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  (Extremely well-written online textbook (a little outdated))
- Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

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

  ("Classical" overview of the theory of neural networks (a little outdated))
- Probabilistic Machine Learning: An Introduction by Kevin Murphy
  https://probml.github.io/pml-book/book1.html

  (Greatly advanced ML textbook with (almost) up-to-date basic neural networks.)
- Infinitely many online tutorials on everything (to build intuition)

Suggested: deeplearning.ai courses by Andrew Ng

Course organization

Evaluation:

- Project (Dr. Tomáš Foltýnek)
  - implementation of a selected model + analysis of given data
  - implementation C/C++/Java/Rust **without the 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 (Dr. Tomáš Foltýnek)
  - implementation of a selected model + analysis of given data
  - implementation C/C++/Java/Rust **without the 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 can we not use specialized libraries in projects?

FAQ

Q: Why can we not 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 can we not 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 (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 (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 (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 learn their functionality from data
  (... and thus do not need to be programmed.)

Machine learning in general

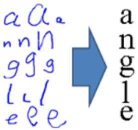
- Machine learning = construction of systems that learn their functionality from data

  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labeled" emails
    - consequently can distinguish spam from ham

Machine learning in general

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

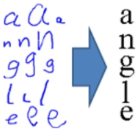
  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labeled" emails
    - consequently can distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labeled by their correct meaning
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



Machine learning in general

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

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

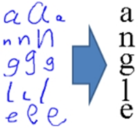


  - . . .
  - and lots of much, much more sophisticated applications ...

Machine learning in general

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

  (... and thus do not need to be programmed.)
  - spam filter
    - learns to recognize spam from a database of "labeled" emails
    - consequently can distinguish spam from ham
  - handwritten text reader
    - learns from a database of handwritten letters (or text) labeled 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.

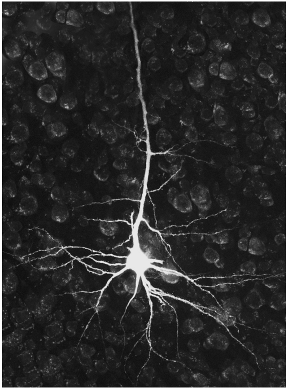
There are many types of models:

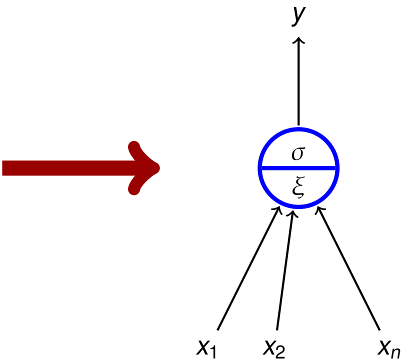
- decision trees
- support vector machines
- hidden Markov models
- Bayes networks and other graphical models
- **neural networks**
- ...

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 the brain receives information?
  - How the information is stored?
  - How the brain develops?
  - . . .

Why artificial neural networks?

Modelling of biological neural networks (computational neuroscience).

- simplified mathematical models help to identify important mechanisms
  - How the brain receives information?
  - How the information is stored?
  - How the 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 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 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 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 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.

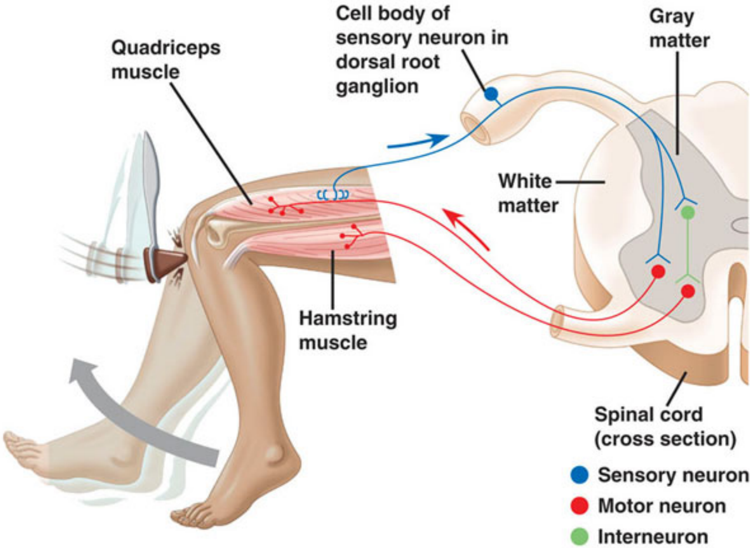
Biological neural network

- 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

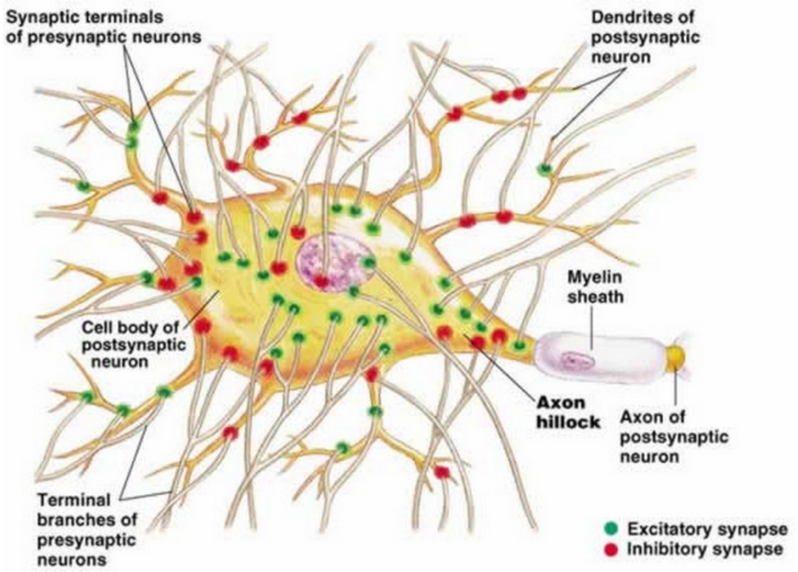
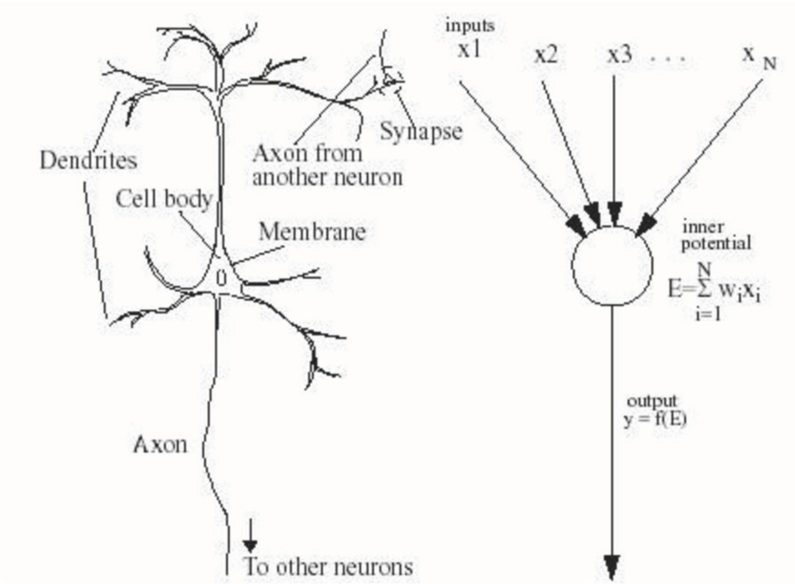
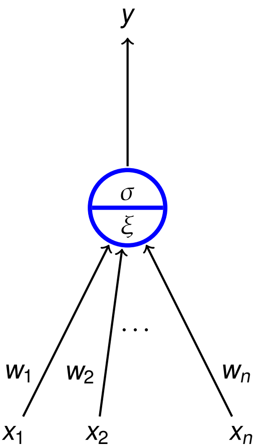


Figure 48.11(a), page 972, Campbell's *Biology, 5th Edition*

Biological and Mathematical neurons

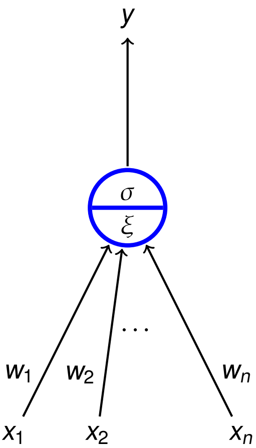


Formal neuron (without bias)



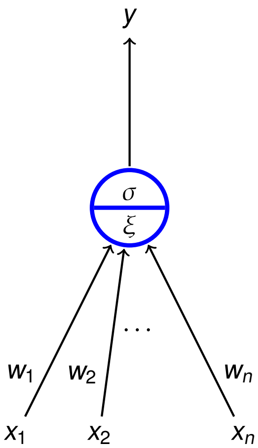
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**

Formal neuron (without bias)



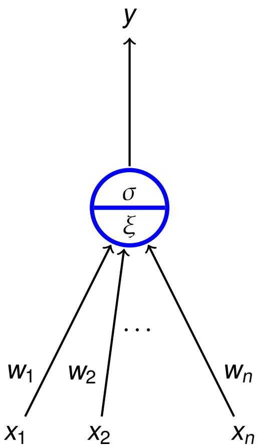
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**

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)



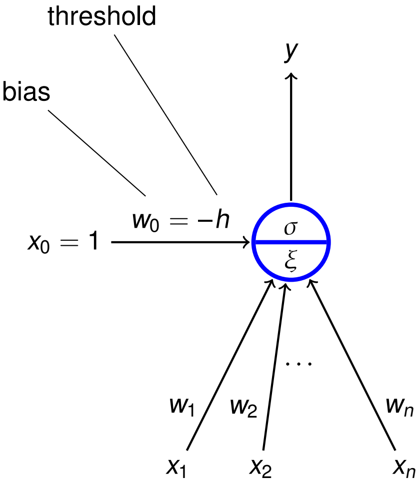
- $x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**
- $w_1, \ldots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**;
  almost always $\xi = \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$
  where σ is an **activation function**;

  e.g. a *unit step function*

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

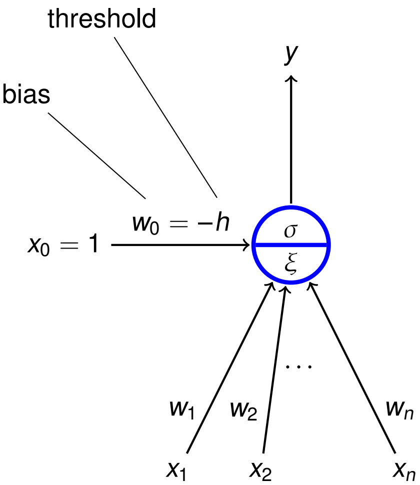
where $h \in \mathbb{R}$ is a *threshold*.

Formal neuron (with bias)



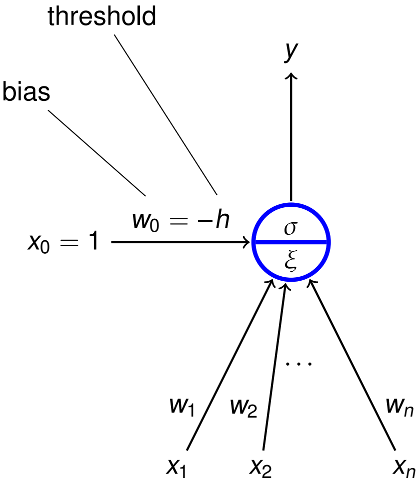
- $x_0 = 1, x_1, \ldots, x_n \in \mathbb{R}$ are **inputs**

Formal neuron (with bias)



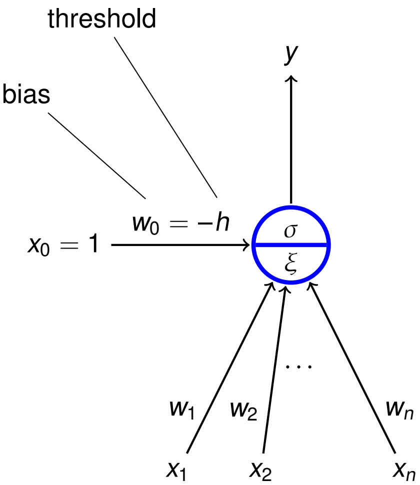
- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**

Formal neuron (with bias)



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

Formal neuron (with bias)



