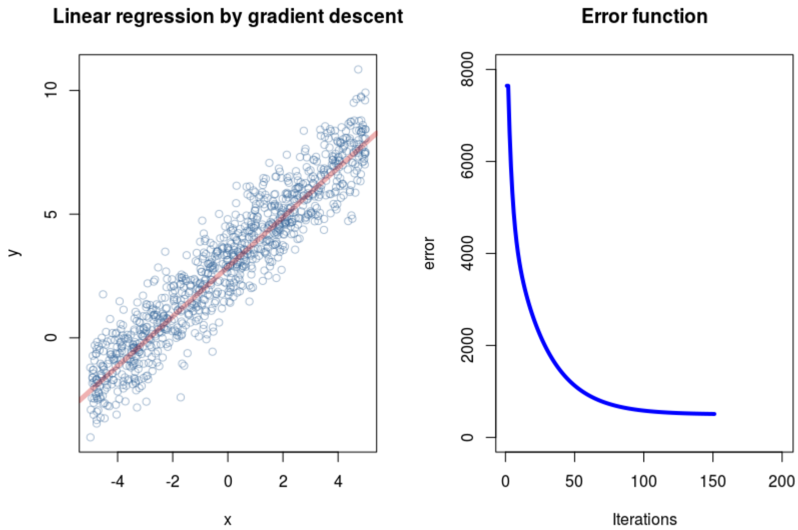


Linear regression - animation

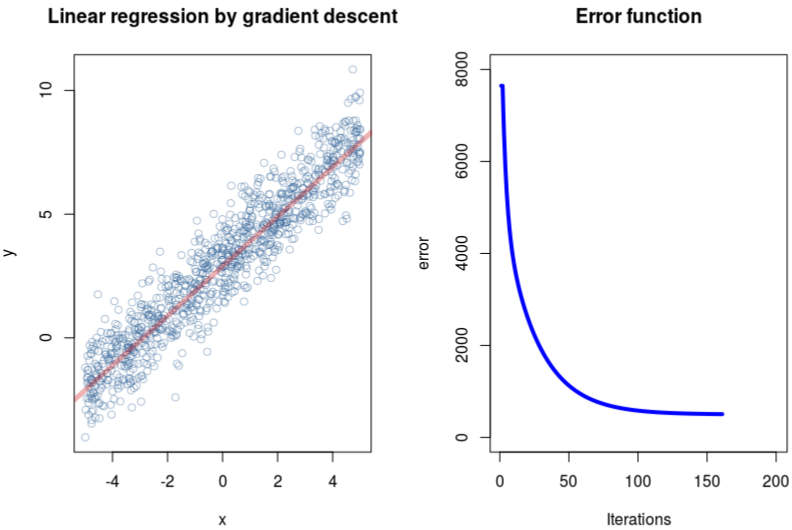




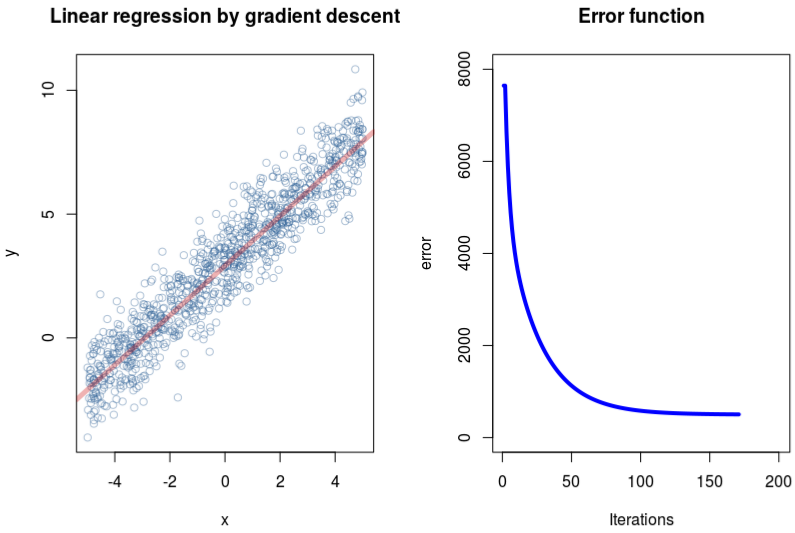
Linear regression - animation



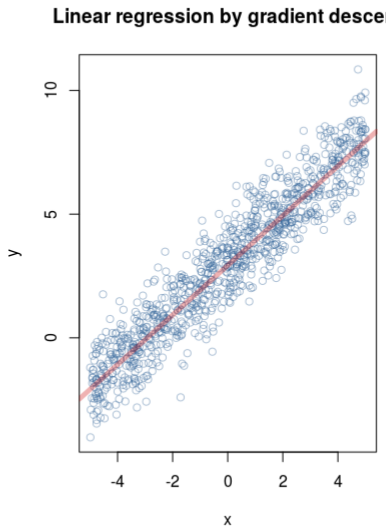
Linear regression - animation

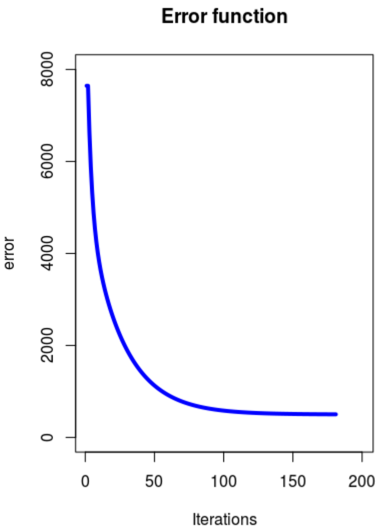


Linear regression - animation

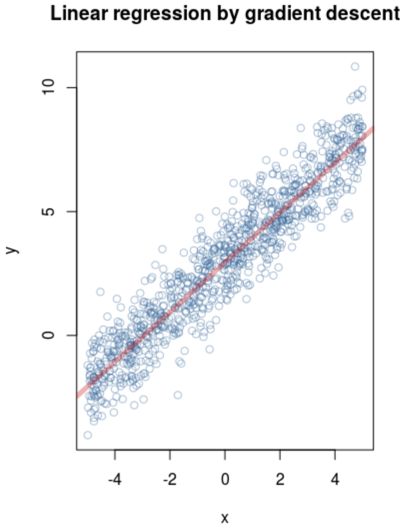


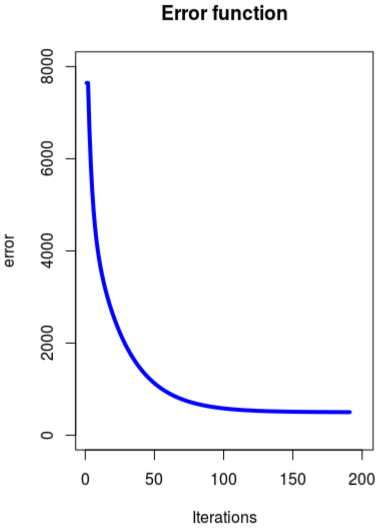
Linear regression - animation



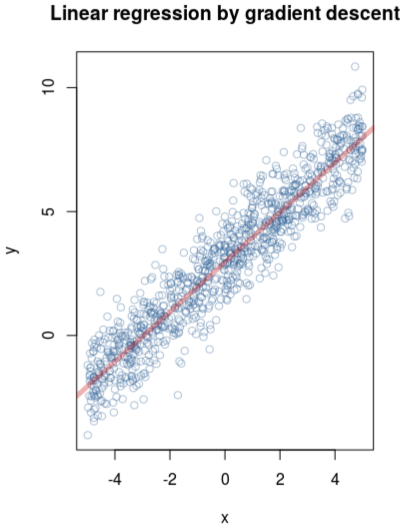


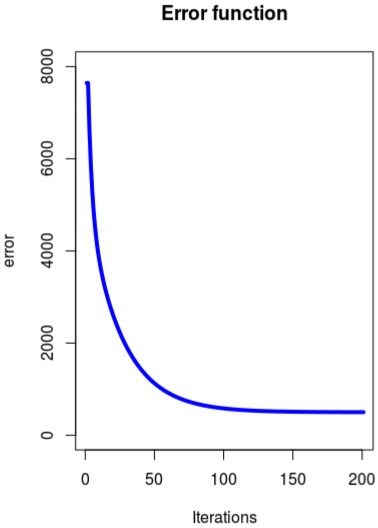
Linear regression - animation





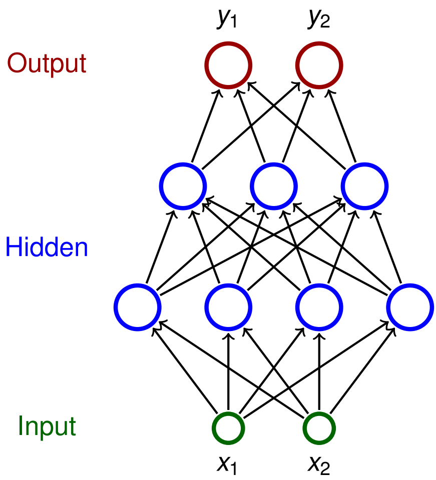
Linear regression - animation





MLP training – theory

Architecture – Multilayer Perceptron (MLP)



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

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)

MLP – architecture

Notation:

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

MLP – architecture

Notation:

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

(define $y_0 = 1$ is the value of the formal unit input)

MLP – architecture

Notation:

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

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

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)

MLP – architecture

Notation:

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

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

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

MLP – architecture

Notation:

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

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

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

MLP – activity

Activity:

- inner potential of neuron j:

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

MLP – activity

Activity:

- inner potential of neuron j:

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

- activation function σ_j for neuron j (arbitrary differentiable)

MLP – activity

Activity:

- inner potential of neuron j:

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

- activation function σ_j for neuron j (arbitrary differentiable)
- State of non-input neuron $j \in Z \setminus X$ after the computation stops:

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

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

MLP – activity

Activity:

- inner potential of neuron j:

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

- activation function σ_j for neuron j (arbitrary differentiable)
- State of non-input neuron $j \in Z \setminus X$ after the computation stops:

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

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

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

MLP – learning

Learning:

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

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

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

MLP – learning

Learning:

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

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

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

- **Error function:**

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

  where

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

MLP – learning algorithm

Batch algorithm (gradient descent):

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

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

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

MLP – learning algorithm

Batch algorithm (gradient descent):

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

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

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

where

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

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

MLP – learning algorithm

Batch algorithm (gradient descent):

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

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

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

where

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

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

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

MLP – error function gradient

For every w_{ji} we have

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

MLP – error function gradient

For every w_{ji} we have

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

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

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

MLP – error function gradient

For every w_{ji} we have

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

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

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

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

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

MLP – error function gradient

For every w_{ji} we have

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

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

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

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

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

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

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

MLP – error function gradient (history)

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

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

MLP – error function gradient (history)

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

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

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

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

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

MLP – computing the gradient

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

MLP – computing the gradient

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

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

MLP – computing the gradient

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

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

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

MLP – computing the gradient

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

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

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

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$

MLP – computing the gradient

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

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

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

1. **forward pass:** compute $y_j = y_j(\vec{w}, \vec{x}_k)$ for all $j \in Z$
2. **backward pass:** compute $\frac{\partial E_k}{\partial y_j}$ for all $j \in Z$ using *backpropagation* (see the next slide!)

MLP – computing the gradient

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

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

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

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

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

MLP – computing the gradient

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

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

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

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

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

4. $\mathcal{E}_{ji} := \mathcal{E}_{ji} + \frac{\partial E_k}{\partial w_{ji}}$

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

MLP – backpropagation

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

MLP – backpropagation

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

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

MLP – backpropagation

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

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

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

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

Complexity of the batch algorithm

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

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

1. forward pass, i.e. computes $y_j(\vec{w}, \vec{x}_k)$
2. backpropagation, i.e. computes $\frac{\partial E_k}{\partial y_j}$

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

The steps 1. - 3. take linear time.

Complexity of the batch algorithm

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

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

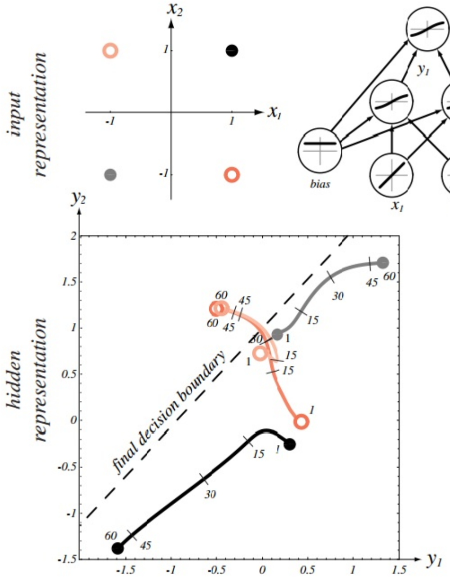
Proof sketch: The algorithm does the following p times:

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

The steps 1. - 3. take linear time.

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

Illustration of the gradient descent – XOR



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

MLP – learning algorithm

Online algorithm:

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

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

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

where

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

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

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

SGD

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

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

    where

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

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

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

Output activations and error functions

Regression:

- The output activation is typically the identity $y_i = \sigma(\xi_i) = \xi_i$.

Output activations and error functions

Regression:

- The output activation is typically the identity $y_i = \sigma(\xi_i) = \xi_i$.
- A **training dataset**

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

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

Output activations and error functions

Regression:

- The output activation is typically the identity $y_i = \sigma(\xi_i) = \xi_i$.
- A **training dataset**

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

  Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $i \in Y$, denote by d_{ki} the desired output of the neuron i for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $(d_{ki})_{i \in Y}$).
- The error function *mean squared error (mse):*

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

  where

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

Output activations and error functions

Classification

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_i) = \frac{e^{\xi_i}}{\sum_{j \in Y} e^{\xi_j}}$$

Output activations and error functions

Classification

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_i) = \frac{e^{\xi_i}}{\sum_{j \in Y} e^{\xi_j}}$$

- A **training dataset**

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

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \{0, 1\}^{|Y|}$ is the desired network output. For every $i \in Y$, denote by d_{ki} the desired output of the neuron i for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $(d_{ki})_{i \in Y}$).

Output activations and error functions

Classification

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_i) = \frac{e^{\xi_i}}{\sum_{j \in Y} e^{\xi_j}}$$

- A **training dataset**

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

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \{0, 1\}^{|Y|}$ is the desired network output. For every $i \in Y$, denote by d_{ki} the desired output of the neuron i for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $(d_{ki})_{i \in Y}$).

- The error function *(categorical) cross entropy:*

$$E(\vec{w}) = -\frac{1}{p} \sum_{k=1}^{p} \sum_{i \in Y} d_{ki} \log(y_i(\vec{w}, \vec{x}_k))$$

Gradient with Softmax & Cross-Entropy

Assume that V is the layer just below the output layer Y.

$$
\begin{aligned}
E(\vec{w}) &= -\frac{1}{p}\sum_{k=1}^{p}\sum_{i\in Y} d_{ki}\log(y_i(\vec{w},\vec{x}_k)) \\
&= -\frac{1}{p}\sum_{k=1}^{p}\sum_{i\in Y} d_{ki}\log\left(\frac{e^{\xi_i}}{\sum_{j\in Y} e^{\xi_j}}\right) \\
&= -\frac{1}{p}\sum_{k=1}^{p}\sum_{i\in Y} d_{ki}\left(\xi_i - \log\left(\sum_{j\in Y} e^{\xi_j}\right)\right) \\
&= -\frac{1}{p}\sum_{k=1}^{p}\sum_{i\in Y} d_{ki}\left(\sum_{\ell\in V} w_{i\ell}y_\ell - \log\left(\sum_{j\in Y} e^{\sum_{\ell\in V} w_{j\ell}y_\ell}\right)\right)
\end{aligned}
$$

Now compute the derivatives $\frac{\delta E}{\delta y_\ell}$ for $\ell \in V$.

Output activations and error functions

Binary classification

Assume a single output neuron $o \in Y = \{o\}$.

- The output activation function *logistic sigmoid*:

$$\sigma_o(\xi_o) = \frac{e^{\xi_o}}{e^{\xi_o} + 1} = \frac{1}{1 + e^{-\xi_o}}$$

Output activations and error functions

Binary classification

Assume a single output neuron $o \in Y = \{o\}$.

- The output activation function *logistic sigmoid*:

$$\sigma_o(\xi_o) = \frac{e^{\xi_o}}{e^{\xi_o} + 1} = \frac{1}{1 + e^{-\xi_o}}$$

- 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 desired output.

Output activations and error functions

Binary classification

Assume a single output neuron $o \in Y = \{o\}$.

- The output activation function *logistic sigmoid*:

$$\sigma_o(\xi_o) = \frac{e^{\xi_o}}{e^{\xi_o} + 1} = \frac{1}{1 + e^{-\xi_o}}$$

- 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 desired output.

- The error function *(Binary) cross-entropy:*

$$E(\vec{w}) = \sum_{k=1}^{p} -(d_k \log(y_o(\vec{w}, \vec{x}_k)) + (1 - d_k) \log(1 - y_o(\vec{w}, \vec{x}_k)))$$

But what is the meaning of the sigmoid?

