Artificial neural networks and backpropagation

E. Decencière

MINES ParisTech
PSL Research University
Center for Mathematical Morphology

Contents

Contents

Artificial neural networks and deep learning history

For a very complete state of the art on deep learning, see the overview by Schmidhuber [Schmidhuber, 2015].

- 1958: Rosenblatt's perceptron [Rosenblatt, 1958]

Artificial neural networks and deep learning history

For a very complete state of the art on deep learning, see the overview by Schmidhuber [Schmidhuber, 2015].

- 1958: Rosenblatt's perceptron [Rosenblatt, 1958]
- 1979: Neocognitron (convolutional neural network architecture) [Fukushima, 1979, Fukushima, 1980]

Artificial neural networks and deep learning history

For a very complete state of the art on deep learning, see the overview by Schmidhuber [Schmidhuber, 2015].

- 1958: Rosenblatt's perceptron [Rosenblatt, 1958]
- 1979: Neocognitron (convolutional neural network architecture) [Fukushima, 1979, Fukushima, 1980]
- 1980's: the backpropagation algorithm (see, for example, the work of LeCun [LeCun, 1985])

Artificial neural networks and deep learning history

For a very complete state of the art on deep learning, see the overview by Schmidhuber [Schmidhuber, 2015].

- 1958: Rosenblatt's perceptron [Rosenblatt, 1958]
- 1979: Neocognitron (convolutional neural network architecture) [Fukushima, 1979, Fukushima, 1980]
- 1980's: the backpropagation algorithm (see, for example, the work of LeCun [LeCun, 1985])
- 2006-: CNN implementations using Graphical Processing Units (GPU): up to a 50 speed-up factor.

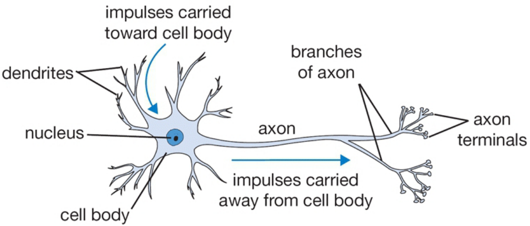
Artificial neural networks and deep learning history

For a very complete state of the art on deep learning, see the overview by Schmidhuber [Schmidhuber, 2015].

- 1958: Rosenblatt's perceptron [Rosenblatt, 1958]
- 1979: Neocognitron (convolutional neural network architecture) [Fukushima, 1979, Fukushima, 1980]
- 1980's: the backpropagation algorithm (see, for example, the work of LeCun [LeCun, 1985])
- 2006-: CNN implementations using Graphical Processing Units (GPU): up to a 50 speed-up factor.
- 2012: Imagenet image classification won by a CNN with AlexNet [Krizhevsky et al., 2012].

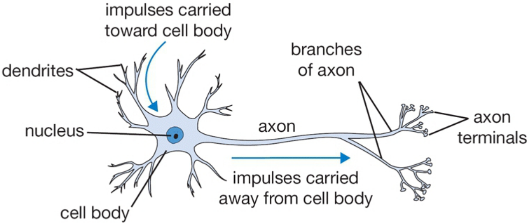
Contents

Neuron



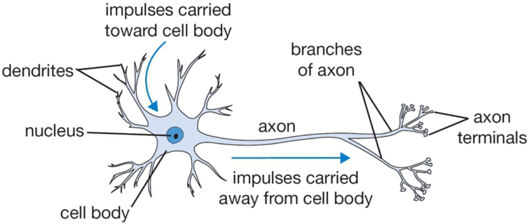
- The human brain contains 100 billion (10^{11}) neurons

Neuron



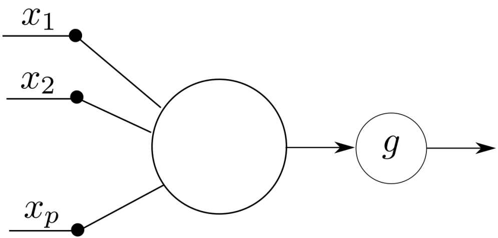
- The human brain contains 100 billion (10^{11}) neurons
- A human neuron can have several thousand dendrites

Neuron



- The human brain contains 100 billion (10^{11}) neurons
- A human neuron can have several thousand dendrites
- The neuron sends a signal through its axon if during a given interval of time the net input signal (sum on excitatory and inhibitory signals received through its dendrites) is larger than a threshold.