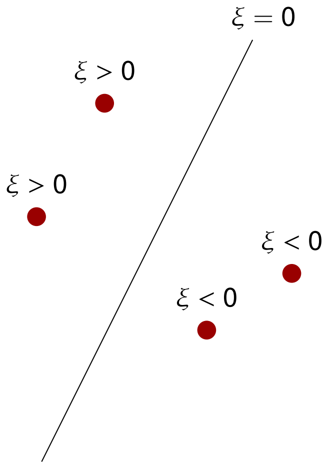
- $x_0 = 1, x_1, \dots, x_n \in \mathbb{R}$ are **inputs**
- $w_0, w_1, \dots, w_n \in \mathbb{R}$ are **weights**
- ξ is an **inner potential**; almost always $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$
- y is an **output** given by $y = \sigma(\xi)$ where σ is an **activation function**;

  e.g. a *unit step function*

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

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

Neuron and linear separation



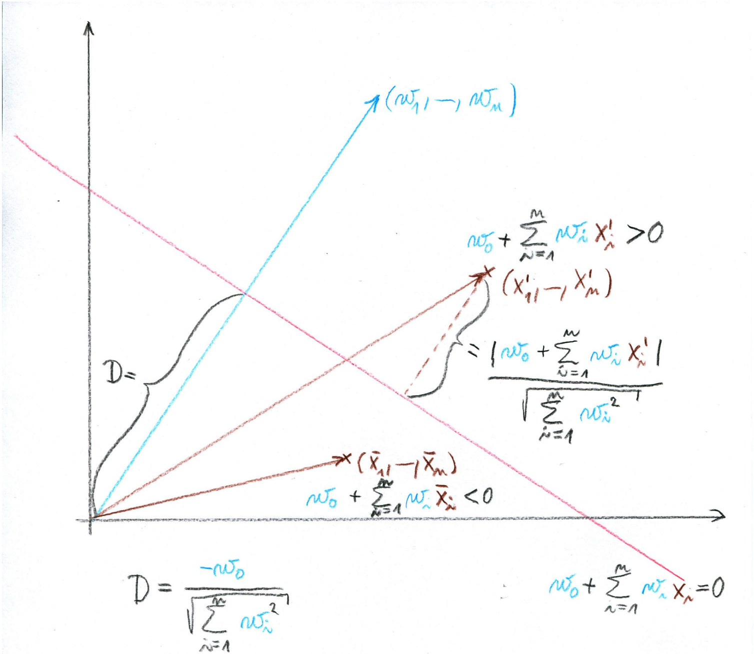
- inner potential

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

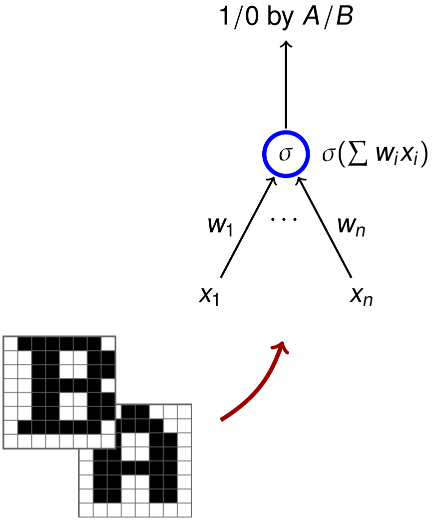
determines a separation hyperplane in the *n*-dimensional **input space**

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

Neuron geometry

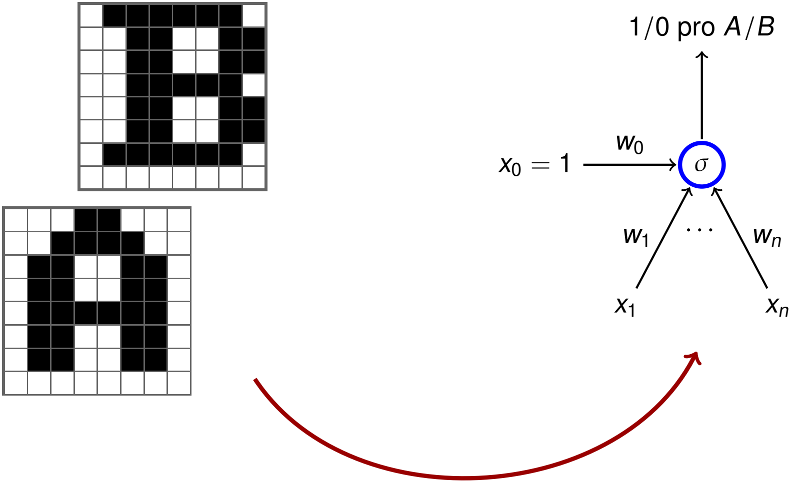


Neuron and linear separation



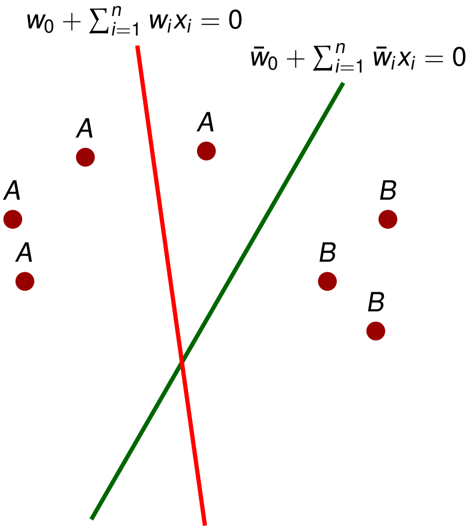
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

Neuron and linear separation



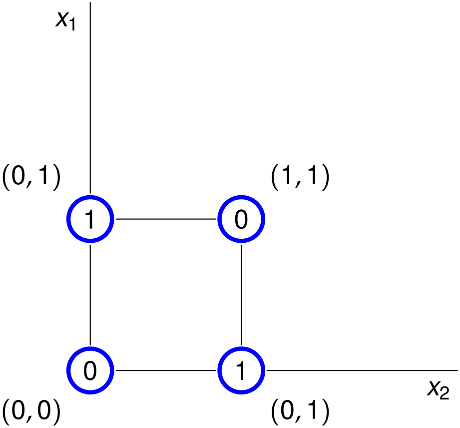
$n = 8 \cdot 8$, i.e. the number of pixels in the images. Inputs are binary vectors of dimension n (black pixel ≈ 1, white pixel ≈ 0).

Neuron and linear separation



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

Neuron and linear separation (XOR)



- No line separates ones from zeros.

Neural networks

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

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

- Architecture

  How the neurons are connected.
- Activity

  How the network transforms inputs to outputs.
- Learning

  How the weights are changed during training.

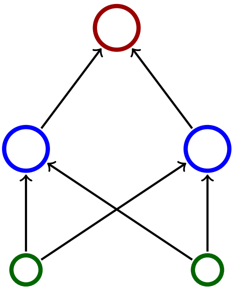
Architecture

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

We distinguish several categories of neurons:

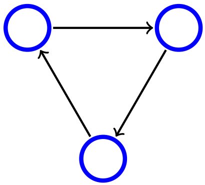
- Output neurons
- Hidden neurons
- Input neurons

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



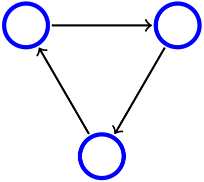
Architecture – Cycles

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

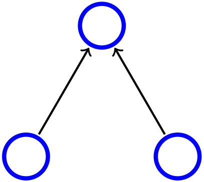


Architecture – Cycles

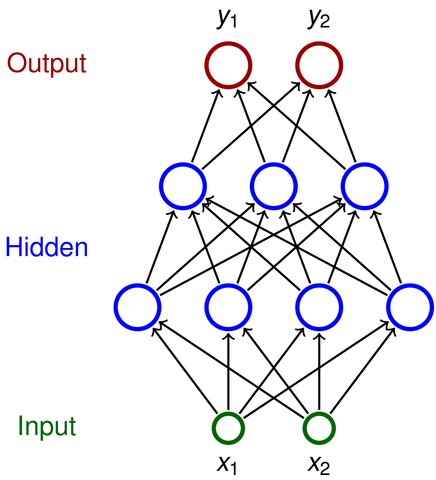
- A network is **cyclic** (recurrent) if its architecture contains a directed cycle.



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

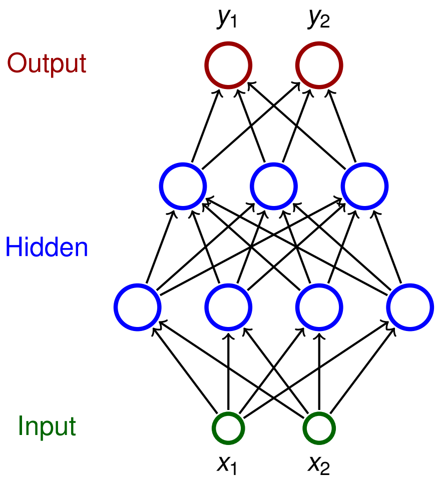


Architecture – Multilayer Perceptron (MLP)



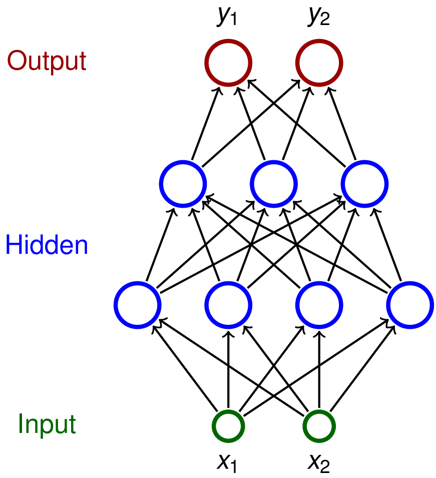
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers

Architecture – Multilayer Perceptron (MLP)



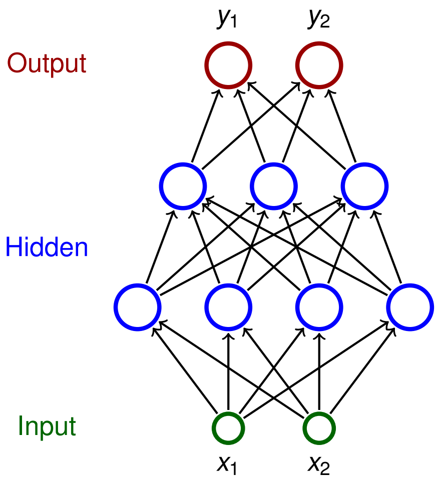
- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer

Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the *i*-th layer are connected with all neurons in the $i + 1$-st layer

Architecture – Multilayer Perceptron (MLP)



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

Activity

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

Activity

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

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

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

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

Activity

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

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

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

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

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

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

Activity

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

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

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

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

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

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

- **Initial state**

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

  Values of the remaining neurons set to 0.

Activity – computation of a network

- **Computation** (typically) proceeds in discrete steps.

Activity – computation of a network

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

Activity – computation of a network

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

Activity – computation of a network

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

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

Activity – computation of a network

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

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

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

Activity – computation of a network

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

  A computation is **finite** on a network input $\vec{x}$ if the state changes only finitely many times (i.e. there is a moment in time after which the state of the network never changes).
  We also say that the network **stops on** $\vec{x}$.
- **Network output** is a vector of values of all output neurons in the network (i.e., an element of $\mathbb{R}^{\ell}$).
  Note that the network output keeps changing throughout the computation!

MLP uses the following selection rule:

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

Activity – semantics of a network

Definition

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

Activity – semantics of a network

Definition

Consider a network with n neurons, k input, ℓ output.
Let $A \subseteq \mathbb{R}^k$ and $B \subseteq \mathbb{R}^\ell$. Suppose that the network stops on every input of A.
Then we say that the network computes a function $F : A \rightarrow B$ *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.*

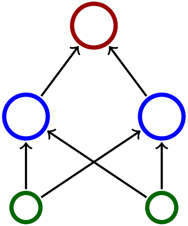
Activity – semantics of a network

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.*

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.

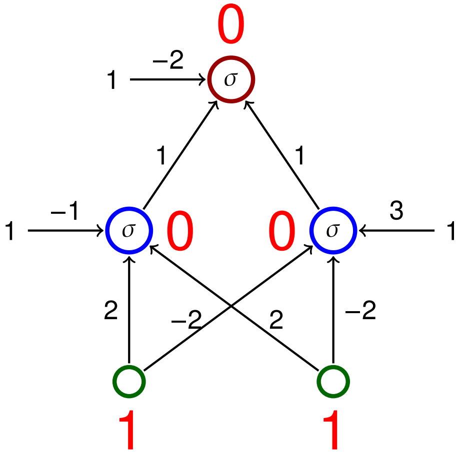
- Hyperbolic tangens

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

- ReLU

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

Activity – XOR



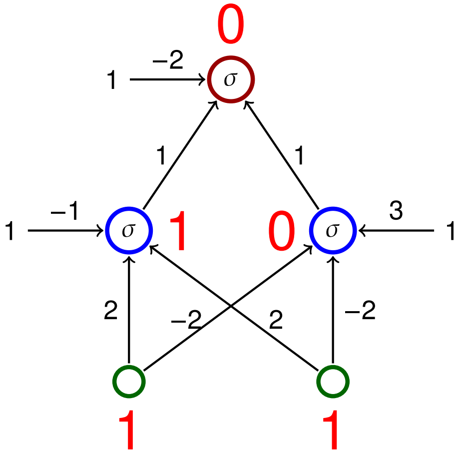
- Activation function is a unit step function

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

- 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



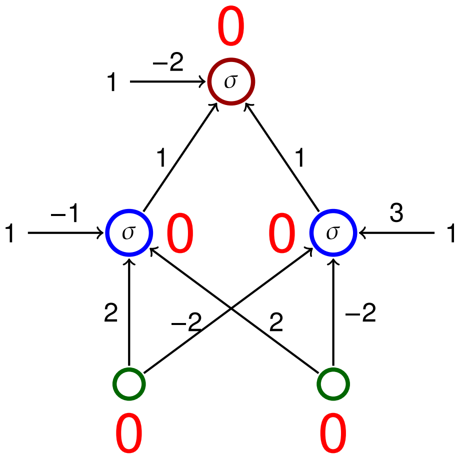
- Activation function is a unit step function