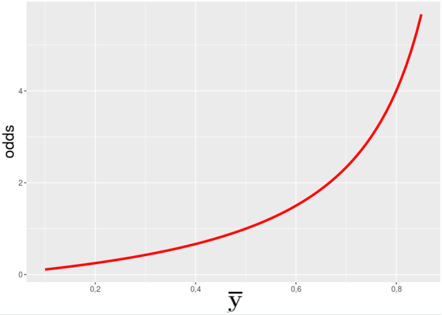
The model gives a probability y_o of the class 1 given an input $\vec{x}$.
But why do we model such a probability using $1/(1 + e^{-\xi_o})$??

But what is the meaning of the sigmoid?

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

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

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



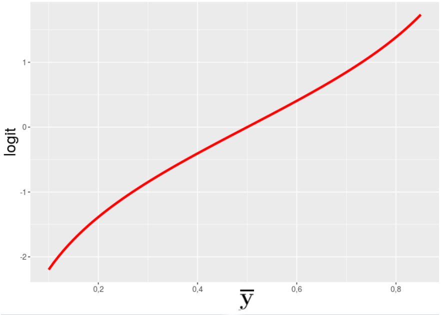
... stretches from 0 to ∞

But what is the meaning of the sigmoid?

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

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

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



... stretches from $-\infty$ to ∞

But what is the meaning of the sigmoid?

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

$$\log(\bar{y}/(1-\bar{y})) = \xi_o$$

(here ξ_o is the inner potential of the output neuron).

But what is the meaning of the sigmoid?

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

$$\log(\bar{y}/(1-\bar{y})) = \xi_o$$

(here ξ_o is the inner potential of the output neuron). Then

$$\log((1-\bar{y})/\bar{y}) = -\xi_o$$

But what is the meaning of the sigmoid?

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

$$\log(\bar{y}/(1-\bar{y})) = \xi_o$$

(here ξ_o is the inner potential of the output neuron). Then

$$\log((1-\bar{y})/\bar{y}) = -\xi_o$$

and

$$(1-\bar{y})/\bar{y} = e^{-\xi_o}$$

But what is the meaning of the sigmoid?

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

$$\log(\bar{y}/(1-\bar{y})) = \xi_o$$

(here ξ_o is the inner potential of the output neuron). Then

$$\log((1-\bar{y})/\bar{y}) = -\xi_o$$

and

$$(1-\bar{y})/\bar{y} = e^{-\xi_o}$$

and

$$\bar{y} = \frac{1}{1+e^{-\xi_o}}$$

That is, modeling the probability using the *classification model* (with the logistic output activation) corresponds to modeling log-odds using the *regression model* (with the identity output activation).

Log likelihood is your friend!

What is the statistical meaning of the cross-entropy?

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

Log likelihood is your friend!

What is the statistical meaning of the cross-entropy?

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\bar{y}$ and is unknown!

Log likelihood is your friend!

What is the statistical meaning of the cross-entropy?

- Let's have a "coin" (sides 0 and 1).
- The probability of 1 is $\bar{y}$ 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

Log likelihood is your friend!

What is the statistical meaning of the cross-entropy?

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

Log likelihood is your friend!

What is the statistical meaning of the cross-entropy?

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

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

Log likelihood is your friend!

Keep in mind our dataset:

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

Log likelihood is your friend!

Keep in mind our dataset:

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

Assume that the data was generated by independent trials, then the probability of getting exactly $\mathcal{T}$ 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 giving a probability $y_o(\vec{w}, \vec{x})$ given input $\vec{x}$.

Let the coin depend on the input

Consider our model giving a probability $y_o(\vec{w}, \vec{x})$ given input $\vec{x}$. Recall that the training dataset is

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

Here $\vec{x}_k = (x_{k0}, x_{k1} \dots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, 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 giving a probability $y_o(\vec{w}, \vec{x})$ given input $\vec{x}$. Recall that the training dataset is

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

Here $\vec{x}_k = (x_{k0}, x_{k1} \dots, x_{kn}) \in \mathbb{R}^{n+1}$, $x_{k0} = 1$, is the k-th input, and $d_k \in \{0, 1\}$ is the expected output.

The *likelihood*:

$$L(\vec{w}) = \prod_{k=1}^{p} \left(y_o(\vec{w}, \vec{x}_k)\right)^{d_k} \cdot \left(1 - y_o(\vec{w}, \vec{x}_k)\right)^{(1-d_k)}$$

$\log(L) =$
$\sum_{k=1}^{p} \left(d_k \cdot \log(y_o(\vec{w}, \vec{x}_k)) + (1 - d_k) \cdot \log(1 - y_o(\vec{w}, \vec{x}_k)) \right)$
and thus $-\log(L)$ = the cross-entropy.

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

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Squared error $E(w) = \frac{1}{2}(y - d)^2$.

$$\frac{\delta E}{\delta w} = (y - d) \cdot y \cdot (1 - y) \cdot x$$

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Squared error $E(w) = \frac{1}{2}(y - d)^2$.

$$\frac{\delta E}{\delta w} = (y - d) \cdot y \cdot (1 - y) \cdot x$$

Thus

- If $d = 1$ and $y \approx 0$, then $\frac{\delta E}{\delta w} \approx 0$
- If $d = 0$ and $y \approx 1$, then $\frac{\delta E}{\delta w} \approx 0$

The gradient of E is small even though *the model is wrong*!

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Cross-entropy error $E(w) = d \cdot \log(y) + (1 - d) \cdot \log(1 - y)$.

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Cross-entropy error $E(w) = d \cdot \log(y) + (1 - d) \cdot \log(1 - y)$.

For $d = 1$

$$\frac{\delta E}{\delta w} = \frac{1}{y} \cdot y \cdot (1 - y) \cdot x = (1 - y) \cdot x$$

which is close to x for $y \approx 0$.

Squared Error vs Logistic Output Activation

Consider a single neuron model $y = \sigma(w \cdot x) = 1/(1 + e^{-w \cdot x})$ where $w \in \mathbb{R}$ is the weight (ignore the bias).

A training dataset $\mathcal{T} = \{(x, d)\}$ where $x \in \mathbb{R}$ and $d \in \{0, 1\}$.

Cross-entropy error $E(w) = d \cdot \log(y) + (1 - d) \cdot \log(1 - y)$.

For $d = 1$

$$\frac{\delta E}{\delta w} = \frac{1}{y} \cdot y \cdot (1 - y) \cdot x = (1 - y) \cdot x$$

which is close to x for $y \approx 0$.

For $d = 0$

$$\frac{\delta E}{\delta w} = \frac{1}{1 - y} \cdot (-y) \cdot (1 - y) \cdot x = -y \cdot x$$

which is close to $-x$ for $y \approx 1$.

MLP training – practical issues

Practical issues of gradient descent

- Training efficiency:
  - What size of a minibatch?
  - How to choose the learning rate $\varepsilon(t)$ and control SGD ?
  - How to pre-process the inputs?
  - How to initialize weights?
  - How to choose desired output values of the network?

Practical issues of gradient descent

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

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

Issues in gradient descent

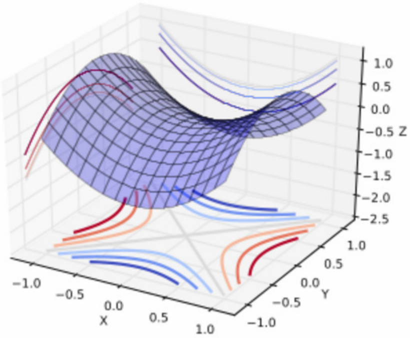
- Small networks: Lots of local minima where the descent gets stuck.
- The model identifiability problem: Swapping incoming weights of neurons i and j leaves the same network topology – **weight space symmetry.**
- Recent studies show that for sufficiently large networks all local minima have low values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

Issues in gradient descent

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

Saddle points

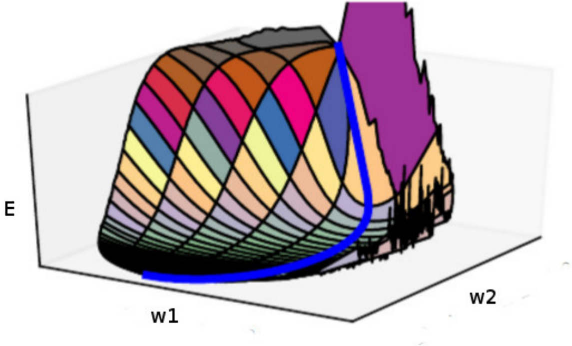
One can show (by a combinatorial argument) that larger networks have exponentially more saddle points than local minima.



Issues in gradient descent – too slow descent

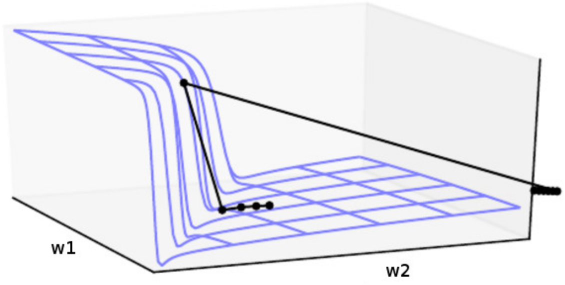
- flat regions

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

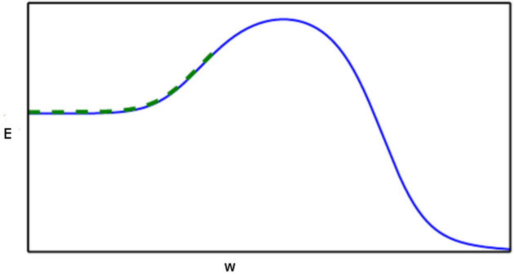


Issues in gradient descent – too fast descent

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



Issues in gradient descent – local vs global structure



What if we initialize on the left?

Gradient Descent in Large Networks

Theorem

Assume (roughly),

- *activation functions: "smooth" ReLU (softplus)*

  $$\sigma(z) = \log(1 + \exp(z))$$

  In general: Smooth, non-polynomial, analytic, Lipschitz continuous.

- *inputs $\vec{x}_k$ of Euclidean norm equal to 1, desired values d_k satisfying $|d_k| \in O(1)$,*
- *the number of hidden neurons per layer sufficiently large (polynomial in certain numerical characteristics of inputs roughly measuring their similarity, and exponential in the depth of the network),*
- *the learning rate constant and sufficiently small.*

The gradient descent converges (with high probability w.r.t. random initialization) to a global minimum with zero error at linear rate.

Later we get to a special type of networks called ResNet where the above result demands only polynomially many neurons per layer (w.r.t. depth).

Issues in computing the gradient

- vanishing and exploding gradients

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

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

Issues in computing the gradient

- vanishing and exploding gradients

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

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

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

Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.

Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches.

Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches.
- If all examples in the batch are to be processed in parallel (as is the typical case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.

Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches.
- If all examples in the batch are to be processed in parallel (as is the typical case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- It is common (especially when using GPUs) for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.

Minibatch size

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

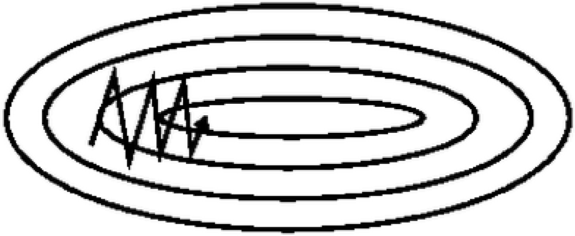
  It has been observed in practice that when using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize.

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

Momentum

Issue in the gradient descent:

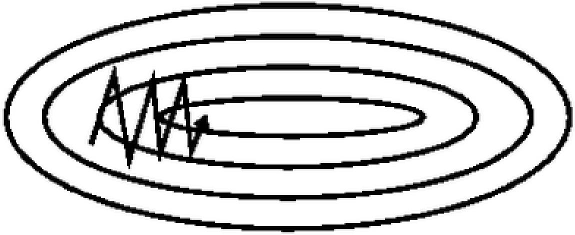
- $\nabla E(\vec{w}^{(t)})$ constantly changes direction (but the error steadily decreases).



Momentum

Issue in the gradient descent:

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

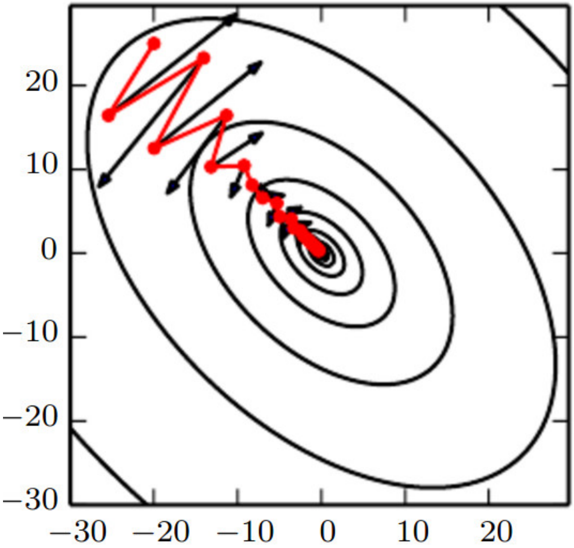


Solution: In every step add the change made in the previous step (weighted by a factor α):

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

where $0 < \alpha < 1$.

Momentum – illustration



SGD with momentum

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

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

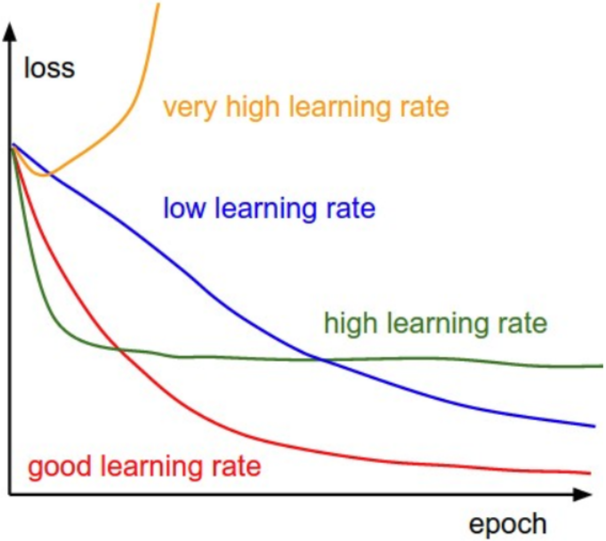
    where

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

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

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

Learning rate



Search for the learning rate

- Use settings from a successful solution of a similar problem as a baseline.
- Search for the learning rate using the learning monitoring:
  - Search through values from small (e.g. 0.001) to (0.1), possibly multiplying by 2.
  - Train for several epochs, observe the learning curves (see cross-validation later).

Adaptive learning rate

- Power scheduling: Set $\epsilon(t) = \epsilon_0/(1 + t/s)$ where ϵ_0 is an initial learning rate and s a number of steps

  (after s steps the learning rate is $\epsilon_0/2$, after $2s$ it is $\epsilon_0/3$ etc.)

Adaptive learning rate

- Power scheduling: Set $\epsilon(t) = \epsilon_0/(1 + t/s)$ where ϵ_0 is an initial learning rate and s a number of steps

  (after s steps the learning rate is $\epsilon_0/2$, after $2s$ it is $\epsilon_0/3$ etc.)

- Exponential scheduling: Set $\epsilon(t) = \epsilon_0 \cdot 0.1^{t/s}$.

  (the learning rate decays faster than in the power scheduling)

Adaptive learning rate

- Power scheduling: Set $\epsilon(t) = \epsilon_0/(1 + t/s)$ where ϵ_0 is an initial learning rate and s a number of steps

  (after s steps the learning rate is $\epsilon_0/2$, after $2s$ it is $\epsilon_0/3$ etc.)

- Exponential scheduling: Set $\epsilon(t) = \epsilon_0 \cdot 0.1^{t/s}$.

  (the learning rate decays faster than in the power scheduling)

- Piecewise constant scheduling: A constant learning rate for a number of steps/epochs, then a smaller learning rate, and so on.

Adaptive learning rate

- Power scheduling: Set $\epsilon(t) = \epsilon_0/(1 + t/s)$ where ϵ_0 is an initial learning rate and s a number of steps

  (after s steps the learning rate is $\epsilon_0/2$, after $2s$ it is $\epsilon_0/3$ etc.)

- Exponential scheduling: Set $\epsilon(t) = \epsilon_0 \cdot 0.1^{t/s}$.

  (the learning rate decays faster than in the power scheduling)

- Piecewise constant scheduling: A constant learning rate for a number of steps/epochs, then a smaller learning rate, and so on.

- 1cycle scheduling: Start by increasing the initial learning rate from ϵ_0 linearly to ϵ_1 (approx. $\epsilon_1 = 10\epsilon_0$) halfway through training. Then decrease from ϵ_1 linearly to ϵ_0. Finish by dropping the learning rate by several orders of magnitude (still linearly).

  According to a 2018 paper by Leslie Smith this may converge much faster (100 epochs vs 800 epochs on CIFAR10 dataset).

For comparison of some methods see: AN EMPIRICAL STUDY OF LEARNING RATES IN DEEP NEURAL NETWORKS FOR SPEECH RECOGNITION, Senior et al

AdaGrad

So far we have considered fixed schedules for learning rates.