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

- 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



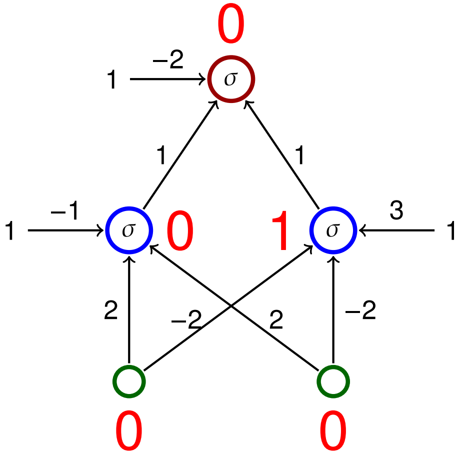
- Activation function is a unit step function

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

- 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



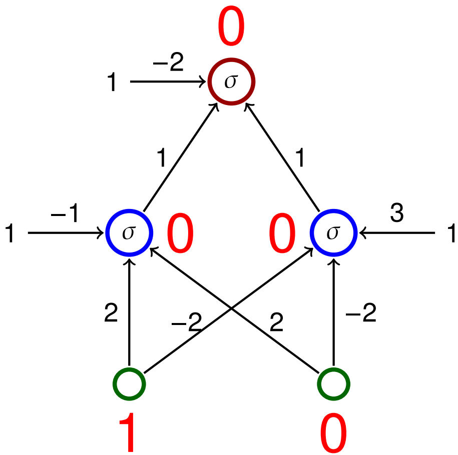
- Activation function is a unit step function

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

- 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



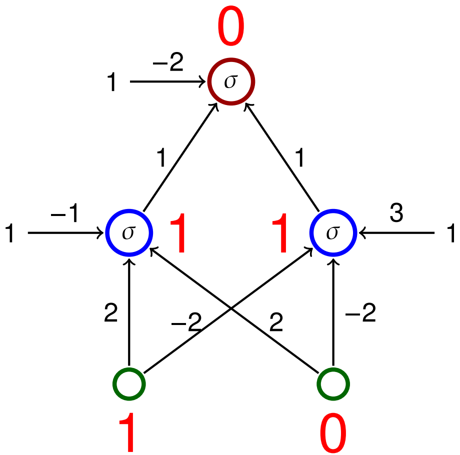
- Activation function is a unit step function

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

- 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



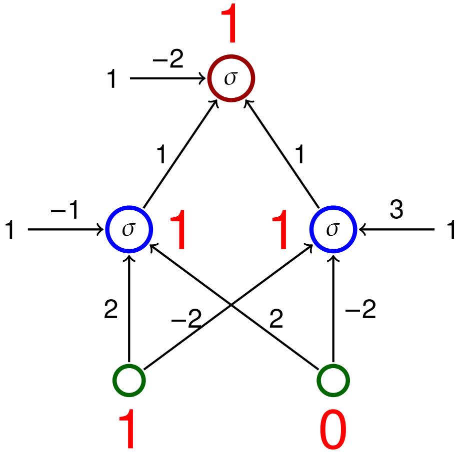
- Activation function is a unit step function

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

- 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



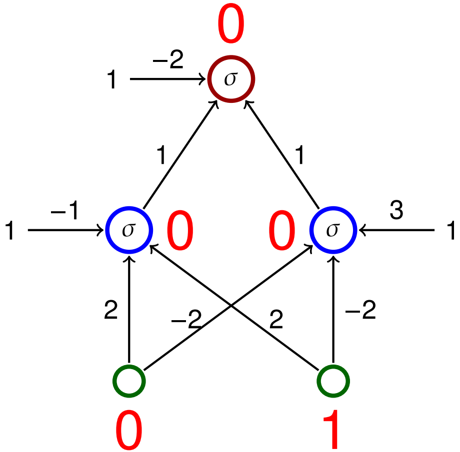
- Activation function is a unit step function

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

- 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



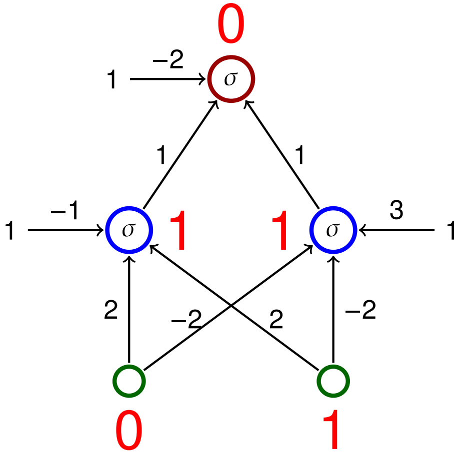
- Activation function is a unit step function

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

- 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



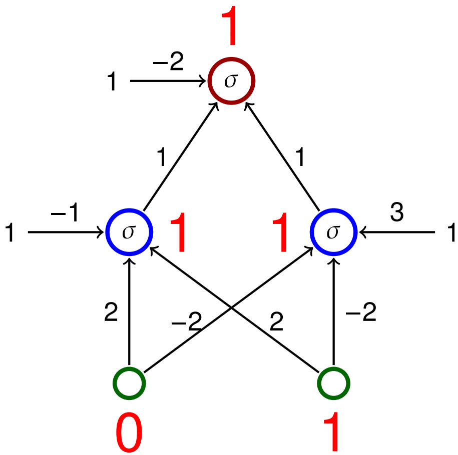
- Activation function is a unit step function

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

- 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



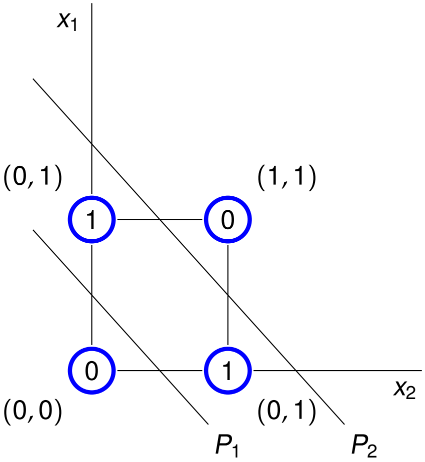
- Activation function is a unit step function

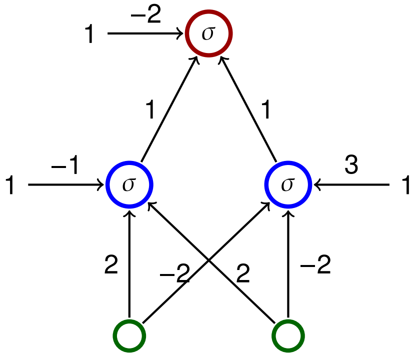
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

- 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

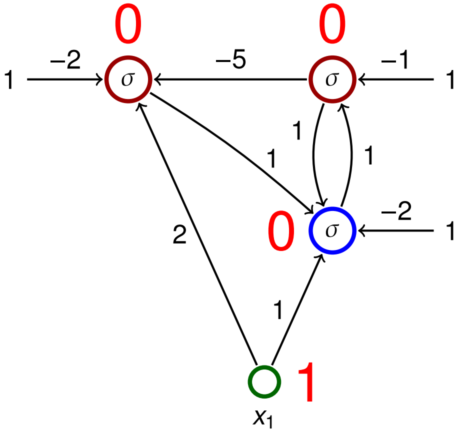
Activity – MLP and linear separation





- 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

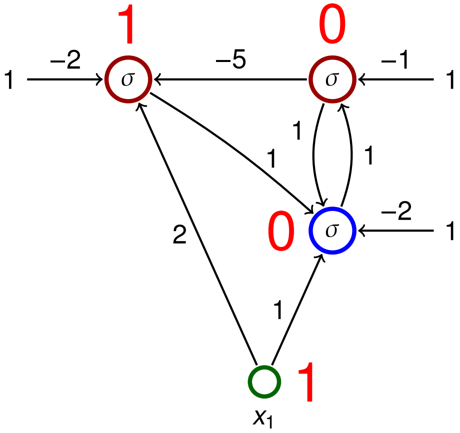


The activation function is the unit step function

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

The input is equal to 1

Activity – example

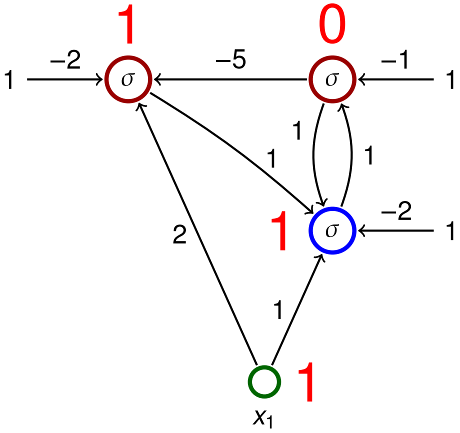


The activation function is the unit step function

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

The input is equal to 1

Activity – example

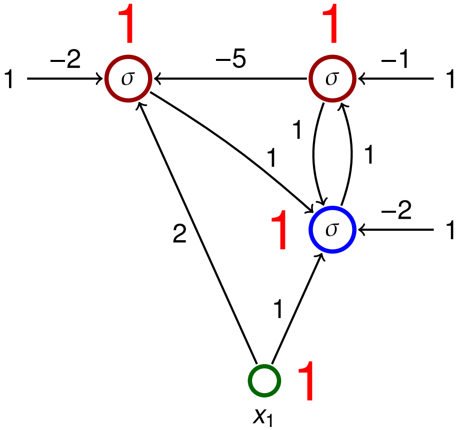


The activation function is the unit step function

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

The input is equal to 1

Activity – example

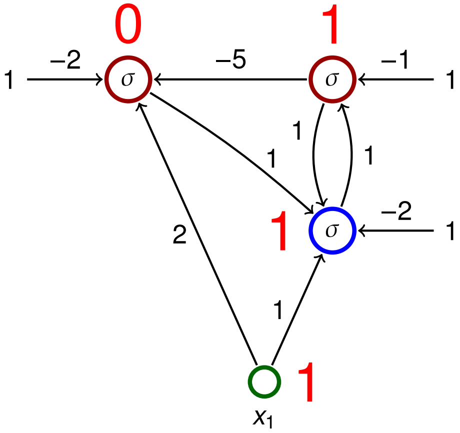


The activation function is the unit step function

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

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)

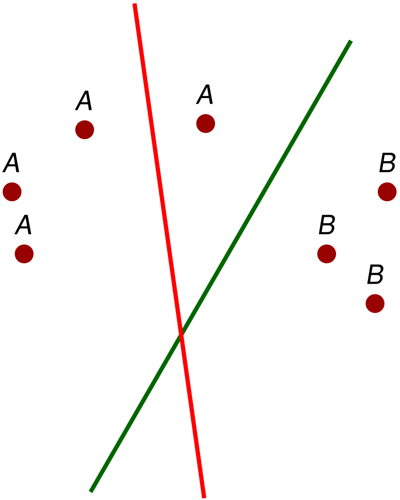
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.

Learning algorithms

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

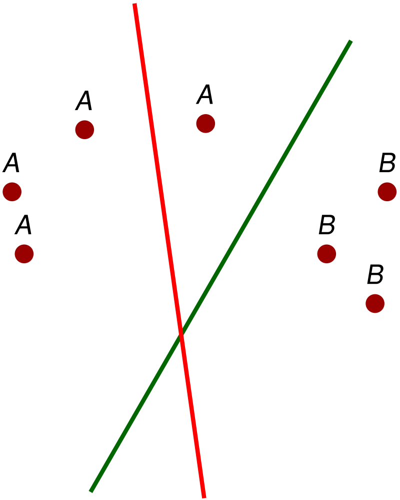
- Supervised learning
  - The desired function is described using *training examples* that are pairs of the form (input, output).
  - Learning algorithm searches for a configuration which "corresponds" to the training examples, typically by minimizing an error function.
- Unsupervised learning
  - The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

Supervised learning – illustration



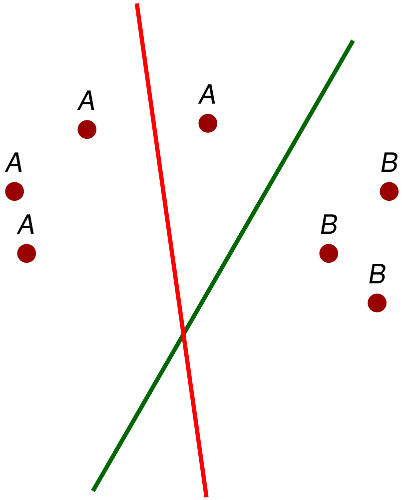
- classification in the plane using a single neuron

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

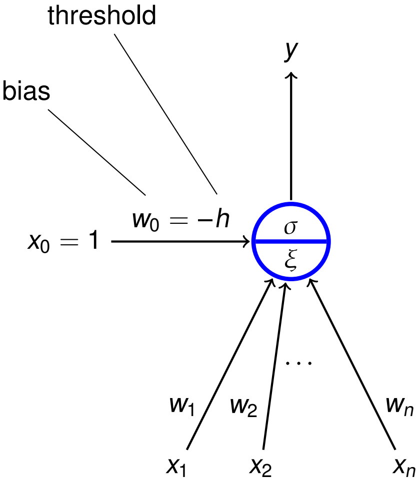
- Massive parallelism
  - neurons can be evaluated in parallel
- Learning
  - many sophisticated learning algorithms used to "program" neural networks
- generalization and robustness
  - information is encoded in a distributed manner in weights
  - "close" inputs typicaly get similar values

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 manner 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*

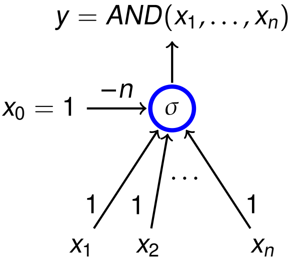
$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

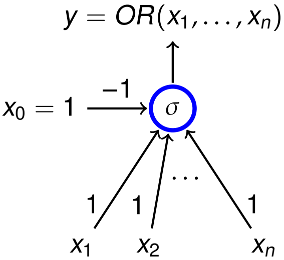
Boolean functions

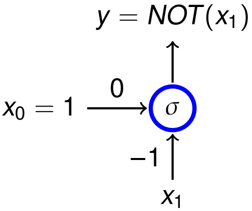
Activation function: *unit step function* $\sigma(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$

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

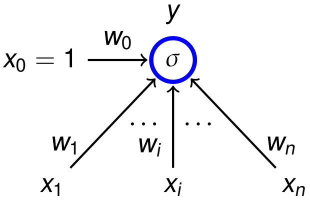
Boolean functions

Theorem

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

Proof.

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

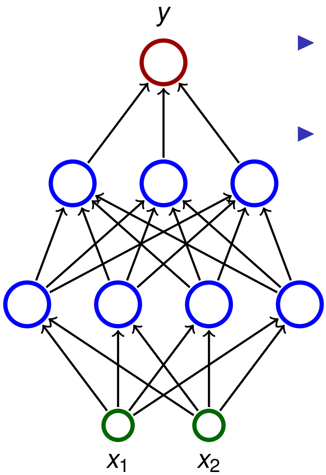


$$w_0 = -\sum_{i=1}^{n} v_i$$

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

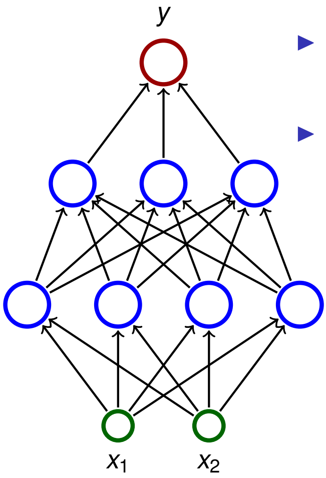
- Now let us connect all outputs of all neurons $N_{\vec{v}}$ satisfying $F(\vec{v}) = 1$ using a neuron implementing OR. □

Non-linear separation



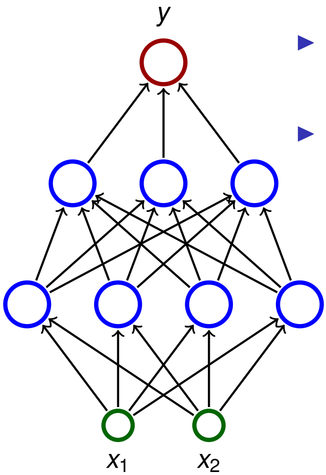
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).

Non-linear separation



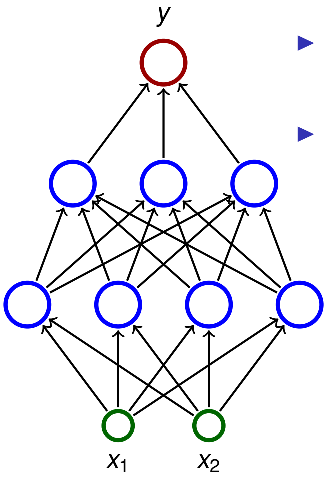
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.

Non-linear separation



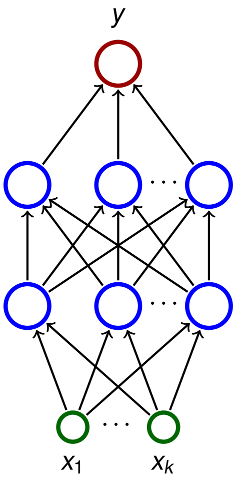
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation



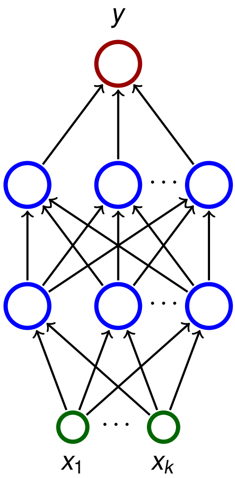
- Consider a three layer network; each neuron has the unit step activation function.
- The network divides the input space in two subspaces according to the output (0 or 1).
  - The first (hidden) layer divides the input space into half-spaces.
  - The second layer may e.g. make intersections of the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

Non-linear separation – illustration



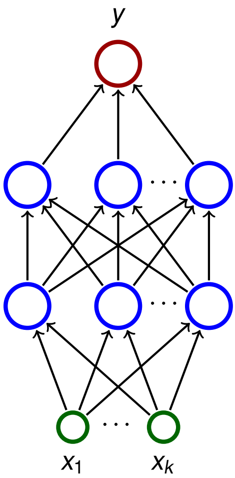
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.

Non-linear separation – illustration



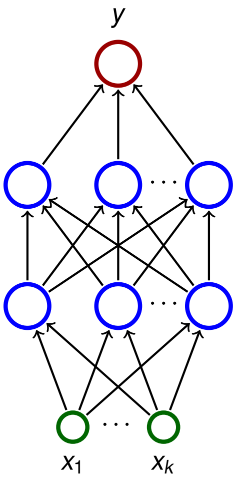
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)

Non-linear separation – illustration



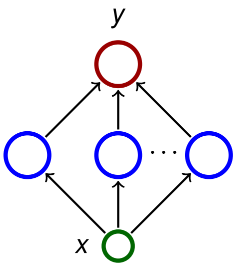
- Consider three layer networks; each neuron has the unit step activation function.
- Three layer nets are capable of "approximating" any "reasonable" subset A of the input space $\mathbb{R}^k$.
  - Cover A with hypercubes (in 2D squares, in 3D cubes, ...)
  - Each hypercube K can be separated using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

Non-linear separation – illustration



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

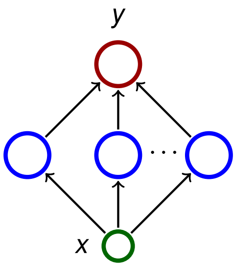
Power of ReLU



Consider a two layer network

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

Power of ReLU

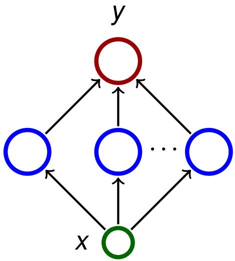


Consider a two layer network

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

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

Power of ReLU



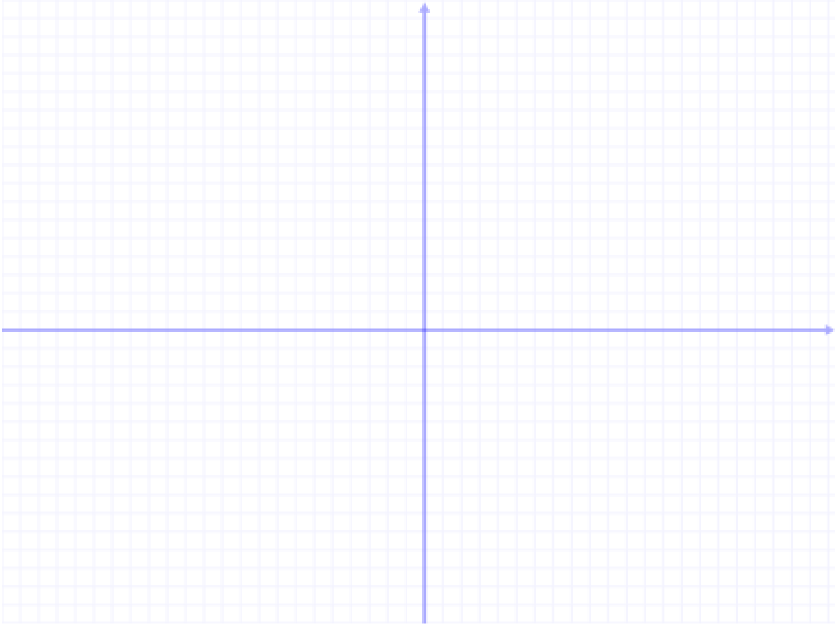
Consider a two layer network

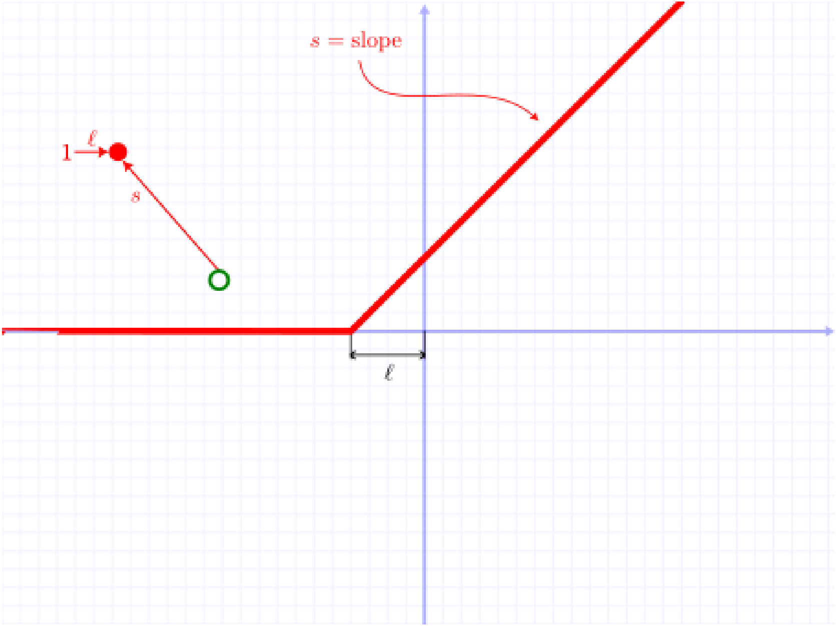
- with a single input and single output;
- hidden neurons with the ReLU activation: $\sigma(\xi) = \max(\xi, 0)$;
- the output neuron with identity activation: $\sigma(\xi) = \xi$ (linear model)

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

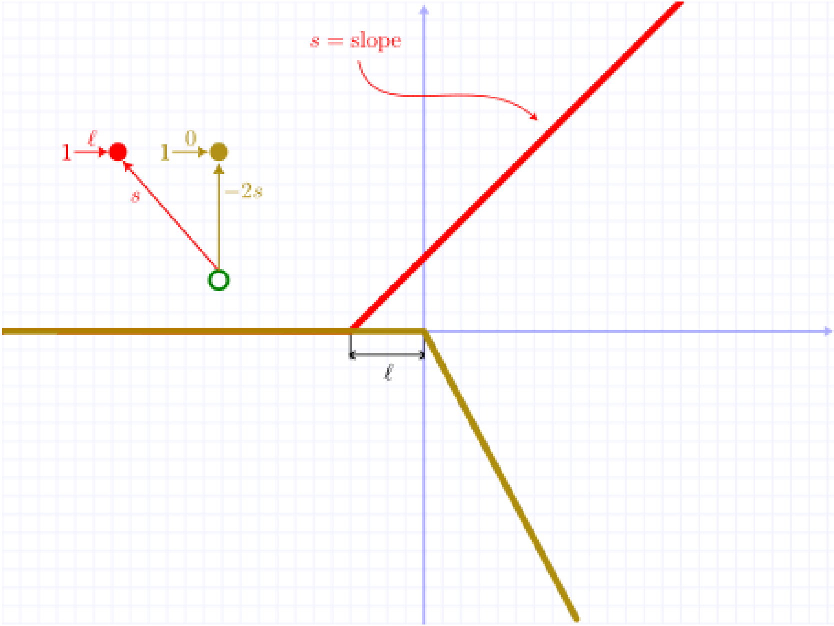
For every open subset $A \subseteq [0, 1]$ there is a network of the above type such that for "most" $x \in [0, 1]$ we have that $x \in A$ iff the network's output is > 0 for the input x.