It is better to have

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

AdaGrad uses individually adapting learning rate for each weight.

SGD with AdaGrad

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

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

SGD with AdaGrad

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

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

    where

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

    and

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

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

RMSProp

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

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

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

SGD with RMSProp

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

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

SGD with RMSProp

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

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

    where

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

    and

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

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

Other optimization methods

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

A natural question: Which algorithm should one choose?

Other optimization methods

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

A natural question: Which algorithm should one choose?

Unfortunately, there is currently no consensus on this point.

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

Other optimization methods

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

A natural question: Which algorithm should one choose?

Unfortunately, there is currently no consensus on this point.

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

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

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

Choice of (hidden) activations

Generic requirements imposed on activation functions:

1. differentiability

   (to do gradient descent)

2. non-linearity

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

3. monotonicity

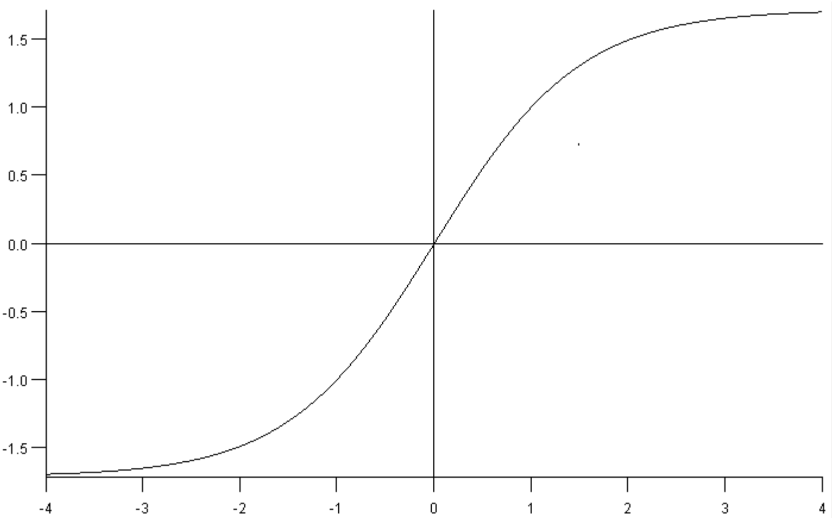
   (local extrema of activation functions induce local extrema of the error function)

4. "linearity"

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

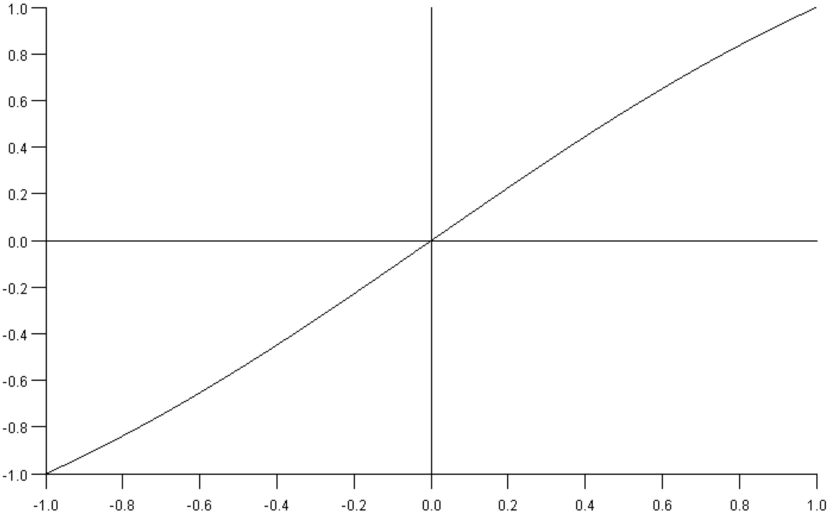
The choice of activation functions is closely related to input preprocessing and the initial choice of weights. I will illustrate the reasoning on sigmoidal functions; say few words about other activation functions later.

Activation functions – tanh



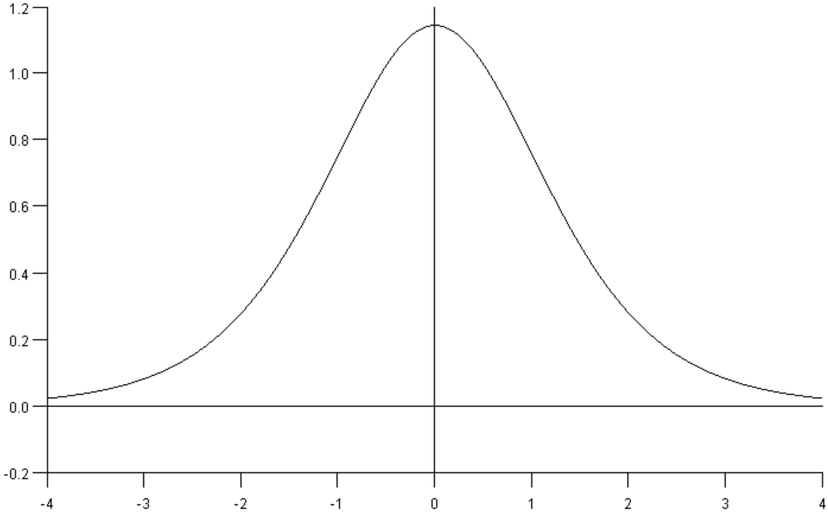
$\sigma(\xi) = 1.7159 \cdot \mathrm{tanh}(\frac{2}{3} \cdot \xi)$, we have $\lim_{\xi \to \infty} \sigma(\xi) = 1.7159$ and $\lim_{\xi \to -\infty} \sigma(\xi) = -1.7159$

Activation functions – tanh



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

Activation functions – tanh



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

Input preprocessing

- Some inputs may be much larger than others.

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

Input preprocessing

- Some inputs may be much larger than others.

  E.g..: Height vs weight of a person, maximum speed of a car (in km/h) vs its price (in CZK), etc.
- Large inputs have greater influence on the training than the small ones. In addition, too large inputs may slow down learning (saturation of activation functions).

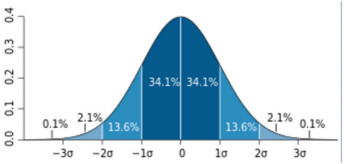
Input preprocessing

- Some inputs may be much larger than others.

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

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

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



(illustration of standard deviation)

Initial weights (for tanh)

- Assume weights chosen in random. What distribution?

Initial weights (for tanh)

- Assume weights chosen in random. What distribution?
- Consider the activation function $\sigma(\xi) = 1.7159 \cdot \mathtt{tanh}(\frac{2}{3} \cdot \xi)$ for all neurons.
  - σ is almost linear on $[-1, 1]$,
  - σ saturates out of the interval $[-4, 4]$ (i.e. it is close to its limit values and its derivative is close to 0.

Initial weights (for tanh)

- Assume weights chosen in random. What distribution?
- Consider the activation function $\sigma(\xi) = 1.7159 \cdot \mathtt{tanh}(\frac{2}{3} \cdot \xi)$ for all neurons.
  - σ is almost linear on $[-1, 1]$,
  - σ saturates out of the interval $[-4, 4]$ (i.e. it is close to its limit values and its derivative is close to 0.

  Thus

  - for too small weights we may get (almost) linear model.
  - for too large weights the activations may get saturated and the learning will be very slow.

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

Normal LeCun initialization

- Assume the input data have the mean $= 0$ and the variance $= 1$.

  Consider a neuron j from the first layer with n inputs. Assume its weights chosen randomly by the normal distribution $\mathcal{N}(0, w^2)$.

  Assume that all random choices are independent of each other.

- **The rule:** Choose the standard deviation of weights w so that the *standard deviation* of ξ_j (denote by o_j) satisfies $o_j \approx 1$.

Normal LeCun initialization

- Assume the input data have the mean $= 0$ and the variance $= 1$.

  Consider a neuron j from the first layer with n inputs. Assume its weights chosen randomly by the normal distribution $\mathcal{N}(0, w^2)$.

  Assume that all random choices are independent of each other.
- **The rule:** Choose the standard deviation of weights w so that the *standard deviation* of ξ_j (denote by o_j) satisfies $o_j \approx 1$.
- Basic properties of the variance of independent variables give $o_j = \sqrt{n} \cdot w$.

  Thus by putting $w = \sqrt{\frac{1}{n}}$ we obtain $o_j = 1$.

Normal LeCun initialization

- Assume the input data have the mean $= 0$ and the variance $= 1$.

  Consider a neuron j from the first layer with n inputs. Assume its weights chosen randomly by the normal distribution $\mathcal{N}(0, w^2)$.

  Assume that all random choices are independent of each other.
- **The rule:** Choose the standard deviation of weights w so that the *standard deviation* of ξ_j (denote by o_j) satisfies $o_j \approx 1$.
- Basic properties of the variance of independent variables give $o_j = \sqrt{n} \cdot w$.

  Thus by putting $w = \sqrt{\frac{1}{n}}$ we obtain $o_j = 1$.
- The same works for higher layers, n corresponds to the number of neurons in the layer one level lower.

This gives *normal LeCun initialization*:

$$w_i \sim \mathcal{N}\left(0, \frac{1}{n}\right)$$

Normal Glorot initialization

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

Normal Glorot initialization

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

Glorot & Bengio (2010) presented a **normalized initialization** by choosing weights randomly from the following normal distribution:

$$N\left(0, \frac{2}{m+n}\right) = N\left(0, \frac{1}{(m+n)/2}\right)$$

Here n is the number of inputs to the layer, m is the number of neurons in the layer above.

Normal Glorot initialization

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

Glorot & Bengio (2010) presented a **normalized initialization** by choosing weights randomly from the following normal distribution:

$$N\left(0, \frac{2}{m+n}\right) = N\left(0, \frac{1}{(m+n)/2}\right)$$

Here n is the number of inputs to the layer, m is the number of neurons in the layer above.

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

This gives *normal Glorot initialization* (also called *normal Xavier initialization*):

$$w_i \sim \mathcal{N}\left((0, \frac{2}{m+n}\right)$$

Uniform LeCun initialization

- Assume that the input data have mean $= 0$ and variance $= 1$.

  Consider a neuron j from the first layer with n inputs. Assume its weights chosen randomly by the uniform distribution $U(-w, w)$.

  Assume that all random choices are independent of each other.

- As before, we want the standard deviation o_j of the inner potential ξ_j to be approximately 1.

- Basic properties of the variance of independent variables give $o_j = \sqrt{\frac{n}{3}} \cdot w$.

  Thus by putting $w = \sqrt{\frac{3}{n}}$ we obtain $o_j = 1$.

We obtain *uniform LeCun initialization*:

$$w_i \sim U\left(-\sqrt{\frac{3}{n}}, \sqrt{\frac{3}{n}}\right)$$

Uniform Glorot initialization

Similarly to the normal case, we want to normalize the initialization w.r.t. both forward and backward passes.

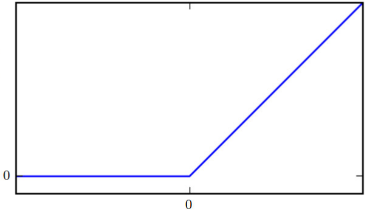
We obtain *uniform Glorot initialization* (aka *uniform Xavier init.*):

$$w_i \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) = U\left(-\sqrt{\frac{3}{(m+n)/2}}, \sqrt{\frac{3}{(m+n)/2}}\right)$$

Here n is the number of inputs to the layer, m is the number of neurons in the layer above.

Modern activation functions

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



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

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

Normal He initialization

- The ReLU is not as sensitive to the large variance of the inner potential as sigmoidal functions (large variance does not matter as much).

Normal He initialization

- The ReLU is not as sensitive to the large variance of the inner potential as sigmoidal functions (large variance does not matter as much).
- Still the variance is good to be constant (at least due to the output layer).

Normal He initialization

- The ReLU is not as sensitive to the large variance of the inner potential as sigmoidal functions (large variance does not matter as much).
- Still the variance is good to be constant (at least due to the output layer).
- LeCun initialization cannot be justified for ReLU due to the following reason:

  The ReLU is not a symmetric function. So even if the inner potential ξ_j has mean $= 0$ and variance $= 1$, it is not true of the output (the variance is halved).

Normal He initialization

- The ReLU is not as sensitive to the large variance of the inner potential as sigmoidal functions (large variance does not matter as much).
- Still the variance is good to be constant (at least due to the output layer).
- LeCun initialization cannot be justified for ReLU due to the following reason:

  The ReLU is not a symmetric function. So even if the inner potential ξ_j has mean $= 0$ and variance $= 1$, it is not true of the output (the variance is halved).

Modifying the normal LeCun initialization to take the halving variance into account, we obtain *normal He initialization*:

$$w_i \in \mathcal{N}\left(0, \frac{2}{n}\right) \qquad \left(\text{LeCun is } w_i \in \mathcal{N}\left(0, \frac{1}{n}\right)\right)$$

More modern activation functions

- Leaky ReLU (greenboard):
  - Generalizes ReLU, not dead for negative potentials.
  - Experimentally not much better than ReLU.

More modern activation functions

- Leaky ReLU (greenboard):
  - Generalizes ReLU, not dead for negative potentials.
  - Experimentally not much better than ReLU.
- ELU: "Smoothed" ReLU:

$$\sigma(\xi) = \begin{cases} \alpha(\exp(\xi) - 1) & \text{for } \xi < 0 \\ \xi & \text{for } \xi \geq 0 \end{cases}$$

  Here α is a parameter, ELU converges to $-\alpha$ as $\xi \to -\infty$. As opposed to ReLU: Smooth, always non-zero gradient (but saturates), slower to compute.

More modern activation functions

- Leaky ReLU (greenboard):
  - Generalizes ReLU, not dead for negative potentials.
  - Experimentally not much better than ReLU.
- ELU: "Smoothed" ReLU:

$$\sigma(\xi) = \begin{cases} \alpha(\exp(\xi) - 1) & \text{for } \xi < 0 \\ \xi & \text{for } \xi \geq 0 \end{cases}$$

  Here α is a parameter, ELU converges to $-\alpha$ as $\xi \to -\infty$. As opposed to ReLU: Smooth, always non-zero gradient (but saturates), slower to compute.
- SELU: Scaled variant of ELU: :

$$\sigma(\xi) = \lambda \begin{cases} \alpha(\exp(\xi) - 1) & \text{for } \xi < 0 \\ \xi & \text{for } \xi \geq 0 \end{cases}$$

  Self-normalizing, i.e. output of each layer will tend to preserve a mean (close to) 0 and a standard deviation (close to) 1 for $\lambda \approx 1.050$ and $\alpha \approx 1.673$, properly initialized weights (see below) and normalized inputs (zero mean, standard deviation 1).

Initializing with Normal Distribution

Denote by n the number of inputs to the initialized layer, and m the number of neurons in the layer.

- normal Glorot:

$$w_i \sim \mathcal{N}\left((0, \frac{2}{m+n}\right)$$

  Suitable for none, tanh, logistic, softmax

Initializing with Normal Distribution

Denote by n the number of inputs to the initialized layer, and m the number of neurons in the layer.

- normal Glorot:

  $$w_i \sim \mathcal{N}\left((0, \frac{2}{m+n}\right)$$

  Suitable for none, tanh, logistic, softmax

- normal He:

  $$w_i \in \mathcal{N}\left(0, \frac{2}{n}\right)$$

  Suitable for ReLU, leaky ReLU

Initializing with Normal Distribution

Denote by n the number of inputs to the initialized layer, and m the number of neurons in the layer.

- normal Glorot:

  $$w_i \sim \mathcal{N}\left((0, \frac{2}{m+n}\right)$$

  Suitable for none, tanh, logistic, softmax

- normal He:

  $$w_i \in \mathcal{N}\left(0, \frac{2}{n}\right)$$

  Suitable for ReLU, leaky ReLU

- normal LeCun:

  $$w_i \sim \mathcal{N}\left(0, \frac{1}{n}\right)$$

  Suitable for SELU (by the authors)

How to choose activation of hidden neurons

- The default is ReLU.
- According to Aurélien Géron:

$$\textit{SELU} > \textit{ELU} > \textit{leakyReLU} > \textit{ReLU} > \textit{tanh} > \textit{logistic}$$

For discussion see: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurélien Géron

Batch normalization (roughly)

Intuition: Instead of keeping mean $= 0$ and variance $= 1$ implicitly due to a clever weight initialization, we may **renormalize values of neurons** throughout the layers.

Batch normalization (roughly)

Intuition: Instead of keeping mean $= 0$ and variance $= 1$ implicitly due to a clever weight initialization, we may **renormalize values of neurons** throughout the layers.

Consider the ℓ-th layer of the network.

Note that the output values of neurons in the ℓ-th layer can be seen as inputs to the sub-network consisting of all layers above the ℓ-th one.

Batch normalization (roughly)

Intuition: Instead of keeping mean $= 0$ and variance $= 1$ implicitly due to a clever weight initialization, we may **renormalize values of neurons** throughout the layers.

Consider the ℓ-th layer of the network.

Note that the output values of neurons in the ℓ-th layer can be seen as inputs to the sub-network consisting of all layers above the ℓ-th one.

What if we standardize the values of the ℓ-th layer as we did with the input data?