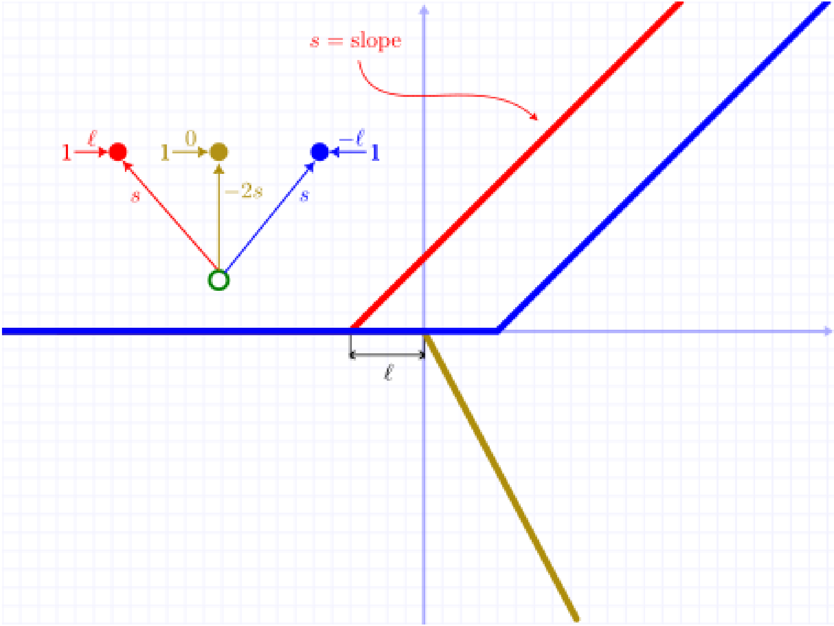
Just consider a continuous function f where $f(x)$ is the minimum difference between x and a point on the boundary of A. Then uniformly approximate f using the networks.

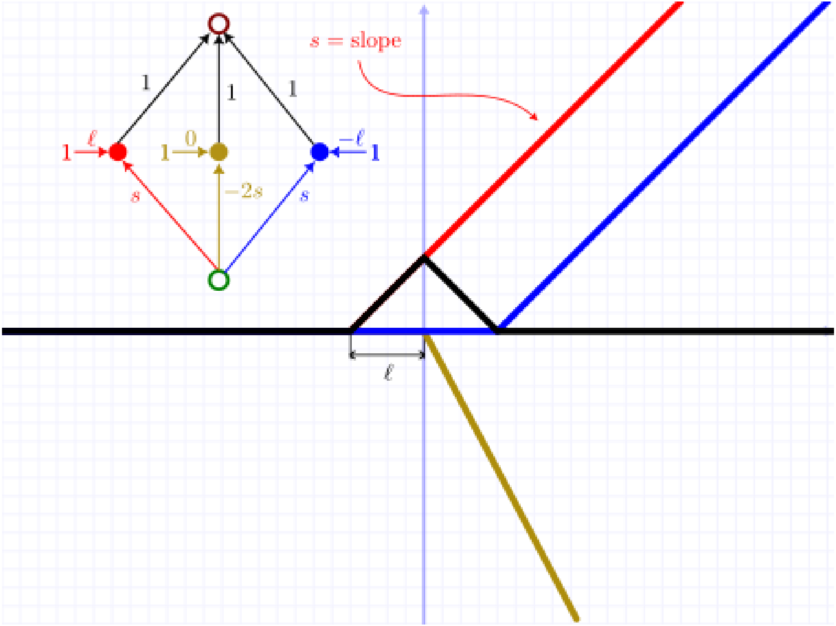


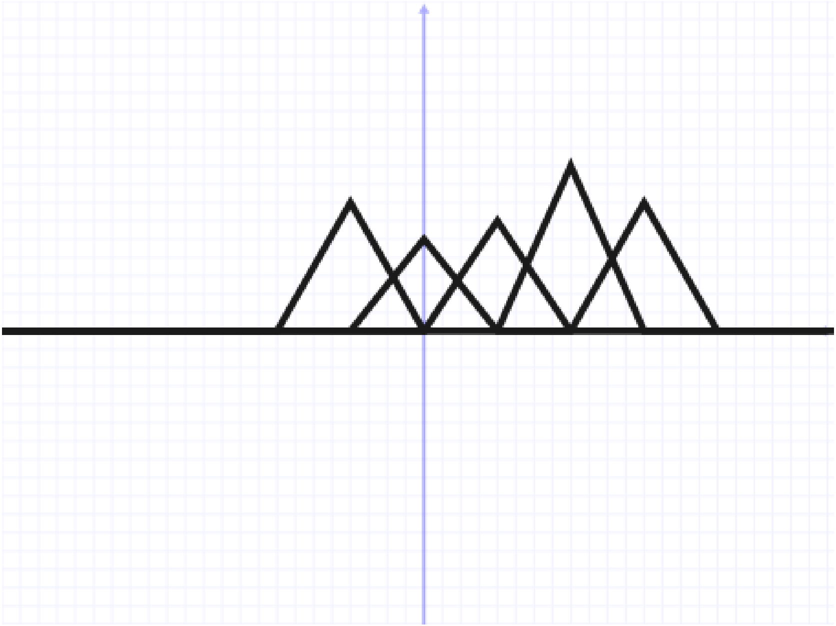


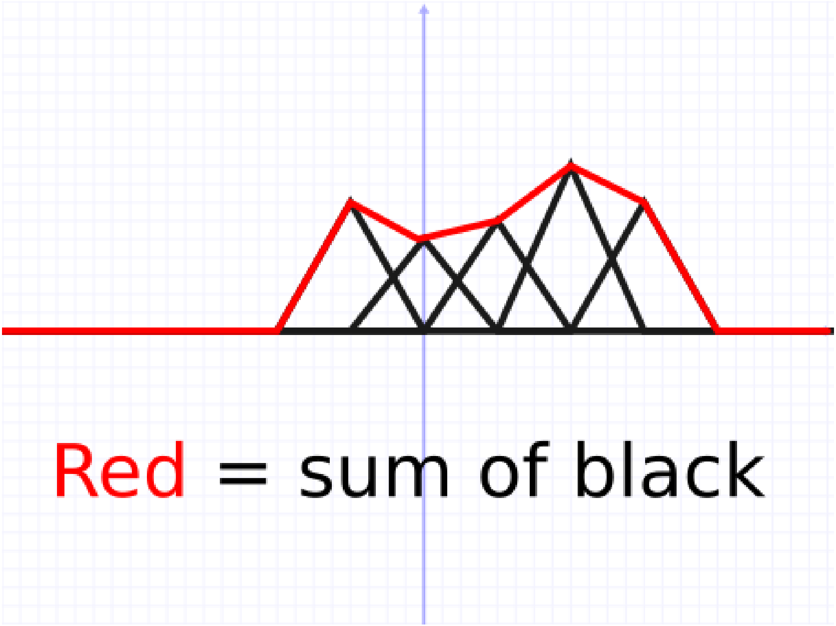
s = slope
1
ℓ
s
ℓ


s = slope
1
ℓ
1
0
s
−2s
ℓ


s = slope
1
ℓ
0
1
−ℓ
1
s
−2s
s
ℓ


s = slope
1
1
1
1
ℓ
1
0
−ℓ
1
s
−2s
s
ℓ




Red = sum of black

Non-linear separation - sigmoid

Theorem (Cybenko 1989 - informal version)

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

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

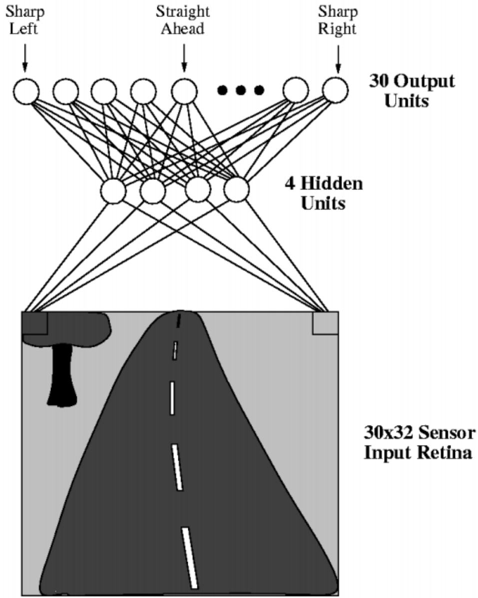
For every "reasonable" set $A \subseteq [0,1]^n$, there is a **two layer network** *where each hidden neuron has the activation function σ (output neurons are linear), that satisfies the following:*

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:

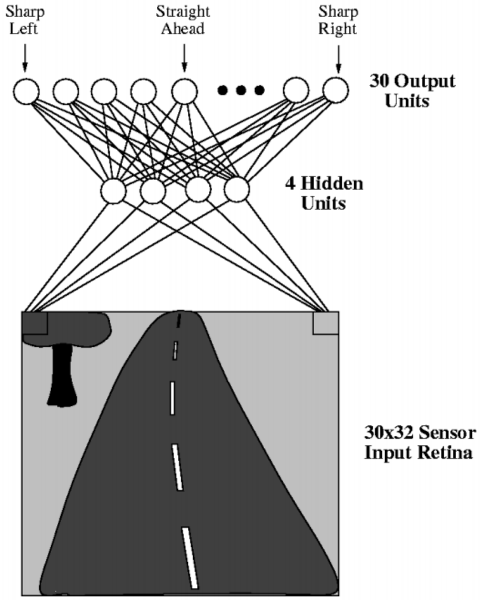
- "reasonable" means Lebesgue measurable
- "most" means that the set of incorrectly classified vectors has the Lebesgue measure smaller than a given $\varepsilon > 0$

Non-linear separation - practical illustration



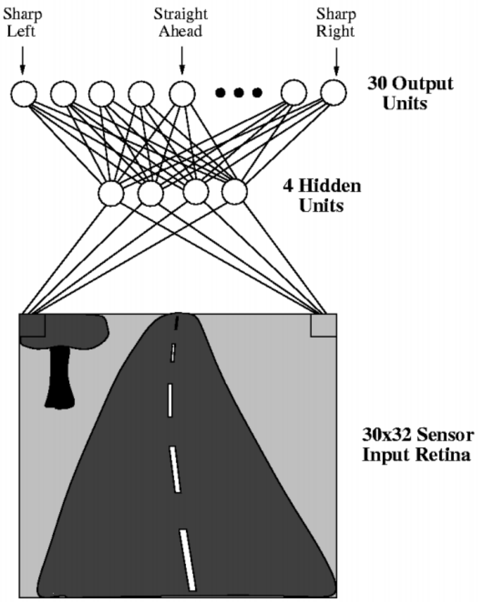
- ALVINN drives a car

Non-linear separation - practical illustration



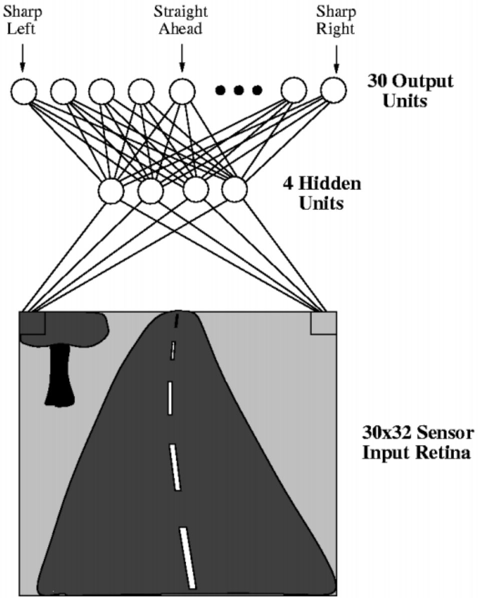
- ALVINN drives a car
- The net has $30 \times 32 = 960$ inputs (the input space is thus $\mathbb{R}^{960}$)

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".

Image source: `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

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

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*

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

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 $F : A \rightarrow \mathbb{R}$ 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);

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 advantages of neural networks are: learning, generalization, robustness.

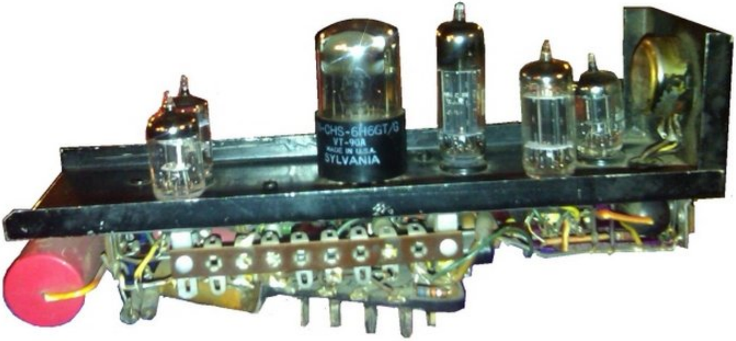
Neural networks vs classical computers

	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.)



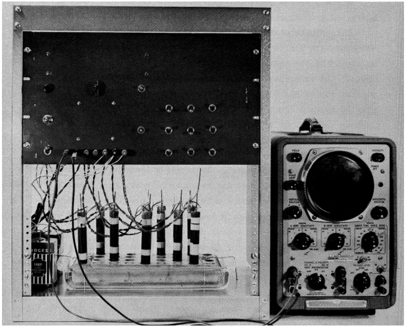
History of neurocomputers

- 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-4 generates pretty realistic articles, short plays (for a theatre) have been successfully generated, etc.

Example

This slide was automatically generated byaskig GPT-4 "Give me a beamer slide with complexity of Steepest descent, Neton’s method and BFGS".

Computational Complexity

Algorithm	Computational Complexity
Steepest Descent	$O(n)$ per iteration
Newton's Method	$O(n^3)$ to compute Hessian and solve system
BFGS	$O(n^2)$ to update Hessian approximation

Table: Summary of the computational complexity for each optimization algorithm.

- Steepest Descent: Simple but often slow, requiring many iterations.
- Newton's Method: Fast convergence but expensive per iteration.
- BFGS: Quasi-Newton, no Hessian needed, good speed and iteration count balance.

Example Source

```
\begin{frame}{Computational Complexity}
\begin{table}
\begin{tabular}{l c}
\hline
\textbf{Algorithm} & \textbf{Computational Complexity} \\
\hline
Steepest Descent & $O(n)$ per iteration \\
Newton's Method & $O(n^3)$ to compute Hessian and solve system \\
BFGS & $O(n^2)$ to update Hessian approximation \\
\hline
\end{tabular}
\caption{Summary of the computational complexity for each optimization algorithm.}
\end{table}

\begin{itemize}
    \item Steepest Descent: Simple but often slow, requiring many iterations.
    \item Newton's Method: Fast convergence but expensive per iteration.
    \item BFGS: Quasi-Newton, no Hessian needed, good speed and iteration count balance.
\end{itemize}
\end{frame}
```

History in waves ...

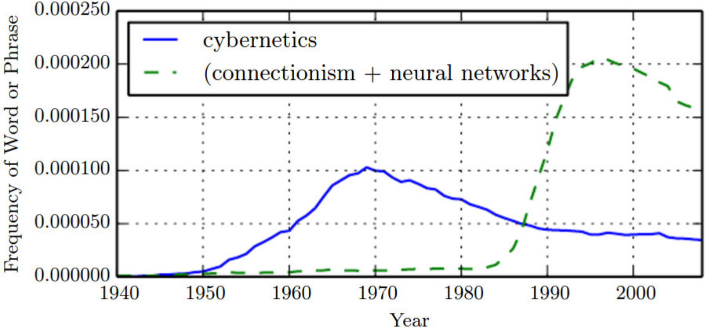
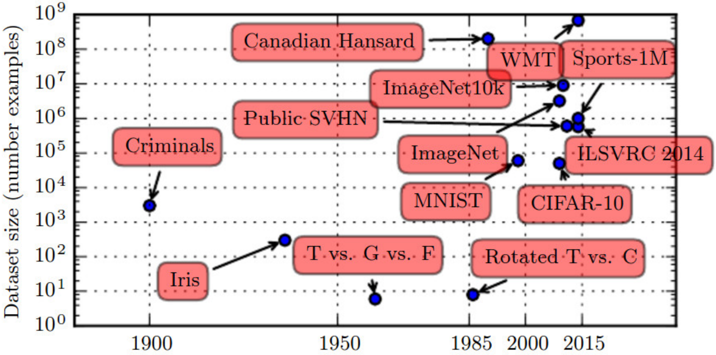


Figure: The figure shows two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks" according to Google Books (the third wave is too recent to appear).

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

GPT-3 Training Dataset

45 TB text data from multiple sources

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Common Crawl corpus contains petabytes of data collected over 8 years of web crawling. The corpus contains raw web page data, metadata extracts and text extracts with light filtering.

WebText2 is the text of web pages from all outbound Reddit links from posts with 3+ upvotes.

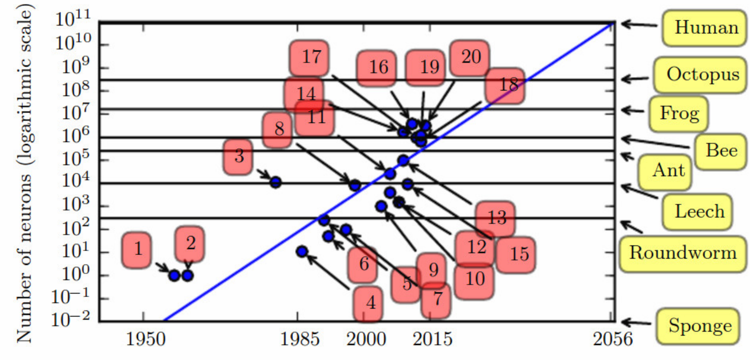
Books1 & Books2 are two internet-based books corpora.

Wikipedia pages in the English language are also part of the training corpus.

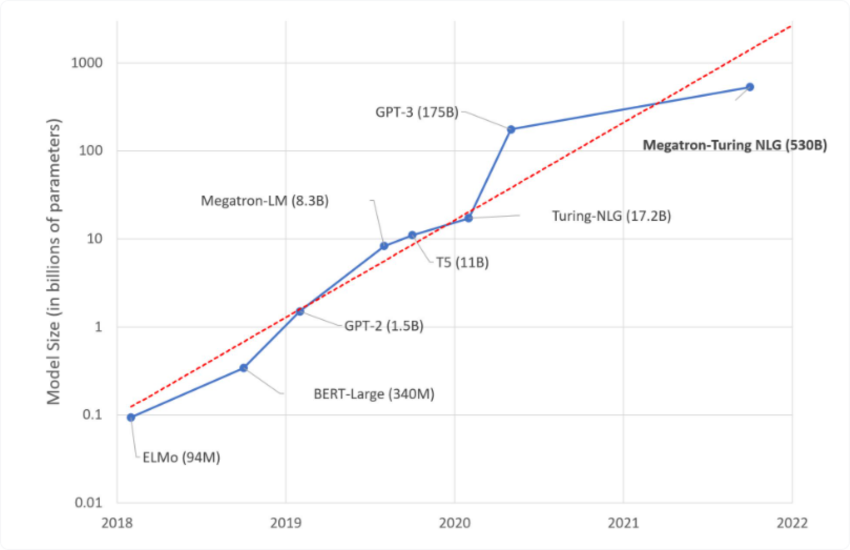
Source: Kindra Cooper. OpenAI GPT-3: Everything You Need to Know. Springboard. 2023

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)



GPT-4's Scale: GPT-4 has 1.8 trillion parameters across 120 layers, which is over 10 times larger than GPT-3.

Current hardware – What do we face?

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

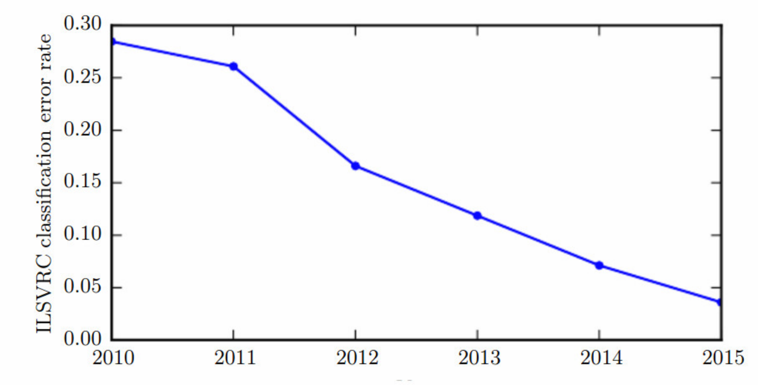


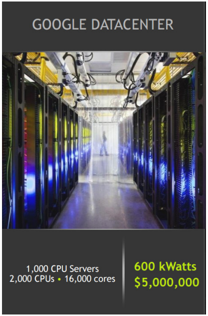
Figure: Since deep networks reached the scale necessary to compete in the ImageNetLarge Scale Visual Recognition Challenge, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky et al. (2014b) and He et al. (2015).

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

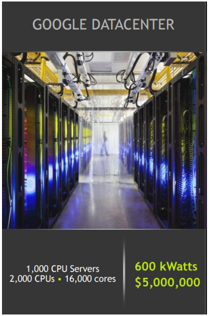
In 2012, Google trained a large network of 1.7 billion weights and 9 layers

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.

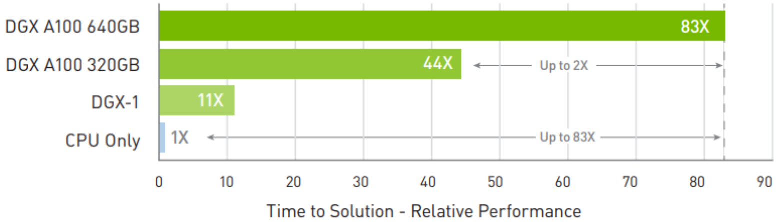
Able to train 1 billion parameter networks on just 3 machines in a couple of days.
Able to scale to 11 billion weights (approx. 6.5 times larger than the Google model) on 16 GPUs.



Current hardware – NVIDIA DGX Station

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

Up to 83X Higher Throughput than CPU, 2X Higher Throughput than DGX A100 320GB on Big Data Analytics Benchmark



Deep learning in clouds

Big companies offer cloud services for deep learning:

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

Advantages:

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

Disadvatages:

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

Current software

- **TensorFlow** (Google)
  - open source software library for numerical computation using data flow graphs
  - allows implementation of most current neural networks
  - allows computation on multiple devices (CPUs, GPUs, ...)
  - Python API
  - **Keras**: a part of TensorFlow that allows easy description of most modern neural networks
- **PyTorch** (Facebook)
  - similar to TensorFlow
  - object oriented
  - ... majority of new models in research papers implemented in PyTorch

    https://www.cioinsight.com/big-data/pytorch-vs-tensorflow/
- **Theano (dead)**:
  - The "academic" grand-daddy of deep-learning frameworks, written in Python. Strongly inspired TensorFlow (some people developing Theano moved on to develop TensorFlow).
- There are others: Caffe, Deeplearning4j, ...