For this we need to form a "dataset" of values of the ℓ-th layer.

Batch normalization (roughly)

Let us consider the ℓ-th layer with n neurons.

Consider a batch of training examples:

$$\{(\vec{x}_k, \vec{d}_k) \mid k = 1, \ldots, p\}$$

(This is typically a minibatch.)

Batch normalization (roughly)

Let us consider the ℓ-th layer with n neurons.

Consider a batch of training examples:

$$\{(\vec{x}_k, \vec{d}_k) \mid k = 1, \ldots, p\}$$

(This is typically a minibatch.)

- For every $k = 1, \ldots, p$: Compute the values of neurons in the ℓ-th layer for the input $\vec{x}_k$ and obtain a vector

$$\vec{z}_k = (\vec{z}_{k1}, \ldots, \vec{z}_{kn})$$

Batch normalization (roughly)

Let us consider the ℓ-th layer with n neurons.

Consider a batch of training examples:

$$\{(\vec{x}_k, \vec{d}_k) \mid k = 1, \ldots, p\}$$

(This is typically a minibatch.)

- For every $k = 1, \ldots, p$: Compute the values of neurons in the ℓ-th layer for the input $\vec{x}_k$ and obtain a vector

$$\vec{z}_k = (\vec{z}_{k1}, \ldots, \vec{z}_{kn})$$

- Set all components of all vectors $\vec{z}_k$ to the mean $= 0$ and the variance $= 1$ and obtain *normalized vectors*: $\hat{z}_1, \ldots, \hat{z}_p$.

Batch normalization (roughly)

Let us consider the ℓ-th layer with n neurons.

Consider a batch of training examples:

$$\{(\vec{x}_k, \vec{d}_k) \mid k = 1, \ldots, p\}$$

(This is typically a minibatch.)

- For every $k = 1, \ldots, p$: Compute the values of neurons in the ℓ-th layer for the input $\vec{x}_k$ and obtain a vector

  $$\vec{z}_k = (\vec{z}_{k1}, \ldots, \vec{z}_{kn})$$

- Set all components of all vectors $\vec{z}_k$ to the mean $= 0$ and the variance $= 1$ and obtain *normalized vectors*: $\hat{z}_1, \ldots, \hat{z}_p$.
- For every $k = 1, \ldots, p$ give

  $$\vec{\gamma} \cdot \hat{z}_k + \vec{\delta}$$

  as the output of the ℓ-th layer instead of $\vec{z}_k$. Here $\vec{\gamma}$ and $\vec{\delta}$ are new trainable weights.

Generalization

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

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

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

Generalization

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

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

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

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

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

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

The network should fit g_j not the noise.

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

Regularization

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

Regularization

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

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

... and I ask you prof. Neumann:

What can you fit with 40GB of parameters??

Early stopping

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

When to stop?

Early stopping

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

When to stop?

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

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

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

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

Early stopping

Divide your dataset into several subsets:

- **training set** (e.g. 60%) – train the network here
- **validation set** (e.g. 20%) – use to stop the training
- **test set** (e.g. 20%) – use to evaluate the final model

What to use as a stopping rule?

Early stopping

Divide your dataset into several subsets:

- **training set** (e.g. 60%) – train the network here
- **validation set** (e.g. 20%) – use to stop the training
- **test set** (e.g. 20%) – use to evaluate the final model

What to use as a stopping rule?

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

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

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

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

Size of the network

Similar problem as in the case of the training duration:

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

Solution: Optimal number of neurons :-)

Size of the network

Similar problem as in the case of the training duration:

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

Solution: Optimal number of neurons :-)

- there are some (useless) theoretical bounds
- there are algorithms dynamically adding/removing neurons (not much use nowadays)
- In practice: Start with an existing network solving similar problem.

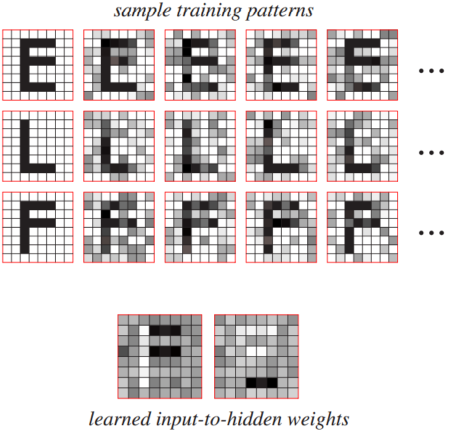
  If you are trully desperate trying to solve a brand new problem, you may try an ancient rule of thumb: the number of neurons $\approx$ ten times less than the number of training instances.

  Experiment, experiment, experiment.

Feature extraction

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

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



Ensemble methods

Techniques for reducing generalization error by combining several models.

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

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

Ensemble methods

Techniques for reducing generalization error by combining several models.

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

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

Bagging:

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

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

Dropout

The algorithm: In every step of the gradient descent

- choose randomly a set N of neurons, each neuron is included in N independently with probability $1/2$,

  (in practice, different probabilities are used as well).

- do forward and backward propagations only using the selected neurons

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

Dropout

The algorithm: In every step of the gradient descent

- choose randomly a set N of neurons, each neuron is included in N independently with probability $1/2$,

  (in practice, different probabilities are used as well).

- do forward and backward propagations only using the selected neurons

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

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

Dropout is not exactly the same as bagging: The models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.
In the case of bagging, each model is trained to convergence on its respective training set. This would be infeasible for large networks/training sets.

Dropout – details

- The inner potential of a neuron j **without dropout**:

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

- The inner potential of a neuron j **with dropout**:

$$r_i \sim \mathtt{Bernoulli}(1/2) \quad \text{for all} \quad i \in j_{\leftarrow} \smallsetminus \{0\}$$

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

  (Intuitively, randomly chosen neurons are masked out.)

- During inference do not drop out neurons and multiply values of neurons with $1/2$.

  This compensates for the fact that without the drop out there are twice as many neurons.

Weight decay and L2 regularization

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

Weight decay and L2 regularization

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

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

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

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

Weight decay and L2 regularization

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

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

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

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

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

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

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

More optimization, regularization ...

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

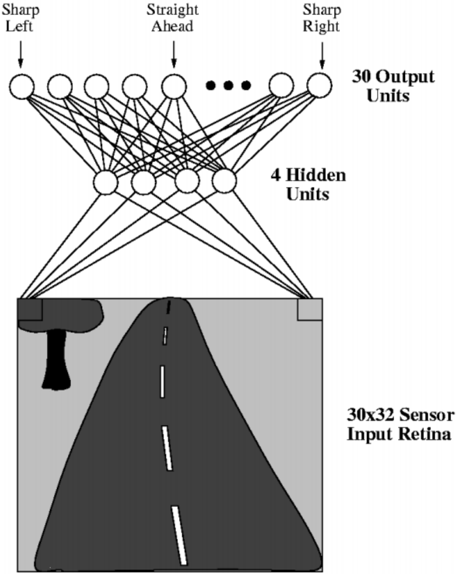
For a very nice survey see

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

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

Some applications

ALVINN (history)



Architecture:

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

ALVINN

Architecture:

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

Activity:

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

ALVINN

Learning: Trained during (live) drive.

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

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

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

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

ALVINN – Selection of training samples

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

ALVINN – Selection of training samples

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

Problems:

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

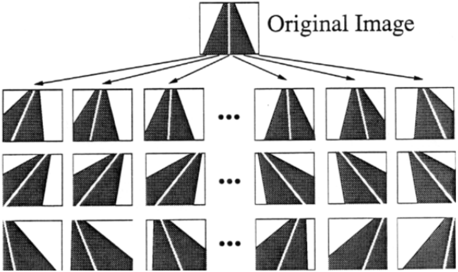
ALVINN – Selection of training examples

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

ALVINN – Selection of training examples

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

- 15 distorted copies of each image:

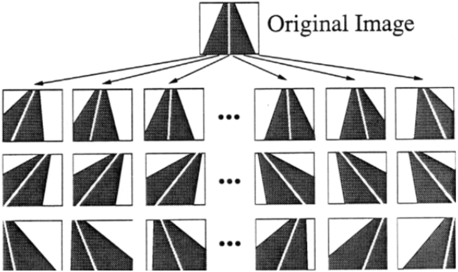


- desired output generated for each copy

ALVINN – Selection of training examples

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

- 15 distorted copies of each image:



- desired output generated for each copy

"Boring" images solved as follows:

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

ALVINN - learning

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

Results:

- Trained for 5 minutes, speed 4 miles per hour.
- ALVINN was able to drive well on a new road it has never seen (in different weather conditions).

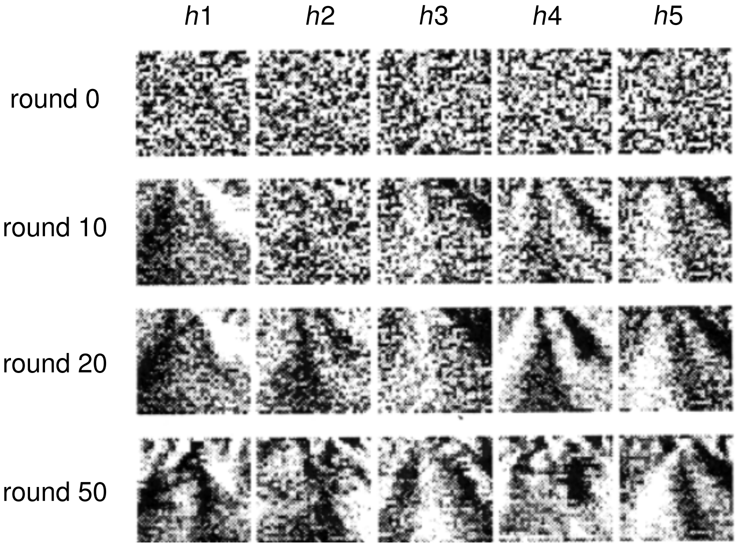
ALVINN - learning

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

Results:

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

ALVINN - weight development



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

MNIST – handwritten digits recognition

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

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

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

MNIST – handwritten digits recognition

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

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

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

Allows comparison of various methods.

MNIST

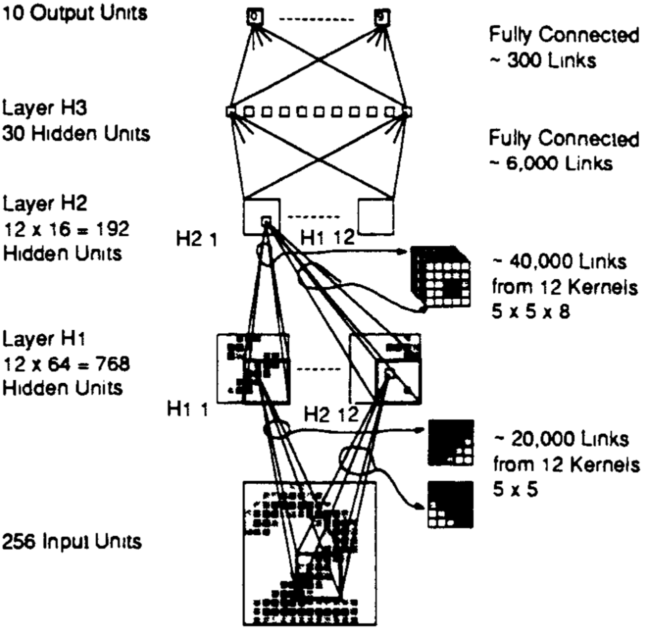
One of the best "old" results is the following:

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

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

A famous application of a learning convolutional network LeNet-1 in 1998.

MNIST – LeNet1



MNIST – LeNet1

Interpretation of output:

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

Learning:

Inputs:

- training on 7291 samples, tested on 2007 samples

Results:

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

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

Modern convolutional networks

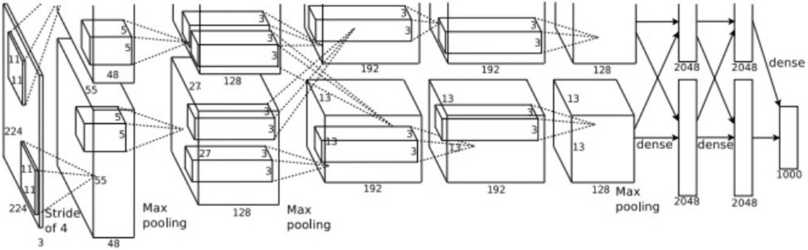
The rest of the lecture is based on the online book Neural Networks and Deep Learning by Michael Nielsen.
`http://neuralnetworksanddeeplearning.com/index.html`

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

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

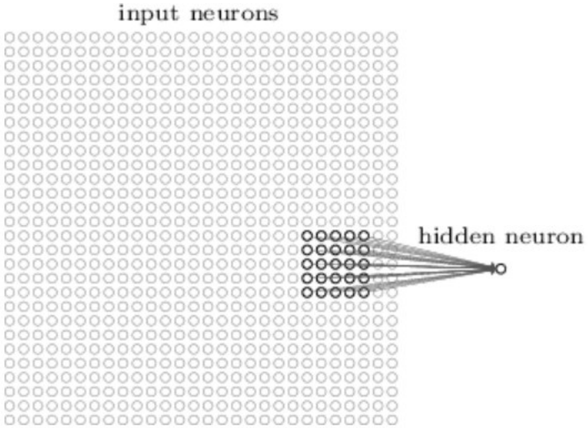
AlexNet

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



A convolutional network, trained on two GPUs.

Convolutional networks - local receptive fields

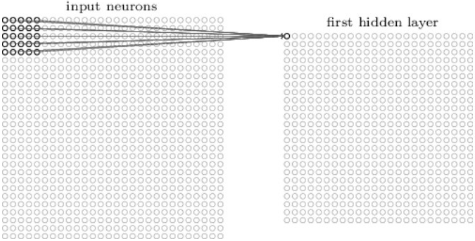


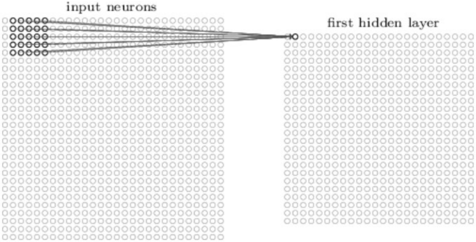
Every neuron is connected with a field of $k \times k$ (in this case 5×5) neurons in the lower layer (this filed is *receptive field*).

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

Convolutional networks - stride length

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

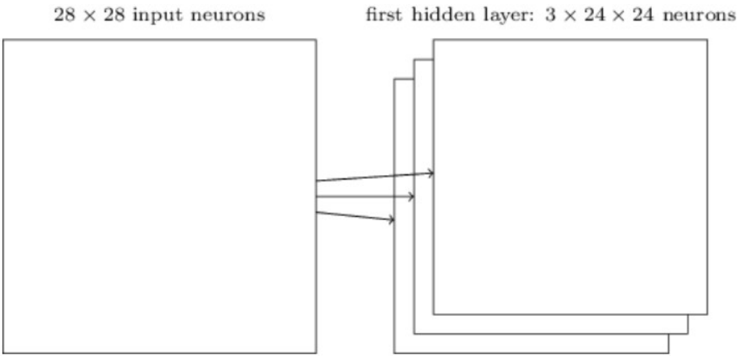




The "size" of the slide is called *stride length*.

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

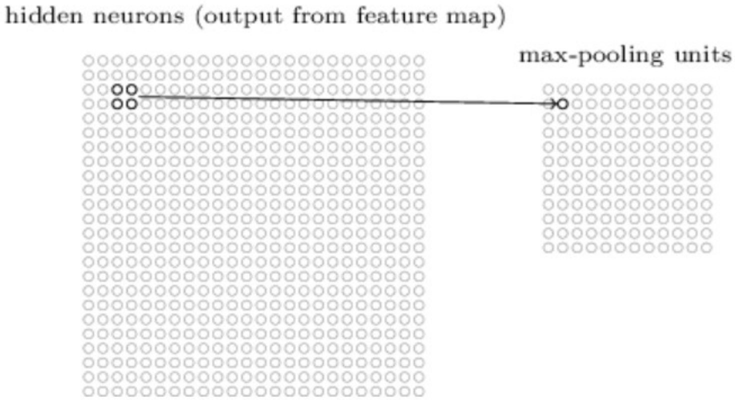
Feature maps



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

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

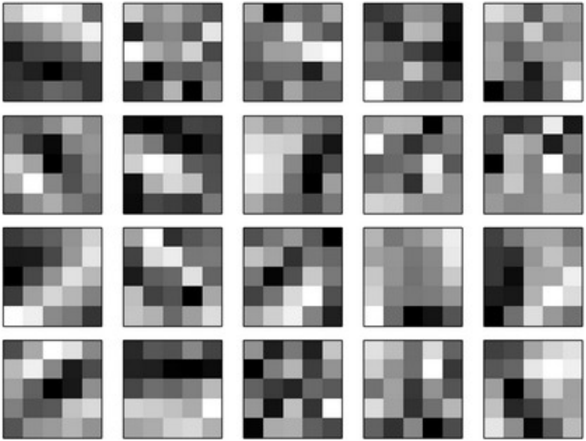
Pooling



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

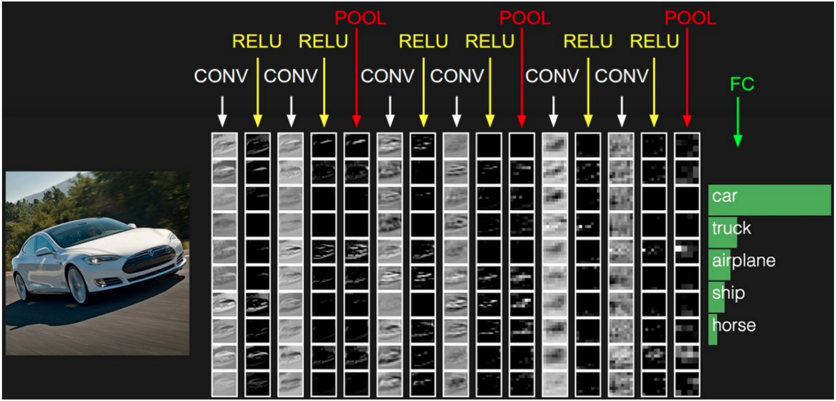
- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- ...