Current software – Keras

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

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

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

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

Current software – Keras functional API

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

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

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

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

Current software – TensorFlow

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

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

Current software – TensorFlow

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

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

Current software – PyTorch

```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

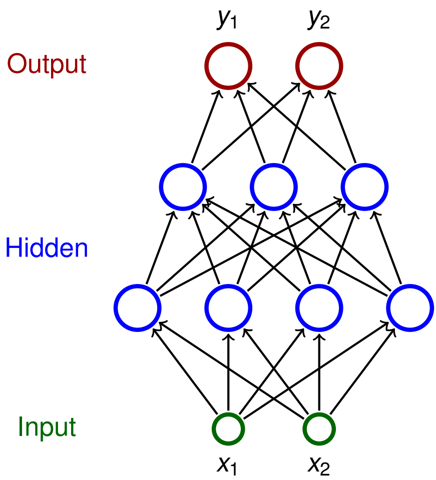
Most "mathematical" software packages contain some support of neural networks:

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

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

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., a 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 the 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

- inner potential of neuron j:

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

MLP – activity

- inner potential of neuron j:

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

- activation function σ_j for neuron j (arbitrary differentiable)

MLP – activity

- inner potential of neuron j:

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

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

- 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

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

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

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

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

- **Error function:**

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

where

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

This is just an example of an error function; we shall see other error functions later.

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 $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-

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

Derivation of backprop.

Consider $k = 1, \ldots, p$ and a weight w_{ji}. By the chain rule:

$$\frac{\partial E_k}{\partial w_{ji}} =$$

Derivation of backprop.

Consider $k = 1, \ldots, p$ and a weight w_{ji}. By the chain rule:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ji}} =$$

Derivation of backprop.

Consider $k = 1, \ldots, p$ and a weight w_{ji}. By the chain rule:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot \frac{\partial \xi_j}{\partial w_{ji}} =$$

Derivation of backprop.

Consider $k = 1, \ldots, p$ and a weight w_{ji}. By the chain rule:

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

since

$$\frac{\partial y_j}{\partial \xi_j} = \frac{\partial(\sigma_j(\xi_j))}{\partial \xi_j} = \sigma_j'(\xi_j)$$

$$\frac{\partial \xi_j}{\partial w_{ji}} = \frac{\partial \left(\sum_{r \in j_{\leftarrow}} w_{jr} y_r\right)}{\partial w_{ji}} = y_i$$

Derivation of backdrop. (cont.)

For $j \in Y$: $\frac{\partial E_k}{\partial y_j} = \frac{\partial \left(\frac{1}{2} \sum_{r \in Y} (y_r - d_{kr})^2 \right)}{\partial y_j} = y_j - d_{kj}$

Derivation of backdrop. (cont.)

$$\text{For } j \in Y : \frac{\partial E_k}{\partial y_j} = \frac{\partial \left(\frac{1}{2}\sum_{r \in Y}(y_r - d_{kr})^2\right)}{\partial y_j} = y_j - d_{kj}$$

... and another application of the chain rule:

$$\text{For } j \notin Y : \frac{\partial E_k}{\partial y_j} =$$

Derivation of backdrop. (cont.)

$$\text{For } j \in Y: \frac{\partial E_k}{\partial y_j} = \frac{\partial \left(\frac{1}{2}\sum_{r \in Y}(y_r - d_{kr})^2\right)}{\partial y_j} = y_j - d_{kj}$$

... and another application of the chain rule:

$$\text{For } j \notin Y: \frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \frac{\partial y_r}{\partial y_j} =$$

Derivation of backdrop. (cont.)

$$\text{For } j \in Y: \frac{\partial E_k}{\partial y_j} = \frac{\partial \left(\frac{1}{2} \sum_{r \in Y} (y_r - d_{kr})^2 \right)}{\partial y_j} = y_j - d_{kj}$$

... and another application of the chain rule:

$$\text{For } j \notin Y: \frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \frac{\partial y_r}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \frac{\partial y_r}{\partial \xi_r} \cdot \frac{\partial \xi_r}{\partial y_j}$$

$$=$$

Derivation of backdrop. (cont.)

$$\text{For } j \in Y: \frac{\partial E_k}{\partial y_j} = \frac{\partial \left(\frac{1}{2} \sum_{r \in Y} (y_r - d_{kr})^2 \right)}{\partial y_j} = y_j - d_{kj}$$

... and another application of the chain rule:

$$\text{For } j \notin Y: \frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \frac{\partial y_r}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \frac{\partial y_r}{\partial \xi_r} \cdot \frac{\partial \xi_r}{\partial y_j}$$

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

since

$$\frac{\partial y_r}{\partial \xi_r} = \frac{\partial(\sigma_r(\xi_r))}{\partial \xi_r} = \sigma_r'(\xi_r)$$

$$\frac{\partial \xi_r}{\partial y_j} = \frac{\partial \left(\sum_{s \in r_{\leftarrow}} w_{rs} y_s \right)}{\partial y_j} = w_{rj}$$

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

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

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

MLP – computing the gradient

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

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

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

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

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

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

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

MLP – backpropagation

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

MLP – backpropagation

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

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

MLP – backpropagation

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

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

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

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

Complexity of the batch algorithm

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

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following *p* times:

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

Complexity of the batch algorithm

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

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

Proof sketch: The algorithm does the following p times:

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

The steps 1. - 3. take linear time w.r.t. the number of network weights.

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)

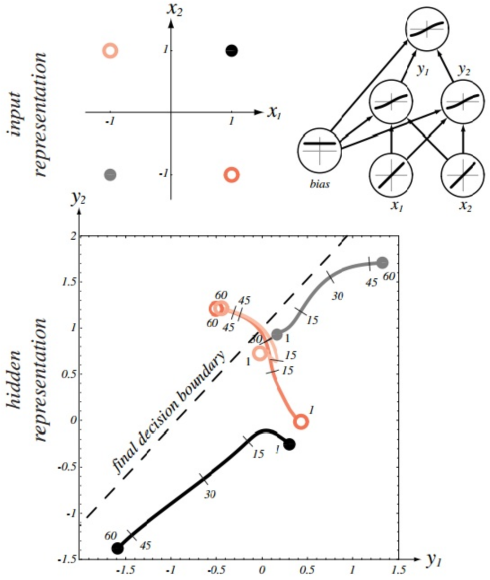
Proof sketch: The algorithm does the following p times:

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 w.r.t. the number of network weights.

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 the 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.

Regression: Output and Error

- For **regression**, the output activation is typically the identity, i.e., $y_i = \sigma(\xi_i) = \xi_i$ for $i \in Y$.

Regression: Output and Error

- For **regression**, the output activation is typically the identity, i.e., $y_i = \sigma(\xi_i) = \xi_i$ for $i \in Y$.
- 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 $(d_{ki})_{i \in Y}$).

Regression: Output and Error

- For **regression**, the output activation is typically the identity, i.e., $y_i = \sigma(\xi_i) = \xi_i$ for $i \in Y$.
- A training dataset

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

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

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

Maximum Likelihood vs Least Squares

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$, $d_k \in \mathbb{R}$.
Consider a single output neuron o.

Maximum Likelihood vs Least Squares

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$, $d_k \in \mathbb{R}$.
Consider a single output neuron o.

Assume that each d_k was generated randomly as follows

$$d_k = y_o(\vec{w}, \vec{x}_k) + \epsilon_k$$

- $\vec{w}$ are **unknown constants**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2

Maximum Likelihood vs Least Squares

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$, $d_k \in \mathbb{R}$.
Consider a single output neuron o.

Assume that each d_k was generated randomly as follows

$$d_k = y_o(\vec{w}, \vec{x}_k) + \epsilon_k$$

- $\vec{w}$ are **unknown constants**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2

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

Maximum Likelihood vs Least Squares

Fix a training set $D = \left\{(x_1, d_1), (x_2, d_2), \ldots, \left(x_p, d_p\right)\right\}$, $d_k \in \mathbb{R}$.
Consider a single output neuron o.

Assume that each d_k was generated randomly as follows

$$d_k = y_o(\vec{w}, \vec{x}_k) + \epsilon_k$$

- $\vec{w}$ are **unknown constants**
- ϵ_k are normally distributed with mean 0 and an unknown variance σ^2

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

Denote by $p(d_1, \ldots, d_p \mid \vec{w}, \sigma^2)$ the probability density of the values $d_1, \ldots, d_n$ assuming fixed $x_1, \ldots, x_p, \vec{w}, \sigma^2$.

(For the interested: The independence and definition of d_k's imply

$$p(d_1, \ldots, d_p \mid \vec{w}, \sigma^2) = \prod_{k=1}^{p} N[y_o(\vec{w}, \vec{x}_k), \sigma^2](d_k)$$

$N[y_o(\vec{w}, \vec{x}_k), \sigma^2](d_k)$ is a normal dist. with the mean $y_o(\vec{w}, \vec{x}_k)$ and var. σ^2.)

Maximum Likelihood vs Least Squares

Our goal is to find the weights $\vec{w}$ that maximize the likelihood

$$L(\vec{w}, \sigma^2) := p(d_1, \ldots, d_p \mid \vec{w}, \sigma^2)$$

But now with the fixed values $d_1, \ldots, d_n$ from the training set!

Maximum Likelihood vs Least Squares

Our goal is to find the weights $\vec{w}$ that maximize the likelihood

$$L(\vec{w}, \sigma^2) := p(d_1, \ldots, d_p \mid \vec{w}, \sigma^2)$$

But now with the fixed values $d_1, \ldots, d_n$ from the training set!

Theorem
The unique $\vec{w}$ that minimize the least squares error $E[\vec{w}]$ maximize $L(\vec{w}, \sigma^2)$ *for an arbitrary variance σ^2.*

Classification: Output and Error

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_{j_1},\ldots,\xi_{j_k}) = \frac{e^{\xi_i}}{\sum_{j\in Y} e^{\xi_j}} \qquad \text{Here } Y = \{j_1,\ldots,j_k\}$$

Classification: Output and Error

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_{j_1}, \ldots, \xi_{j_k}) = \frac{e^{\xi_i}}{\sum_{j \in Y} e^{\xi_j}} \qquad \text{Here } Y = \{j_1, \ldots, j_k\}$$

- 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 $(d_{ki})_{i \in Y}$).

Classification: Output and Error

- The output activation function *softmax*:

$$y_i = \sigma_i(\xi_{j_1}, \ldots, \xi_{j_k}) = \frac{e^{\xi_i}}{\sum_{j \in Y} e^{\xi_j}} \qquad \text{Here } Y = \{j_1, \ldots, j_k\}$$

- 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 $\ell \in V$.

Binary Classification: Output and Error

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

- The output activation function *logistic sigmoid*:

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

Binary Classification: Output and Error

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.

Binary Classification: Output and Error

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:*

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

Cross-entropy vs max likelihood

Consider our model giving a probability $y_o(\vec{w}, \vec{x})$ given input $\vec{x}$.

Cross-entropy vs max likelihood

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

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

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

Cross-entropy vs max likelihood

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

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

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

The *likelihood*:

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

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

Minimizing the cross-entropy 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

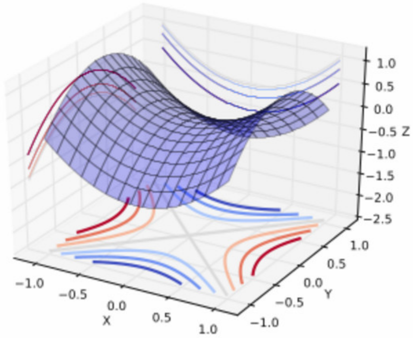
- Small networks: Lots of local minima where the descent gets stuck.
- The model identifiability problem: Swapping incoming weights of neurons *i* and *j* leaves the same network topology – **weight space symmetry**.
- Recent studies show that for sufficiently large networks, all local minima have low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

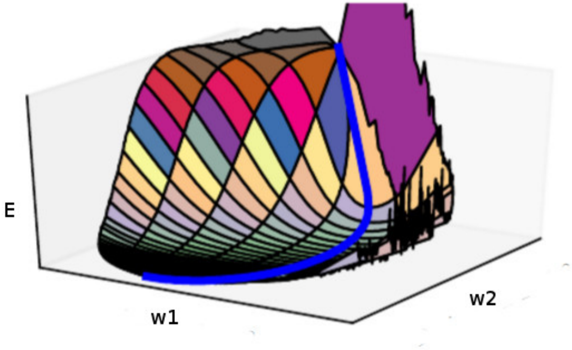
Saddle points

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



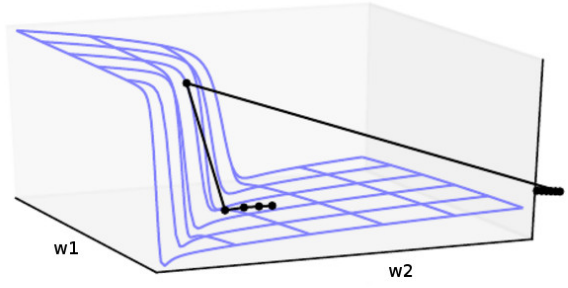
Issues in gradient descent – too slow descent

- flat regions

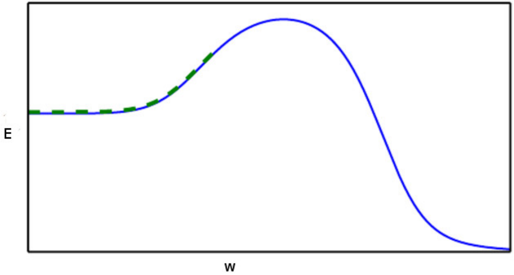


Issues in gradient descent – too fast descent