Trained feature maps

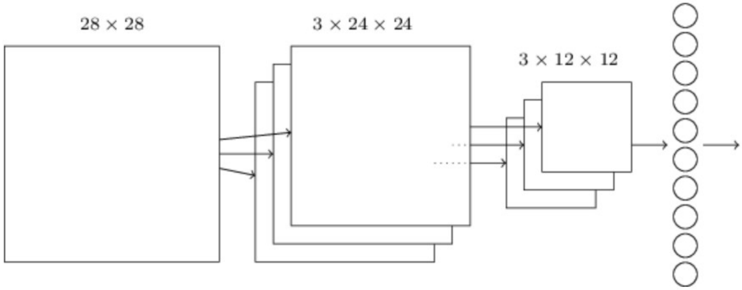


(20 feature maps, receptive fields 5×5)

Trained feature maps



Simple convolutional network

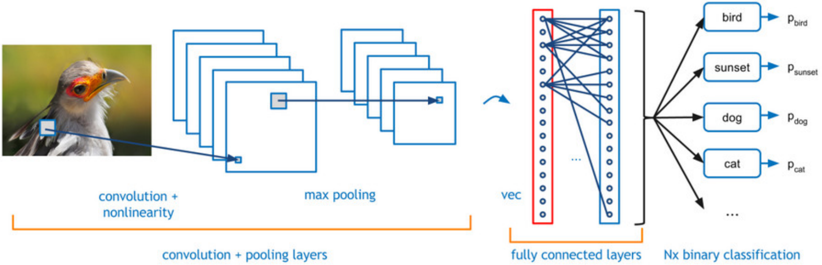


28×28 input image, 3 feature maps, each feature map has its own max-pooling (field 5×5, stride = 1), 10 output neurons.

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

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

Convolutional network



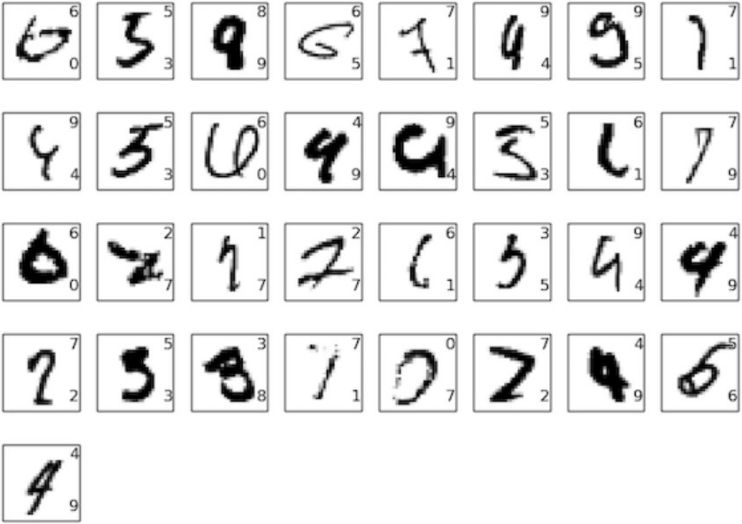
Simple convolutional network vs MNIST

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

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

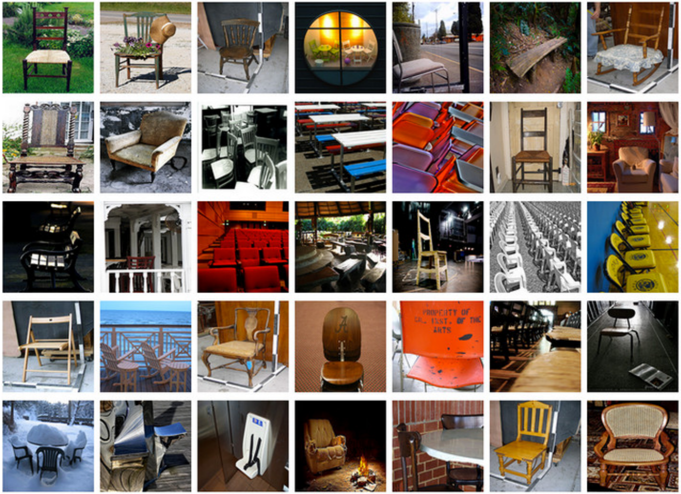
MNIST

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



More complex convolutional networks

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



ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

Competition in classification over a subset of images from ImageNet.

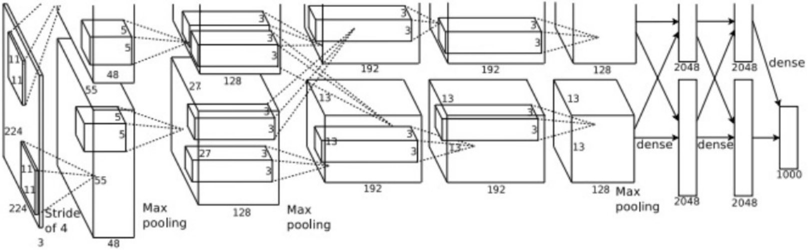
Started in 2010, assisted in breakthrough in image recognition.

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

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

AlexNet

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



Trained on two GPUs (NVIDIA GeForce GTX 580)

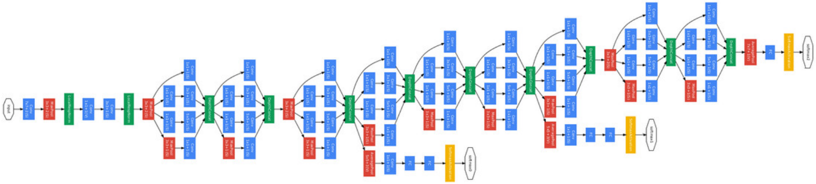
Výsledky:

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

ILSVRC 2014

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

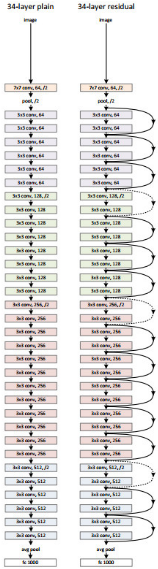
GoogLeNet: deep convolutional network, 22 layers



Results:

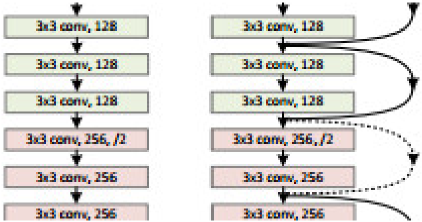
- Accuracy 93.33% top-5

ILSVRC 2015



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

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



ILSVRC 2016

Trimps-Soushen (The Third Research Institute of Ministry of Public Security)

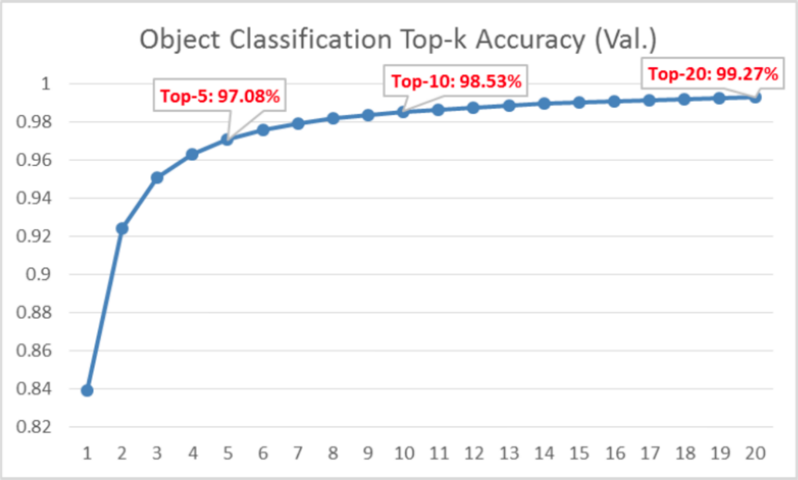
There is no new innovative technology or novelty by Trimps-Soushen.

Ensemble of the pretrained models from Inception-v3, Inception-v4, Inception-ResNet-v2, Pre-Activation ResNet-200, and Wide ResNet (WRN-682).

Each of the models are strong at classifying some categories, but also weak at classifying some categories.

Test error: 2.99%

Top-k accuracy analyzed



https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Top-20 typical errors

Out of 1458 misclassified images in Top-20:

Error Categories	Numbers	Percentages(%)
Label May Wrong	221	15.16
Multiple Objects (>5)	118	8.09
Non-Obvious Main Object	355	24.35
Confusing Label	206	14.13
Fine-grained Label	258	17.70
Obvious Wrong	234	16.05
Partial Object	66	4.53

https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Top-k accuracy analyzed

Predict:
1 *pencil box*
2 *diaper*
3 *bib*
4 *purse*
5 *running shoe*

Ground Truth:
sleeping bag

https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Top-k accuracy analyzed

Predict:
1 *dock*
2 *submarine*
3 *boathouse*
4 *breakwater*
5 *lifeboat*

Ground Truth:
paper towel

https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Top-k accuracy analyzed

Predict:

1 *bolete*

2 *earthstar*

3 *gyromitra*

4 *hen of the woods*

5 *mushroom*

Ground Truth:

stinkhorn

https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Top-k accuracy analyzed

Predict:
1 *apron*
2 *plastic bag*
3 *sleeping bag*
4 *umbrella*
5 *bulletproof vest*

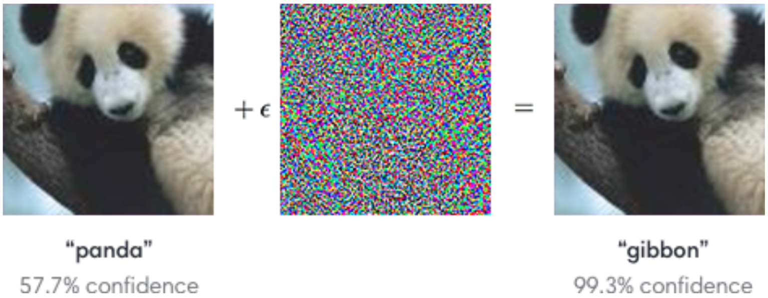
Ground Truth:
poncho

https://towardsdatascience.com/review-trimps-soushen-winner-in-ilsvrc-2016-image-classification-dfbc423111dd

Superhuman convolutional nets?!

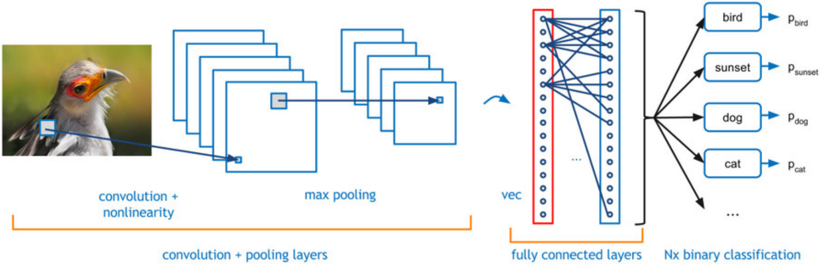
Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought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

Does it really work?

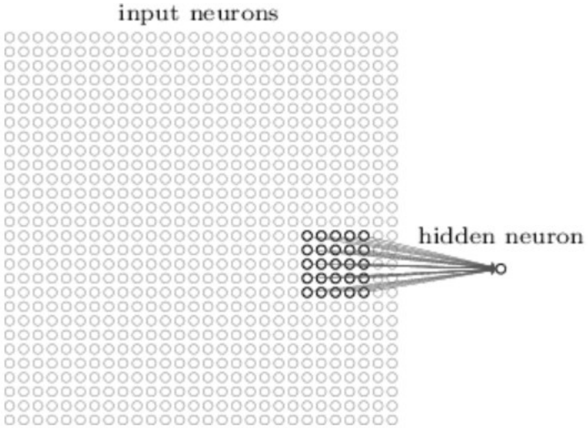


Convolutional networks – theory

Convolutional network



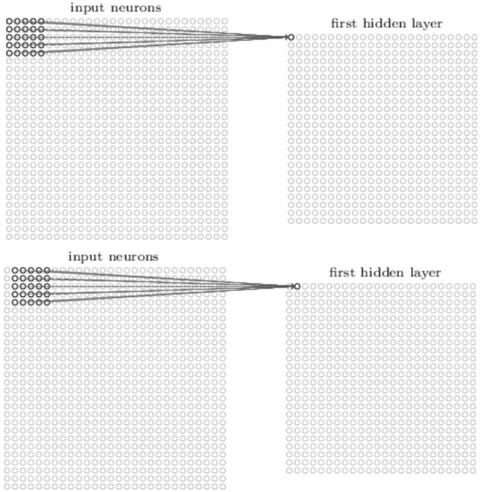
Convolutional layers



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

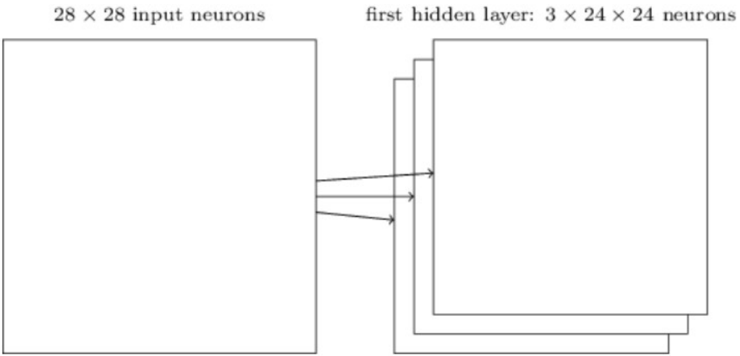
Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

Convolutional layers



Neurons grouped into *feature maps* sharing weights.

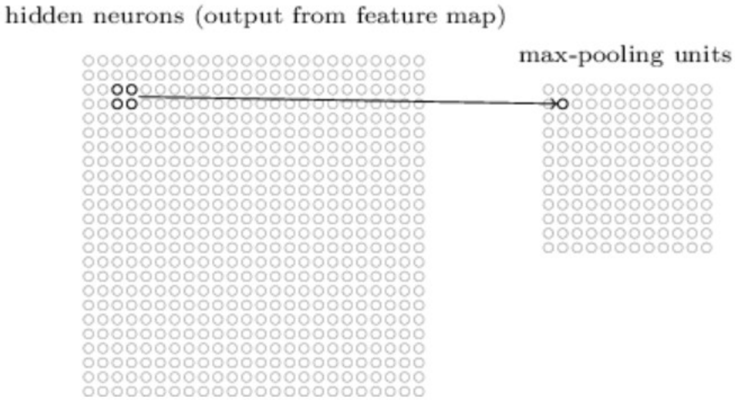
Convolutional layers



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

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

Pooling layers



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

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

Convolutional networks – architecture

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

Convolutional networks – architecture

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

Several types of layers:

- **input** layer L_0

Convolutional networks – architecture

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

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.

Convolutional networks – architecture

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

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional** layer L_m: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)

Convolutional networks – architecture

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

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional** layer L_m: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)
- **pooling** layer: "Neurons" organized into **pooling maps**, all neurons
  - compute a simple aggregate function (such as max),
  - have *disjoint inputs*.

Pooling after convolution is applied to each feature map separately.

I.e. a single pooling map after each feature map.

Convolutional networks – architecture

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

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

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

Convolutional networks – activity

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

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

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

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

Convolutional networks – activity

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

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

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

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

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

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

  - avg-pooling:

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

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

Convolutional networks – learning

Learning:

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

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

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

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

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

  where

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

Convolutional networks – SGD

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

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

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

    where

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

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

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

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

Backprop

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

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

Backprop

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

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

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

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

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

Backprop

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

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

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

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

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

- Now for every w_{ji} where j is in a convolutional layer:

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

- Neurons of pooling layers do not have weights.

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

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

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

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

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

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

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

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

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

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

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

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

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

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

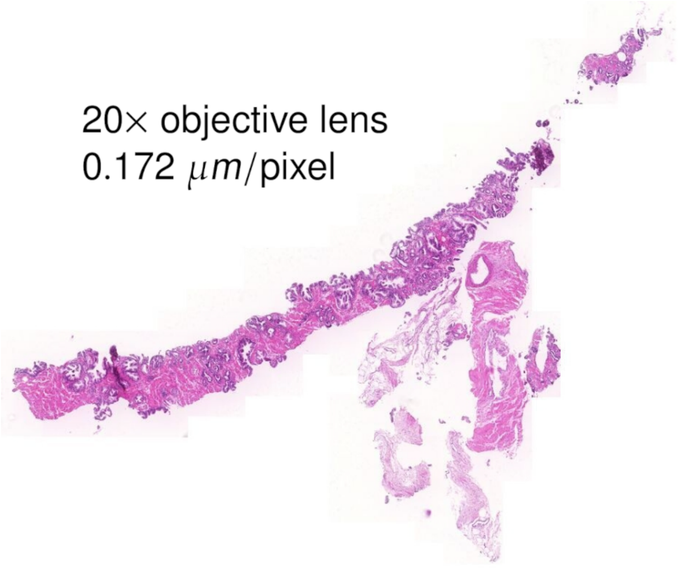
$$\frac{\partial E_k}{\partial y_j} = \begin{cases} \frac{\partial E_k}{\partial y_i} & \text{if } j = arg\ max_{r \in i_{\leftarrow}} y_r \\ 0 & \text{otherwise} \end{cases}$$

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

Convolutional networks – summary

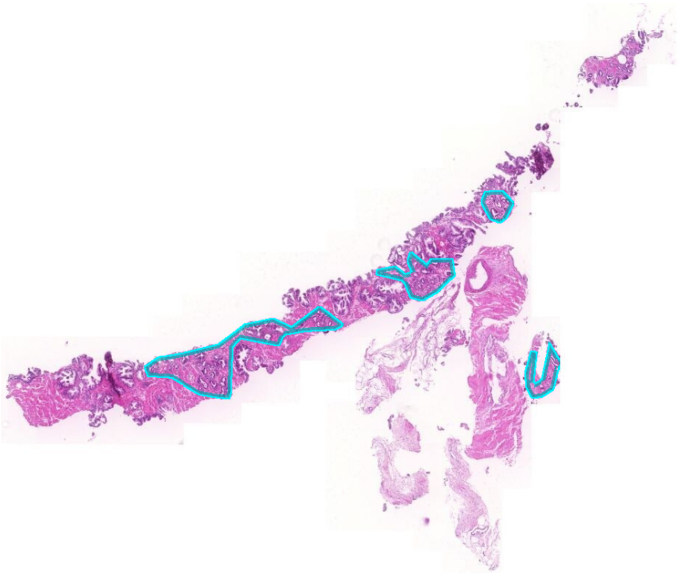
- Conv. nets. are nowadays the most used networks in image processing (and also in other areas where input has some local, "spatially" invariant properties)
- Typically trained using the gradient descent.
- Due to the weight sharing allow (very) deep architectures.
- Typically extended with more adjustments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

The problem of cancer detection in WSI



The problem: Detect cancer in this image.

The problem of cancer detection in WSI

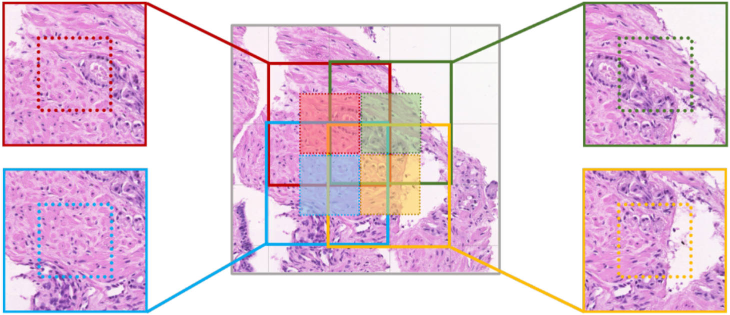


- WSI annotated by pathologists, **not** pixel level precise!

Input data

WSI too large, 105,185 px Œ 221,772 px

Cut into patches of size 512 px Œ 512 px



Patch positive **iff** the inner square intersects the annotation

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

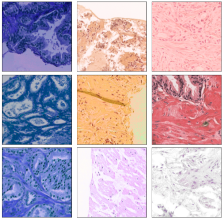
- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

Dataset augmentation:



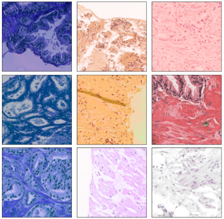
- random vertical and horizontal flips
- random color perturbations

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

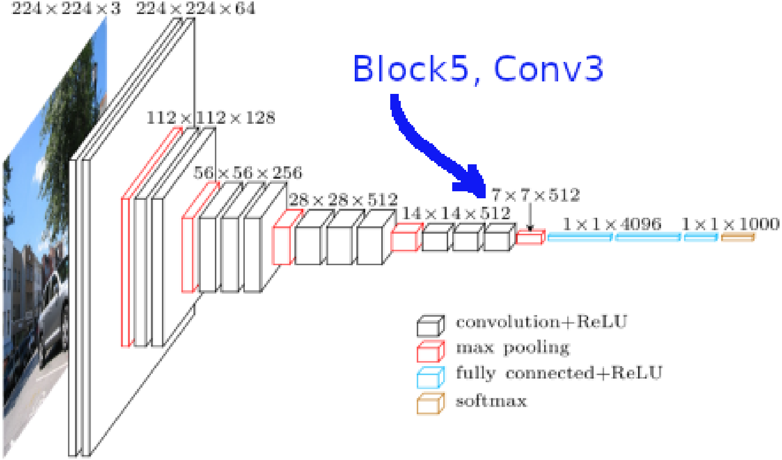
Dataset augmentation:



- random vertical and horizontal flips
- random color perturbations

- Training data three step sampling:
  1. randomly select a label
  2. randomly select a slide containing at least a single patch with the label
  3. randomly select a patch with the label from the slide

VGG16



3×3 convolutions, stride 1, padding 1. Max pooling 2×2, stride 2.

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch
- The error E = cross-entropy

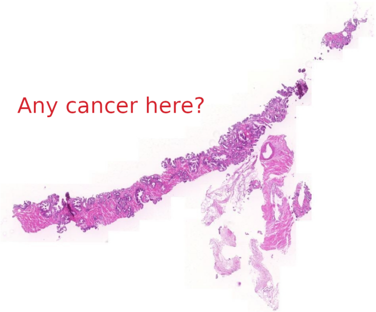
Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch
- The error E = cross-entropy
- Training:
  - RMSprop optimizer
  - The "forgetting" hyperparameter: $\rho = 0.9$
  - The initial learning rate 5×10^{-5}
    - If no improvement in E on validation data for 3 consecutive epochs $\Rightarrow$ half the learning rate
    - If no improvement in ROCAUC on validation data for 5 consecutive epochs $\Rightarrow$ terminate
  - Momentum with the weight $\alpha = 0.9$

Prediction

Model evaluation - attempt 1

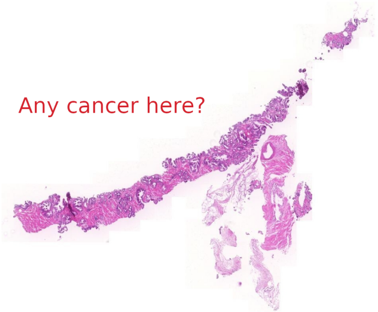
Can we detect cancer somewhere in WSI?



Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?

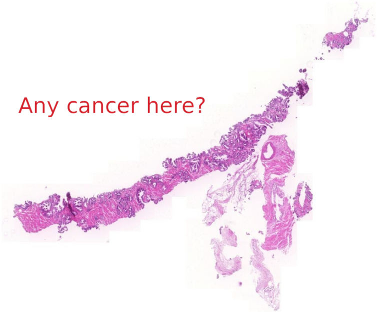


Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Interpret the $F(I)$ as the probability of cancer in the patch.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?



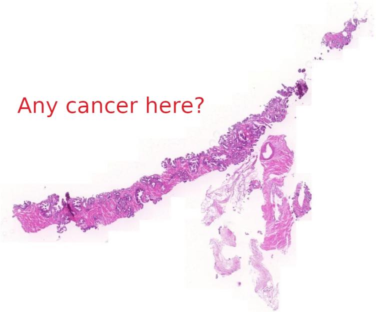
Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Interpret the $F(I)$ as the probability of cancer in the patch.

Predict WSI positive iff at least one patch I satisfies $F(I) \geq t$ for a fixed threshold $t \in [0, 1]$.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?



Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Interpret the $F(I)$ as the probability of cancer in the patch.

Predict WSI positive iff at least one patch I satisfies $F(I) \geq t$ for a fixed threshold $t \in [0, 1]$.

Choosing t close to 1, we have achieved 100% accuracy, i.e., slide positive iff predicted positive. Problem solved ... No?

Model evaluation - attempt 2

Can we detect cancer in patches?

Predict I positive iff $F(I) \geq 0.75$

Single WSI:

		PREDICTED Pos	PREDICTED Neg
TRUE	Pos	805	18
	Neg	48	614

All WSIs:

		PREDICTED Pos	PREDICTED Neg
TRUE	Pos	24796	4340
	Neg	5345	158754

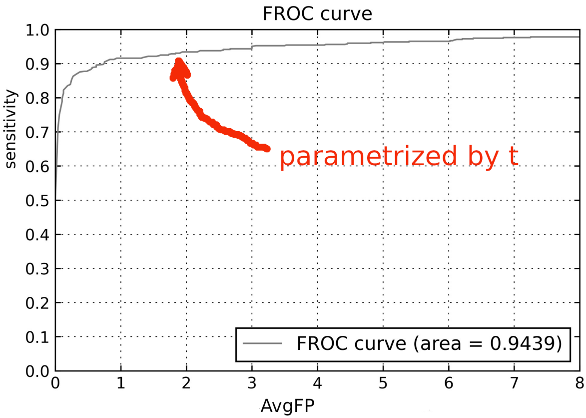
Ok, does it detect cancer?

Model evaluation – attempt 3 – FROC

Detect *particular tumors* ?

How to evaluate the quality of tumor detection?

Model evaluation – attempt 3 – FROC



sensitivity ≈ the proportion of tumors containing at least one patch I with $F(I) \geq t$ w.r.t. all tumors in all slides

AvgFP ≈ average number of patches I with $F(I) \geq t$ in *each non-cancerous slide*

Explainable methods (XAI)

XAI methods

The goal is to understand how and why the network does what it does.

We will consider classification models only.

XAI methods

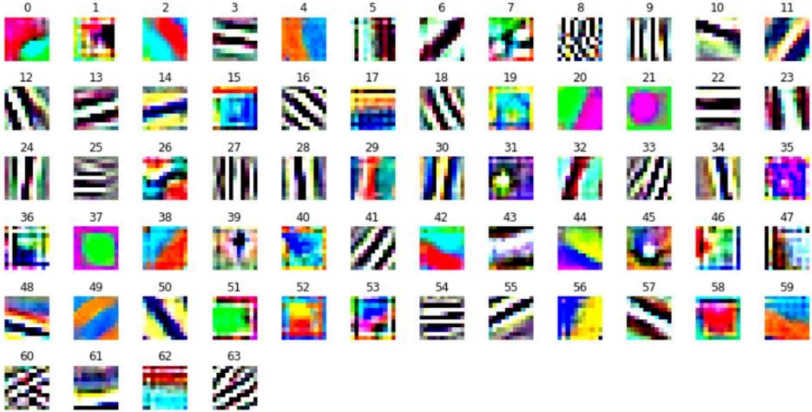
The goal is to understand how and why the network does what it does.

We will consider classification models only.

Methods based on various principles:

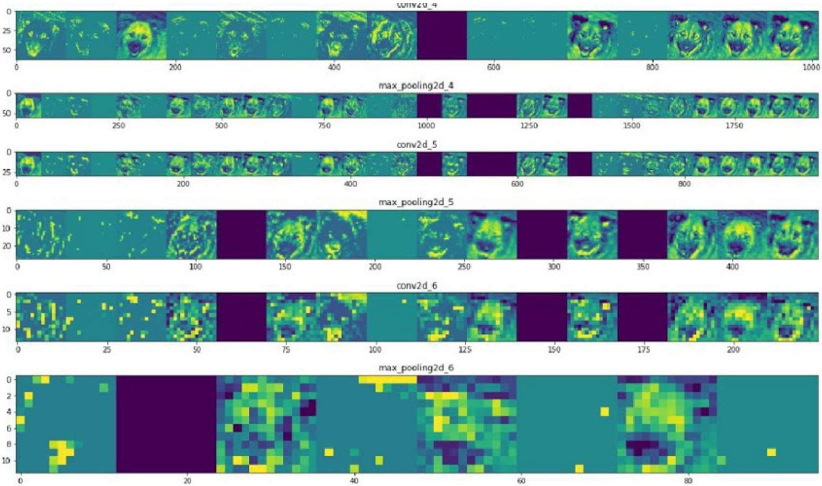
- Visualize weights and feature maps
- Visualize most important inputs for a given class
- Visualize the effect of input perturbations on the output
- Construct an intepretable surrogate model

Alex-net - filters of the first convolutional layer

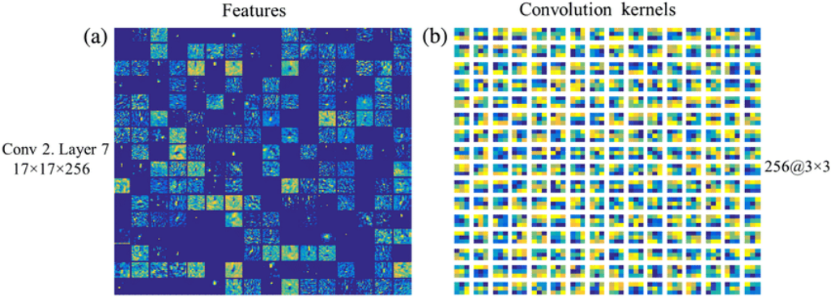


- 64 filters of depth 3 (RGB)
- Combined each filter RGB channels into one RGB image of size 11x11x3.

CNN - feature maps



CNN - feature maps - radar target classification



Synthetic-aperture radar (SAR) – used to create two-dimensional images or three-dimensional reconstructions of objects, such as landscapes.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $\xi_i(\vec{x})$ the inner potential of the *output* neuron $i \in Y$ given a network input vector $\vec{x}$.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $\xi_i(\vec{x})$ the inner potential of the *output* neuron $i \in Y$ given a network input vector $\vec{x}$.
- Maximize

$$\xi_i(\vec{x}) - \lambda \left\|\vec{x}\right\|_2^2$$

  over all input vectors $\vec{x}$.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $\xi_i(\vec{x})$ the inner potential of the *output* neuron $i \in Y$ given a network input vector $\vec{x}$.
- Maximize

  $$\xi_i(\vec{x}) - \lambda \left\|\vec{x}\right\|_2^2$$

  over all input vectors $\vec{x}$.
- A maximizing input vector computed using the gradient descent.
- Gives the most "representative" input vector of the class represented by the neuron i.

Maximizing input - example

dumbbell cup dalmatian

Input specific saliency maps

The goal: Label features in a given input that are "most important" for the output of the network.

Input specific saliency maps

The goal: Label features in a given input that are "most important" for the output of the network.

Various approaches:

- gradient based
  - Gradient saliency maps
  - GradCAM
  - ...
- occlusion based
  - Simple occlusion maps
  - LIME
  - ...

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $\xi_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $\xi_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

  For every input neuron $k \in X$ we consider

  $$\left|\frac{\partial \xi_i}{\partial y_k}(\vec{x})\right|$$

  to measure the importance of the input y_k for the output potential ξ_i with respect to the particular input vector $\vec{x}$.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $\xi_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

  For every input neuron $k \in X$ we consider

  $$\left|\frac{\partial \xi_i}{\partial y_k}(\vec{x})\right|$$

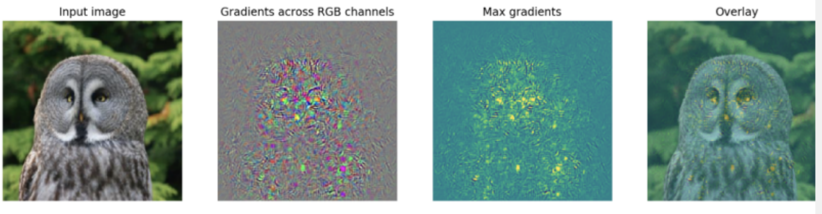
  to measure the importance of the input y_k for the output potential ξ_i with respect to the particular input vector $\vec{x}$.
- Note that saliency comes from a surrogate local linear model given by the first-order Taylor approximation:

  $$\xi_i(\vec{x}') \approx \xi_i(\vec{x}) + \left(\frac{\partial \xi_i}{\partial X}(\vec{x})\right)(\vec{x}' - \vec{x})$$

  Here $\frac{\partial \xi_i}{\partial X}$ is the vector of all partial derivatives $\frac{\partial \xi_i}{\partial y_k}$ where $k \in X$.