- 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 such that all $|d_k|$ are bounded by a constant,*
- *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 a linear rate.

Later, we get to a special type of network 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

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

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

- 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. The typical power of 2 batch sizes ranges 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. The typical power of 2 batch sizes ranges 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.

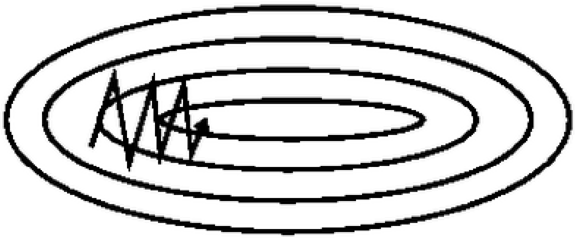
  It has been observed in practice that when using a larger batch, there is a degradation in the quality of the model, as measured by its ability to generalize.

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

Momentum

The issue in the gradient descent:

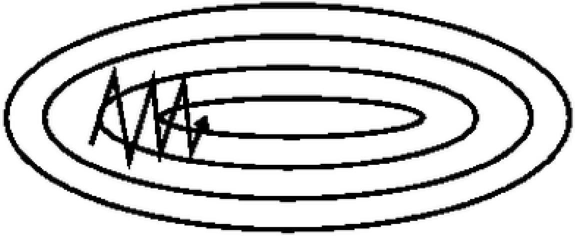
- $\nabla E(\vec{w}^{(t)})$ constantly changes direction (but the error steadily decreases).



Momentum

The issue in the gradient descent:

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

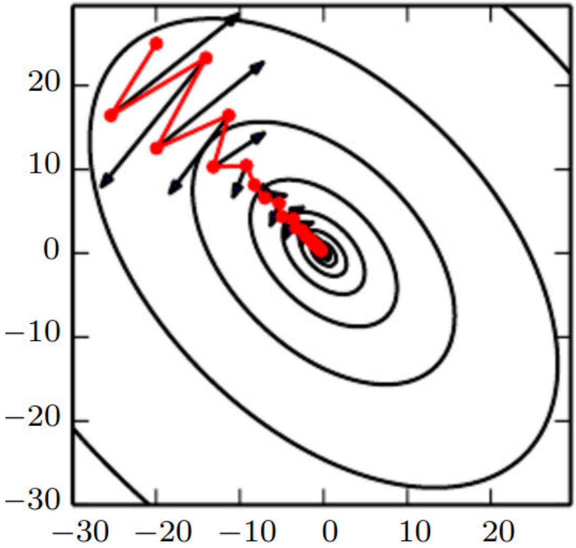


Solution: In every step, add the change made in the previous step (weighted by a factor α):

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

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

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

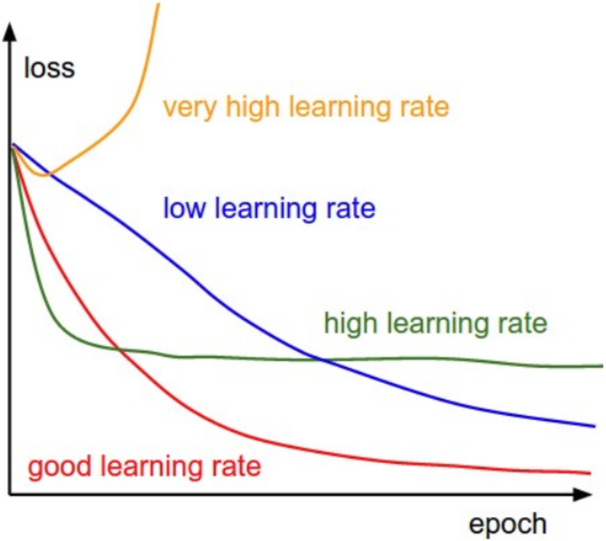
where

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

- $0 < \varepsilon(t) \leq 1$ 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 is a constant number

  (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 is a constant number

  (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 is a constant number

  (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 is a constant number

  (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 the CIFAR10 dataset).

For a 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 rates 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 gradients throughout the 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 and Adam (RMSProp combined with momentum).

A natural question: Which algorithm should one choose?

Other optimization methods

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

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 and Adam (RMSProp combined with momentum).

A natural question: Which algorithm should one choose?

Unfortunately, there is currently no consensus on this point.

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

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

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

Choice of (hidden) activations

Generic requirements imposed on activation functions:

1. differentiability

   (to do gradient descent)

2. non-linearity

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

3. monotonicity

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

4. "linearity"

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

The choice of activation functions is closely related to input preprocessing and the initial choice of weights.

Input preprocessing

- Some inputs may be much larger than others.

  For example, the height vs. weight of a person, the max. speed of a car (in km/h) vs. its price (in CZK), etc.

Input preprocessing

- Some inputs may be much larger than others.

  For example, the height vs. weight of a person, the max. speed of a car (in km/h) vs. its price (in CZK), etc.

- Large inputs have a greater influence on the training than the small ones. Also, too large inputs may slow down learning (saturation of some activation functions).

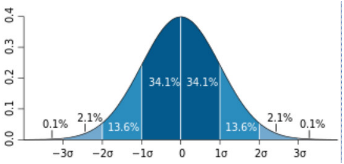
Input preprocessing

- Some inputs may be much larger than others.

  For example, the height vs. weight of a person, the max. speed of a car (in km/h) vs. its price (in CZK), etc.

- Large inputs have a greater influence on the training than the small ones. Also, too large inputs may slow down learning (saturation of some 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 the data (the training set).



(illustration of standard deviation)

Initial weights - intuition

- Assume weights are chosen randomly. What distribution?

Initial weights - intuition

- Assume weights are chosen randomly. What distribution?

  Consider the behavior of a deep network:
  - Small weights make the values of inner potentials vanish.
  - Large weights make the values of inner potentials explode.

  Hence, we want to choose weights so that the inner potentials of neurons are stable
  (similar in all layers of the network).

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 are 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 are 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 are 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*:

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

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Consider all w_i and x_i as **independent** random variables (hence also ξ is a random variable) where

- $w_i \in \mathcal{N}(0, w^2)$ for $i = 1, \ldots, n$ where w is a constant,
- $\mathbb{E}x_i = 0$ and $Var[x_i] = \mathbb{E}[(x_i - \mathbb{E}x_i)^2] = 1$ for $i = 1, \ldots, n$

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Consider all w_i and x_i as **independent** random variables (hence also ξ is a random variable) where

- $w_i \in \mathcal{N}(0, w^2)$ for $i = 1, \ldots, n$ where w is a constant,
- $\mathbb{E}x_i = 0$ and $Var[x_i] = \mathbb{E}[(x_i - \mathbb{E}x_i)^2] = 1$ for $i = 1, \ldots, n$

We prove that $Var[\xi] = n \cdot w^2$ as follows:

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Consider all w_i and x_i as **independent** random variables (hence also ξ is a random variable) where

- $w_i \in \mathcal{N}(0, w^2)$ for $i = 1, \ldots, n$ where w is a constant,
- $\mathbb{E}x_i = 0$ and $Var[x_i] = \mathbb{E}[(x_i - \mathbb{E}x_i)^2] = 1$ for $i = 1, \ldots, n$

We prove that $Var[\xi] = n \cdot w^2$ as follows:

$$\mathbb{E}\xi = \mathbb{E}\sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} \mathbb{E}w_i x_i \overset{ind.}{=} \sum_{i=1}^{n} \mathbb{E}w_i \mathbb{E}x_i = 0$$

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Consider all w_i and x_i as **independent** random variables (hence also ξ is a random variable) where

- $w_i \in \mathcal{N}(0, w^2)$ for $i = 1, \ldots, n$ where w is a constant,
- $\mathbb{E}x_i = 0$ and $Var[x_i] = \mathbb{E}[(x_i - \mathbb{E}x_i)^2] = 1$ for $i = 1, \ldots, n$

We prove that $Var[\xi] = n \cdot w^2$ as follows:

$$\mathbb{E}\xi = \mathbb{E} \sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} \mathbb{E} w_i x_i \overset{ind.}{=} \sum_{i=1}^{n} \mathbb{E} w_i \mathbb{E} x_i = 0$$

and $Var[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - \mathbb{E}[w_i x_i]^2 \overset{ind.}{=} \mathbb{E}[w_i^2]\mathbb{E}[x_i^2] - 0 = w^2$

Derivation of the LeCun initialization

Consider a single neuron without bias with the inner potential

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

Consider all w_i and x_i as **independent** random variables (hence also ξ is a random variable) where

- $w_i \in \mathcal{N}(0, w^2)$ for $i = 1, \ldots, n$ where w is a constant,
- $\mathbb{E}x_i = 0$ and $Var[x_i] = \mathbb{E}[(x_i - \mathbb{E}x_i)^2] = 1$ for $i = 1, \ldots, n$

We prove that $Var[\xi] = n \cdot w^2$ as follows:

$$\mathbb{E}\xi = \mathbb{E}\sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} \mathbb{E}w_i x_i \stackrel{ind.}{=} \sum_{i=1}^{n} \mathbb{E}w_i \mathbb{E}x_i = 0$$

and $Var[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - \mathbb{E}[w_i x_i]^2 \stackrel{ind.}{=} \mathbb{E}[w_i^2]\mathbb{E}[x_i^2] - 0 = w^2$ implies

$$Var[\xi] = Var[\sum_{i=1}^{n} w_i x_i] \stackrel{ind.}{=} \sum_{i=1}^{n} Var[w_i x_i] = \sum_{i=1}^{n} w^2 = n \cdot w^2$$

Normal Glorot initialization

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

Normal Glorot initialization

The previous heuristic for weight initialization ignores the 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 heuristic for weight initialization ignores the 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 are 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.

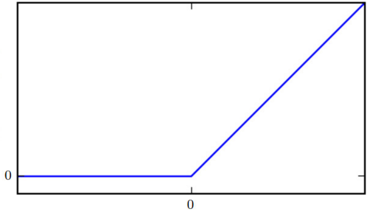
We obtain *uniform Glorot initialization* (aka *uniform Xavier init.*):

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

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 activation functions. Most prominent is ReLU:



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

- THE default activation function recommended for 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 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 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 (green board):
  - Generalizes ReLU, not dead for negative potentials.
  - Experimentally not much better than ReLU.

More modern activation functions

- Leaky ReLU (green board):
  - 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

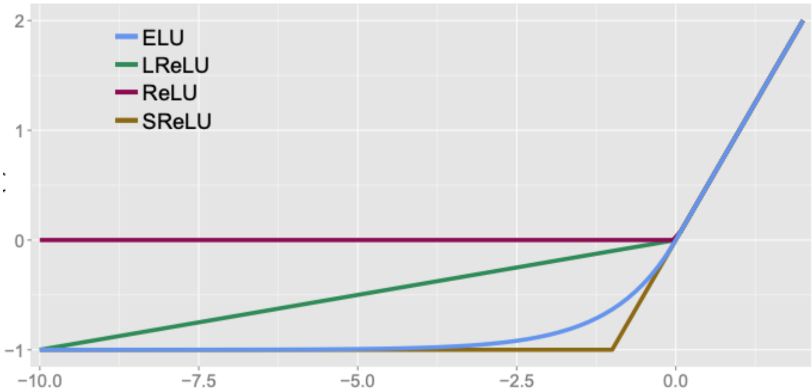
- Leaky ReLU (green board):
  - Generalizes ReLU, not dead for negative potentials.
  - Experimentally not much better than ReLU.
- ELU: "Smoothed" ReLU:

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

  Here α is a parameter, ELU converges to $-\alpha$ as $\xi \rightarrow -\infty$. 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).



ELU
LReLU
ReLU
SReLU
2
1
0
−1
−10.0
−7.5
−5.0
−2.5
0.0

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

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

Batch normalization (roughly)

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

Consider a batch of training examples:

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

(This is typically a minibatch.)

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

$$\vec{z}_k = (z_{k1}, \dots, z_{kn})$$

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

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

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

- 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.

Normalization

During the **training**, the normalized vectors $\hat{z}_1, \ldots, \hat{z}_p$ are computed as follows:

$$\hat{z}_{ki} = \frac{z_{ki} - \mu_i}{\sigma_i}$$

Here

$$\mu_i = \frac{1}{p} \sum_{k=1}^{p} z_{ki}$$

$$\sigma_i = \sqrt{\frac{1}{p} \sum_{k=1}^{p} (z_{ki} - \mu_i)^2}$$

During **inference**, where we have just a single value $\vec{z}$ of the layer ℓ for an input $\vec{x}$, we use μ_i and σ_i estimated on a population (e.g., a larger sample of the training set).

Generalization

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

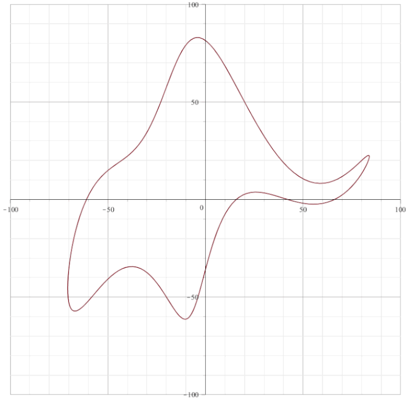
Regularization is a big issue in neural networks, as they typically use a huge amount of parameters and thus are very susceptible to overfitting.

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."**

Elephant

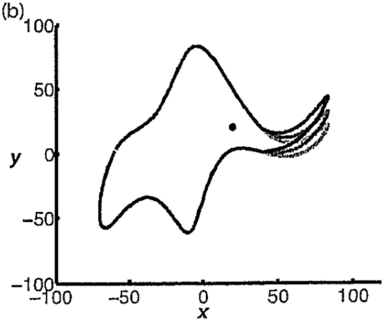


$$x(t) = -60\cos(t) + 30\sin(t) - 8\sin(2t) + 10\sin(3t)$$
$$y(t) = 50\sin(t) + 18\sin(2t) - 12\cos(3t) + 14\cos(5t)$$

The four parameters are complex numbers (e.g., $-60 + 50i$).

Mayer, Jurgen; Khairy, Khaled; Howard, Jonathon (May 12, 2010). "Drawing an elephant with four complex parameters". American Journal of Physics. 78 (6)

Fifth Elephant



Parameter	Real part	Imaginary part
$p_1 = 50 - 30i$	$B_1^x = 50$	$B_1^y = -30$
$p_2 = 18 + 8i$	$B_2^x = 18$	$B_2^y = 8$
$p_3 = 12 - 10i$	$A_3^x = 12$	$B_3^y = -10$
$p_4 = -14 - 60i$	$A_5^x = -14$	$A_1^y = -60$
$p_5 = 40 + 20i$	Wiggle coeff. = 40	$x_{eye} = y_{eye} = 20$

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.

(some 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.

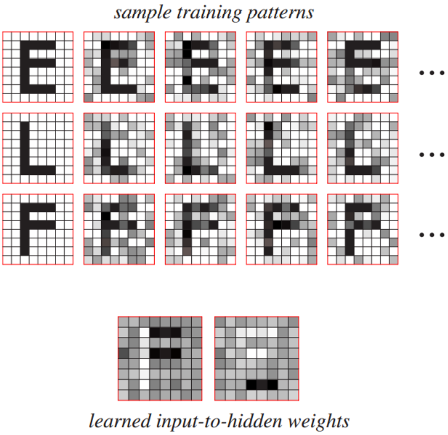
  If you are trully desperate trying to solve a brand new problem, you may try an ancient rule of thumb: the number of neurons $\approx$ ten times less than the number of training instances.

  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 $T_1, ..., T_k$ 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 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 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**:

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

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

(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:

$$w_{ji}^{(t+1)} = (1 - \zeta) w_{ji}^{(t)} - \varepsilon \cdot \frac{\partial E}{\partial w_{ji}} (\vec{w}^{(t)})$$

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:

$$w_{ji}^{(t+1)} = (1 - \zeta) w_{ji}^{(t)} - \varepsilon \cdot \frac{\partial E}{\partial w_{ji}} (\vec{w}^{(t)})$$

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:

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

Here $\frac{\zeta}{2\varepsilon}(\vec{w} \cdot \vec{w})$ is the L2 regularization that penalizes large weights.
We use the gradient descent with a constant learning rate to illustrate the equivalence between L2 regularization and the weight decay. Both methods can be combined with other learning algorithnms (AdaGrad, etc.).

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.