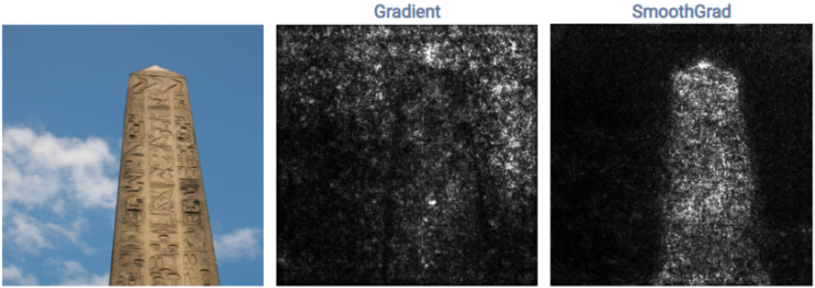
Saliency maps - example

Saliency maps - example



Quite noisy, the signal is spread and does not say much about the perception of the owl.

Saliency maps - example



SmoothGrad:

- Do the following several times:
  - Add noise to the input image
  - Compute a saliency map
- Average the resulting saliency maps.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \ldots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \dots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.
- Fix an output neuron $i \in Y$ with the value y_i.

  Compute the *average importance* of $F^\ell(I)$:

$$\alpha_i^\ell = \frac{1}{|F^\ell|} \sum_{j \in F^\ell} \frac{\partial \xi_i}{\partial y_j}(I)$$

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \ldots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.
- Fix an output neuron $i \in Y$ with the value y_i.

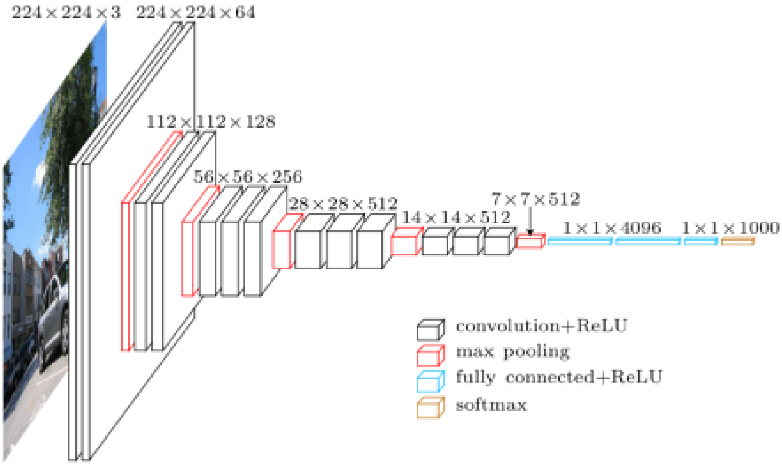
  Compute the *average importance* of $F^\ell(I)$:

$$\alpha_i^\ell = \frac{1}{|F^\ell|} \sum_{j \in F^\ell} \frac{\partial \xi_i}{\partial y_j}(I)$$

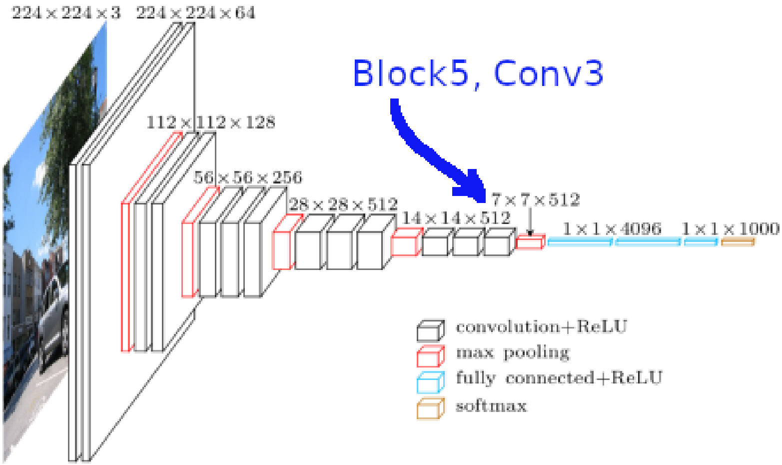
  and the final *gradCAM heat map* for L is obtained using

$$M_i^L = \mathtt{ReLU}\left(\sum_{\ell=1}^{k} \alpha_i^\ell F^\ell(I)\right)$$

GradCAM on VGG16

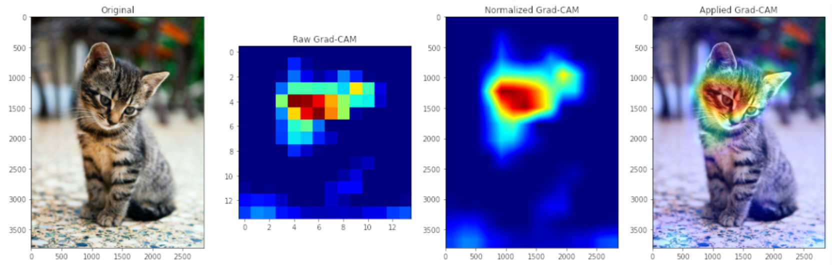


GradCAM on VGG16



Consider the last convolutional layer of the VGG16 (Block5, Conv3)

GradCAM on VGG16



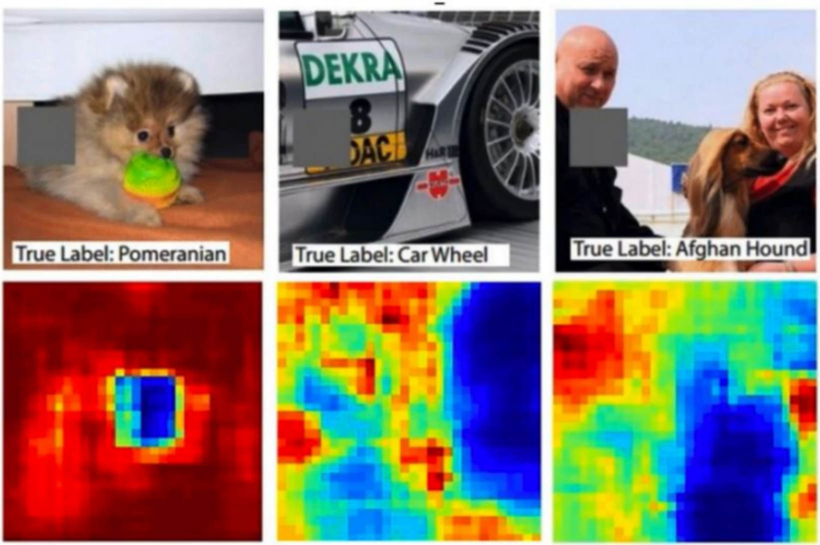
From left to right:

- An image of a cat (has to be resized to 224×224 to fit VGG16)
- The gradCAM heat map for the last convolutional layer and the class "cat"
- Rescaled and smoothed gradCAM heat map.
- The gradCAM overlay.

Occlusion

- Systematically cover parts of the input image.
- Observe the effect on the output value.
- Find regions with the largest effect.

Occlusion - example



Occlusion - example

['harmonica, mouth organ, harp, mouth harp']

LIME - for images

Let us fix an image I to be explained.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.
- Construct a linear model approximating the network aroung the image I with weights corresponding to the superpixels.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.
- Construct a linear model approximating the network aroung the image I with weights corresponding to the superpixels.
- Select the superpixels with weights of large magnitude as the important ones.

Original Image

Interpretable Components

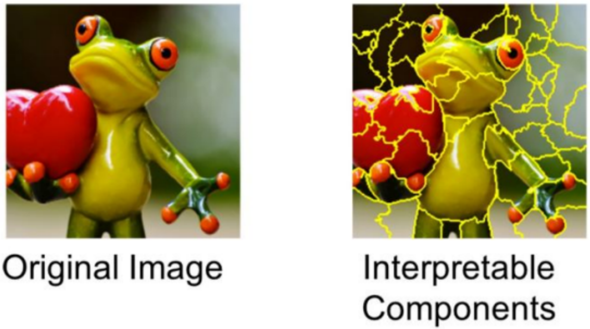
Superpixels as interpretable components

Original Image

Interpretable Components

Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Superpixels as interpretable components



Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Consider binary vectors $\vec{x} = (x_1, \ldots, x_\ell) \in \{0, 1\}^\ell$.

Superpixels as interpretable components

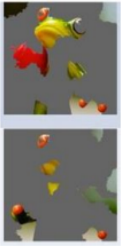
Original Image

Interpretable Components

Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Consider binary vectors $\vec{x} = (x_1, \ldots, x_\ell) \in \{0, 1\}^\ell$.

Each such vector $\vec{x}$ determines a "subimage" $I[\vec{x}]$ of I obtained by removing all P_k with $x_k = 0$.



LIME

- Let us fix an output neuron *i*, we dnote by $\xi_i(J)$ the inner potential of the output neuron *i* for the input image *J*.

LIME

- Let us fix an output neuron i, we dnote by $\xi_i(J)$ the inner potential of the output neuron i for the input image J.
- Given the image I to be interpreted, consider the following training set:

$$\mathcal{T} = \left\{(\vec{x}_1, \xi_i(I[\vec{x}_1])), \ldots, (\vec{x}_p, \xi_i(I[\vec{x}_p])\right\}$$

Here $\vec{x}_h = (x_{h1}, \ldots, x_{h\ell})$ are (some) binary vectors of $\{0, 1\}$. E.g., randomly selected.

LIME

- Let us fix an output neuron i, we dnote by $\xi_i(J)$ the inner potential of the output neuron i for the input image J.
- Given the image I to be interpreted, consider the following training set:

$$\mathcal{T} = \left\{(\vec{x}_1, \xi_i(I[\vec{x}_1])), \ldots, (\vec{x}_p, \xi_i(I[\vec{x}_p])\right\}$$

  Here $\vec{x}_h = (x_{h1}, \ldots, x_{h\ell})$ are (some) binary vectors of $\{0, 1\}$. E.g., randomly selected.
- Train a linear model (ADALINE) with weights $w_0, w_1, \ldots, w_\ell$ on $\mathcal{T}$.

  Intuitively, the linear model approximates the network on "subimages" of I obtained by removing some superpixels.
- Inspect the weights (magnitude and sign).

LIME

More precisely, we train a linear model (ADALINE) F with weights $\vec{w} = w_0, w_1, \ldots, w_\ell$ on $\mathcal{T}$ minimizing the weighted mean-squared error

$$\mathcal{E}(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} \pi_k \cdot (F(\vec{x}_k) - \xi_i(I[\vec{x}_k]))^2 + \Omega(\vec{w})$$

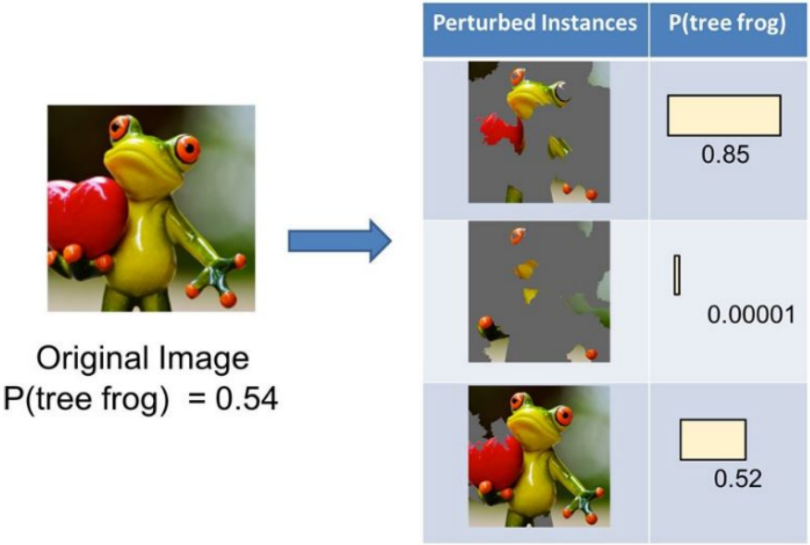
where

- the weights are defined by

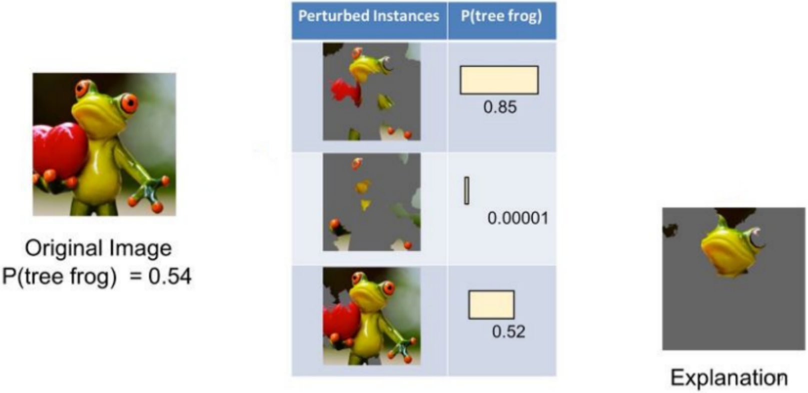
$$\pi_k = \exp\left(\frac{-(\, 1 - \sqrt{1 - (s_k/\ell)}\,)^2}{2\nu^2} \right)$$

  Here s_k is the number of elements in $\vec{x}_k$ equal to zero, ℓ is the number of superpixels.
- $\Omega(\vec{w})$ is a regularization term making the number of non-zero weights as small as possible.

LIME - example



LIME - example



LIME - example

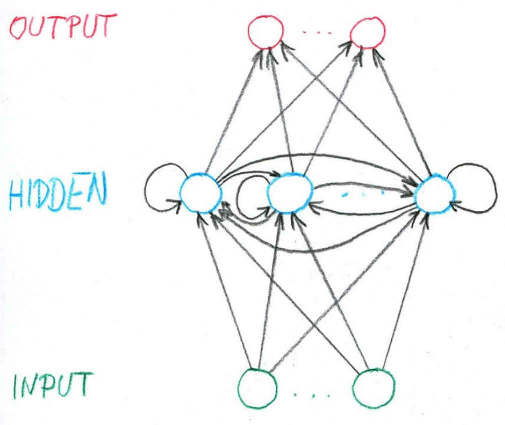
(a) Original Image (b) Explaining *Electric guitar* (c) Explaining *Acoustic guitar* (d) Explaining *Labrador*

LIME - example

(a) Husky classified as wolf (b) Explanation

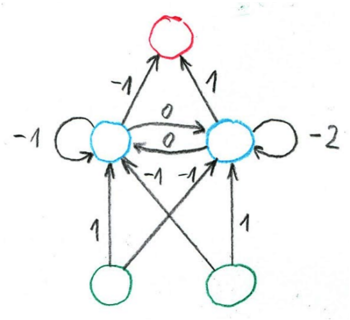
Recurrent Neural Networks - LSTM

RNN



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

RNN example

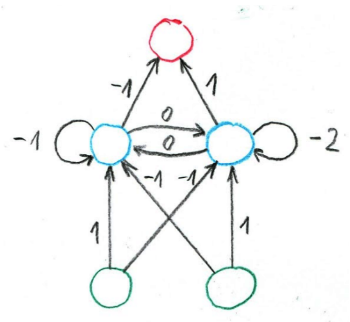


Activation function:

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

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

RNN example

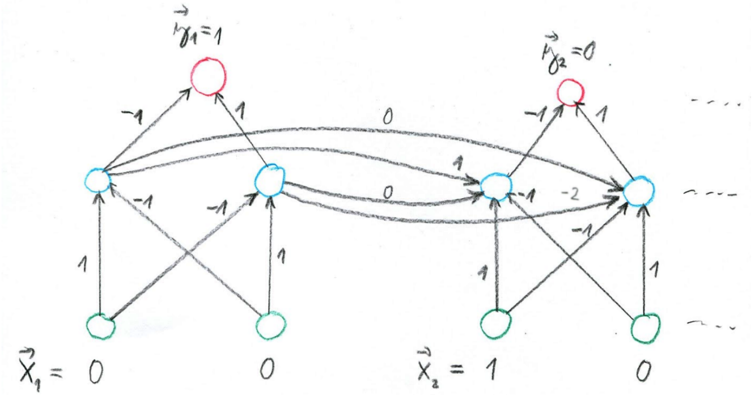


Activation function:

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

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

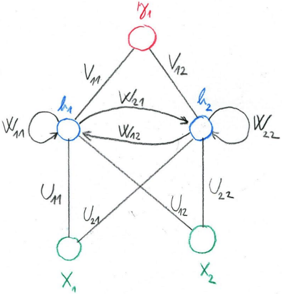
RNN example



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

RNN – formally

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



RNN – formally

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

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

RNN – formally

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

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

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

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

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

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

RNN – formally

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

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

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

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

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

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

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

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

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

RNN – in matrix form

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

RNN – in matrix form

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

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

and

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

RNN – in matrix form

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

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

  and

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

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

  $$y_t = \sigma(Vh_t)$$

RNN – Comments

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

RNN – training

Training set

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

here

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

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

Error function

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

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

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

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

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

Gradient descent (single training example)

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

The algorithm computes a sequence of weight matrices as follows:

Gradient descent (single training example)

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

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.

Gradient descent (single training example)

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

The algorithm computes a sequence of weight matrices as follows:

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

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

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

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

Gradient descent (single training example)

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

The algorithm computes a sequence of weight matrices as follows:

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

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

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

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

The above is THE learning algorithm that modifies weights!

Backpropagation

Computes the derivatives of E, no weights are modified!

Backpropagation

Computes the derivatives of E, no weights are modified!

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

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

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

Backpropagation

Computes the derivatives of E, no weights are modified!

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

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

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

Backpropagation:

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

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

Long-term dependencies

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

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

  LSTM is currently a bit obsolete. The main idea is to decompose W into several matrices, each responsible for a different task. One is concerned about memory, one is concerned about the output at each step, etc.

https://arxiv.org/pdf/2205.13504.pdf

LSTM

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

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

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

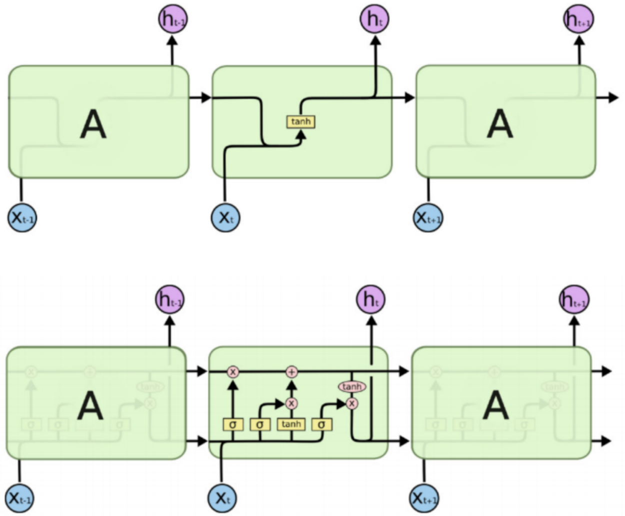
$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t) \qquad \text{output gate}$$

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

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

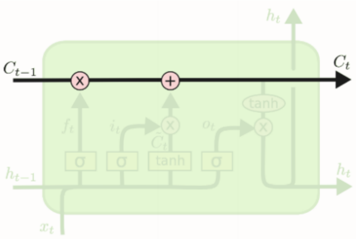
- $\circ$ is the component-wise product of vectors
- $\cdot$ is the matrix-vector product
- σ_h hyperbolic tangents (applied component-wise)
- σ_g logistic sigmoid (aplied component-wise)

RNN vs LSTM



Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM

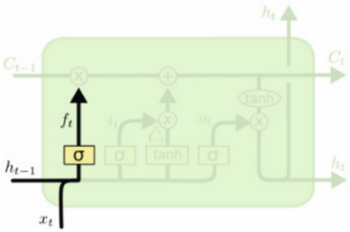


$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$
$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$
$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$
$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM

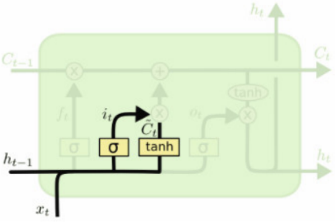


$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$
$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$
$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$
$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

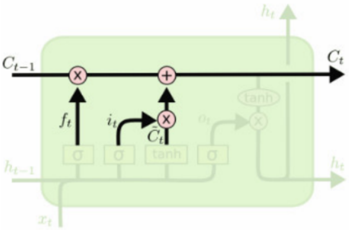
LSTM



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

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$
$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$
$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

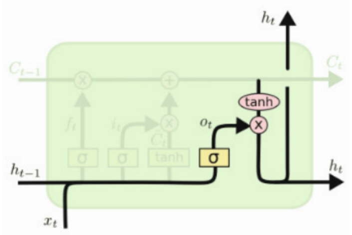


$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$
$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$
$$\Rightarrow \tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$
$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$
$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



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

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

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

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

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

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

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

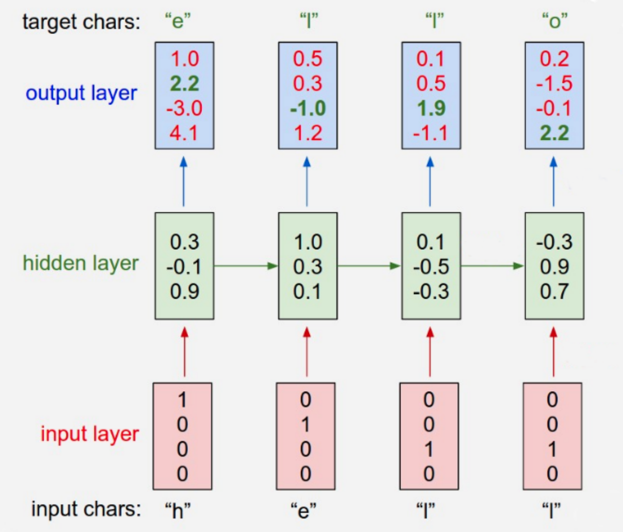
LSTM – summary

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

... but the development goes on ...

RNN text generator

Generating texts letter by letter.



Shakespeare

- Generating Shakespeare letter by letter.
- Trained on Shakespeare's plays (4.4MB).

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy master's ready there My power to give thee but so much as hell: Some service in the noble bondman here, Would show him to her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

Wikipedia

- Hutter Prize 100MB dataset from Wikipedia (96MB)

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servicious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be to the [[Punjab Resolution]] (PJS)[`http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm` Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a

Xml halucination:


```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">
    #REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

LaTeX

- Algebraic geometry textbook.
- LaTeX source (16MB).
- Almost compilable.

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma **??**. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of

Linux source code

- Trained on all source files of Linux kernel concatenated into a single file (474MB of C code).

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

```c
/*
 *  Copyright (c) 2006-2010, Intel Mobile Communications.  All rights reserved.
 *
 *   This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 *        This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *
 *  GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software Foundation,
 *  Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>
```

Evolution of Shakespeare

100 iter.:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

300 iter.:

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

500 iter.:

```
we counter. He stutn co des. His stanted out one ofler that concossions and was
to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn
```

700 iter.:

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

1200 iter.:

```
"Kite vouch!" he repeated by her
door. "But I would be done and quarts, feeling, then, son is people...."
```

2000 iter.:

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

Attention

Consider the following task: Given a sequence of vectors

$$\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$$

generate a new sequence

$$\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_{T'}$$

of possibly different length (i.e., possibly $T \neq T'$).

E.g., a machine translation task, **x** is an embedding of an English sentence, **y** is an embedding of a Czech sentence.

Attention

Consider two recurrent networks:

- Enc the encoder
  - Hidden state $\vec{h}_0$ initialized by standard methods for recurrent networks
  - Reads $\vec{x}_1, \ldots, \vec{x}_T$, does not output anything but produces a sequence of hidden states $\vec{h}_1, \ldots, \vec{h}_T$
- Dec the decoder
  - The initial hidden state is $\vec{h}_T$
  - Does not read anything but outputs the sequence $\vec{y}_1, \ldots, \vec{y}_{T'}$

Trained on pairs of sequences, able to learn a fine translation between major languages (if the recurrent networks are LSTM).

Is not perfect because all info about $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$ is squeezed into the single state vector $\vec{h}_T$.

In particular, the network tends to forget the context of each word.

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

The decoder `Dec` is still a recurrent network but

- the hidden state $\vec{h}'_0$ initialized by standard methods for recurrent networks, a sequence of hidden states $\vec{h}'_0, \ldots, \vec{h}'_{T'}$ is computed,

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

The decoder `Dec` is still a recurrent network but

- the hidden state $\vec{h}'_0$ initialized by standard methods for recurrent networks, a sequence of hidden states $\vec{h}'_0, \ldots, \vec{h}'_{T'}$ is computed,
- reads a sequence of context vectors $\vec{c}_1, \ldots, \vec{c}_{T'}$ where

$$c_i = \sum_{j=1}^{T} \alpha_{ij} \vec{h}_j \qquad \text{where} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

  where $e_{ij} = \mathtt{MLP}(\vec{h}'_{i-1}, \vec{h}_j)$
- outputs the sequence $\vec{y}_1, \ldots, \vec{y}_{T'}$

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.
- Is there a reason for reading the input sequence sequentially?

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.
- Is there a reason for reading the input sequence sequentially?
- Could we remove the recurrent network itself and preserve only the attention?

Self-Attention Layer (is all you need)

Fix an input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

Consider three learnable matrices: W_q, W_k, W_v

Generate a sequences of *queries, keys, and values*:

- $\vec{q}_1, \ldots, \vec{q}_T$ where $\vec{q}_k = W_q \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{k}_1, \ldots, \vec{k}_T$ where $\vec{k}_k = W_k \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{v}_1, \ldots, \vec{v}_T$ where $\vec{v}_k = W_v \vec{x}_k$ for all $k = 1, \ldots, T$

Define a vector score for all $i, j \in \{1, \ldots, T\}$ by

$$s_{ij} = \vec{q}_i \cdot \vec{k}_j$$

Intuitively, s_{ij} measures how much the input at the position i is related to the input at the position j, in other words, how much the query fits the key.

Define a sequence of outputs $\vec{y}_1, \ldots, \vec{y}_T$ by

$$\vec{y}_i = \sum_{j=1}^{T} \left(\mathtt{softmax}\left(\vec{s}_i / \sqrt{d_{attn}}\right)\right)_j \cdot \vec{v}_j$$

where $\vec{s}_i = (s_{i1}, \ldots, s_{iT})$ and d_{attn} is the dimension of $\vec{v}_i$.

Masked Self-Attention Layer (is all you need)

Assume an attention mechanism which given an input sequence $\vec{x}_1, \dots, \vec{x}_T$ generates $\vec{y}_1, \dots, \vec{y}_T$.

The Problem: How to generate $\vec{y}_k$ only based on $\vec{x}_1, \dots, \vec{x}_{k-1}$?

Define a vector score for all $i, j \in \{1, \dots, T\}$ by

$$s_{ij} = \begin{cases} \vec{q}_i \cdot \vec{k}_j & \text{if } j < i \\ -\infty & \text{otherwise.} \end{cases}$$

Define a sequence of outputs $\vec{y}_1, \dots, \vec{y}_T$ by

$$\vec{y}_i = \sum_{j=1}^{T} \left(\mathtt{softmax}\left(\vec{s}_i / \sqrt{d} \right) \right)_j \cdot \vec{v}_j$$

where $\vec{s}_i = (s_{i1}, \dots, s_{iT})$ and d is the dimension of $\vec{v}_i$.

Multi-head Self-Attention Layer (is all you need)

Assume the number of *heads* is H.

For $h = 1, \ldots, H$ the h-th head is an attention mechanism which given the input $\vec{x}_1, \ldots, \vec{x}_T$ produces

$$\vec{y}_1^h, \ldots, \vec{y}_T^h$$

Note that the output may be different which means that, in particular, the matrices W_q, W_k, W_v may be different for each head.

Assume that all vectors $\vec{y}_k^h$ are of the same dimension d_{mid} and consider a learnable matrix W_{out} of dimensions $d_{out} \times (H \cdot d_{mid})$.

The multi-head attention produces the following output:

$$\vec{y}_1, \ldots, \vec{y}_T$$

where

$$\vec{y}_k = W_{out} \cdot \left(\vec{y}_k^1 \odot \vec{y}_k^2 \odot \cdots \vec{y}_k^H \right)$$

Here $\odot$ is a concatenation of vectors.

Multi-head Attention Summary

Input: A sequence $\vec{x}_1, \dots, \vec{x}_T$
Output: A sequence $\vec{y}_1, \dots, \vec{y}_T$

I.e., a sequence of the same length. The dimensions of $\vec{y}_k$ and $\vec{x}_k$ do not have to be equal.

Attention:

Learnable parameters: Matrices W_q, W_k, W_v.

These matrices are used to compute queries, keys, and values from $\vec{x}_1, \dots, \vec{x}_T$ and $\vec{z}_1, \dots, \vec{z}_T$. Output $\vec{y}_1, \dots, \vec{y}_T$ is computed using values "scaled" by the query-key attention.

Multi-head attention:

Learnable parameters:

- Matrices W_q^h, W_k^h, W_v^h where $h = 1, \dots, H$ and H is the number of heads.

  Each attention head operates independently on the input $\vec{x}_1, \dots, \vec{x}_T$.
- Matrix W_{out}.

  Linearly transforms the concatenated results of the attention heads.

Positional encoding

The Goal: To encode a position (index) $k \in \{1, \ldots, T\}$ into a vector $\vec{P}_k$ of real numbers.

Assume that $\vec{P}_k$ should have a dimension d.
Given a position $k \in \{1, \ldots, T\}$ and $i \in \{0, \ldots, d/2$ define

$$P_{k,2i} = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P_{k,(2i+1)} = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Here $n = 10000$.

A user defined constant, the original paper suggests $n = 10000$.

Given an input sequence $\vec{x}_1, \ldots, \vec{x}_T$ we add the position embedding to each $\vec{x}_k$ obtaining a new input sequence $\vec{x}'_1, \ldots, \vec{x}'_T$ where

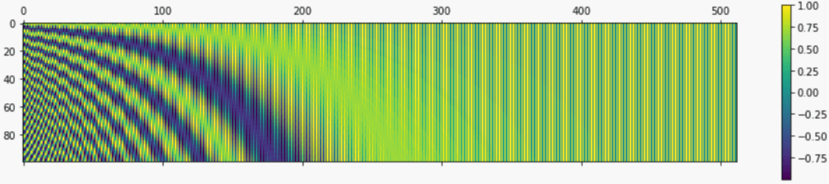
$$\vec{x}'_k = \vec{x}_k + P_k$$

Positional encoding

Positional Encoding Matrix with d=4, n=100

Sequence	Index of token, k	i=0	i=0	i=1	i=1
I	0	$P_{00}=\sin(0)$ $= 0$	$P_{01}=\cos(0)$ $= 1$	$P_{02}=\sin(0)$ $= 0$	$P_{03}=\cos(0)$ $= 1$
am	1	$P_{10}=\sin(1/1)$ $= 0.84$	$P_{11}=\cos(1/1)$ $= 0.54$	$P_{12}=\sin(1/10)$ $= 0.10$	$P_{13}=\cos(1/10)$ $= 1.0$
a	2	$P_{20}=\sin(2/1)$ $= 0.91$	$P_{21}=\cos(2/1)$ $= -0.42$	$P_{22}=\sin(2/10)$ $= 0.20$	$P_{23}=\cos(2/10)$ $= 0.98$
Robot	3	$P_{30}=\sin(3/1)$ $= 0.14$	$P_{31}=\cos(3/1)$ $= -0.99$	$P_{32}=\sin(3/10)$ $= 0.30$	$P_{33}=\cos(3/10)$ $= 0.96$

Positional encoding



The positional encoding matrix for n=10,000, d=512, sequence length=100

Layer normalization

Given a vector $\vec{v} \in \mathbb{R}^d$, the *layer normalization* computes:

$$\vec{v}' = \gamma \cdot \frac{(\vec{v} - \mu)}{\sigma} + \beta$$

Here

- $\mu = \frac{1}{d} \sum_{i=1}^{d} v_i$ and $\sigma^2 = \frac{1}{d} \sum_{i=1}^{d} (v_i - \mu)^2$
- $\gamma, \beta \in \mathbb{R}^d$ are vectors of trainable parameters

In Transformer:
The input to the layer normalization is a sequence of vectors: $\vec{x}_1, \ldots, \vec{x}_T$. The layer normalization is applied to each $\vec{x}_k$, producing a sequence of "normalized" vectors.

Language Model

A sequence of *tokens* $u_1, \dots, u_T \in V^*$

E.g. a corpus, tokens are words from a vocabulary V.

The goal: Desing a model maximizing

$$\sum_{k=\ell+1}^{T} P(u_k \mid u_{k-\ell}, \dots, u_{k-1}; W)$$

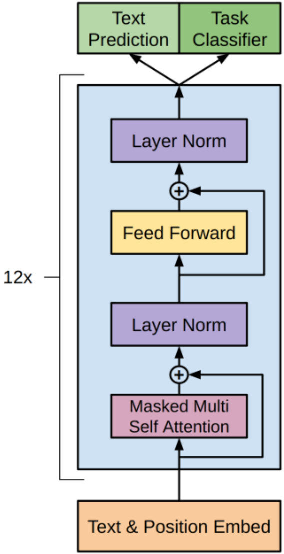
where

- P is the conditional probability measure over V modeled using a neural network with weights W.
- ℓ is the size of the window into which the network "looks".

Can be used to generate text:

Given tokens $v_1, \dots, v_\ell$, sample $v_{\ell+1}$ from $P(v_{\ell+1} \mid v_1, \dots, v_\ell)$

GPT - transformer



A sequence of one-hot encodings of tokens $\vec{u}_1, \ldots, \vec{u}_\ell \in \{0, 1\}^{|V|}$

Embed to vectors and add the position encoding:

$$\vec{x}_k = W_e \cdot \vec{u}_k + P_k \in Rset^d$$

Apply 12x the transformer block to $\vec{x}_1, \ldots, \vec{x}_\ell$ and obtain $\vec{y}_1, \ldots, \vec{y}_\ell$

Sample the next token from

$$u_{\ell+1} \sim \mathtt{softmax}(W_u \cdot \vec{y}_\ell)$$

Here $W_u \in \mathbb{R}^{|V| \times d}$ is a so called *unembedding matrix*.

Feed-forward networks summary

Architectures:

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

Training:

- gradient descent algorithm + heuristics