Artificial neuron

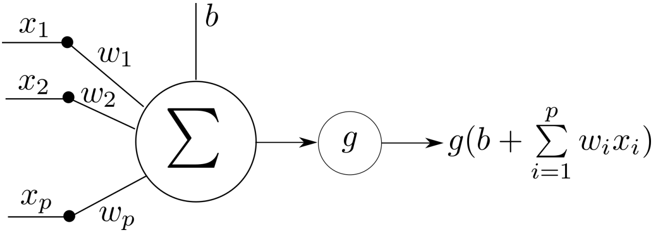


General principle

An artificial neuron takes p inputs $\{x_i\}_{1 \le i \le p}$, combines them to obtain a single value, and applies an activation function g to the result.

- The first artificial neuron model was proposed by [McCulloch and Pitts, 1943]
- Input and output signals were binary
- Input dendrites could be inhibitory or excitatory

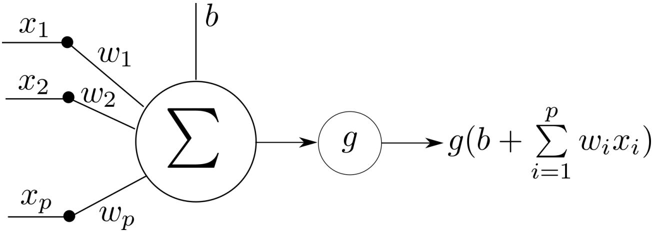
Modern artificial neuron



- The neuron computes a linear combination of the inputs x_i
  - The weights w_i are multiplied with the inputs
  - The bias b can be interpreted as a threshold on the sum
- The activation function g somehow decides, depending on its input, if a signal (the neuron's activation) is produced

Contents

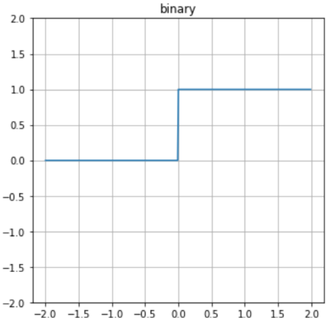
The role of the activation function



- The initial idea behind the activation function is that it works somehow as a gate
- If its input in "high enough", then the neuron is activated, i.e. a signal (other than zero) is produced
- It can be interpreted as a source of abstraction: information considered as unimportant is ignored

Activation: binary

$$g(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

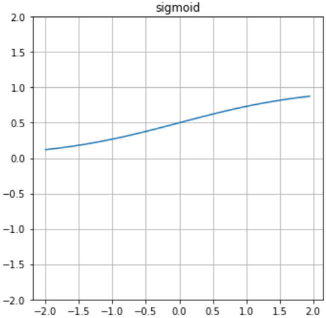


Remarks

- Biologically inspired
- \+ Simple to compute
- \+ High abstraction
- \- Gradient nil except on one point
- In practice, almost never used

Activation: sigmoid

$$g(x) = \frac{1}{1 + e^{-x}}$$

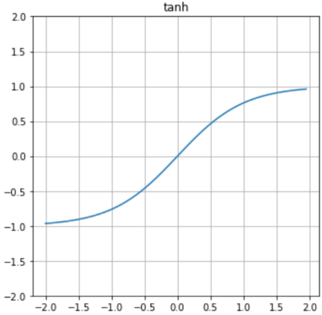


Remarks

+ Similar to binary activation, but with usable gradient
- However, gradient tends to zero when input is far from zero
- More computationally intensive

Activation: hyperbolic tangent

$$\mathrm{g}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

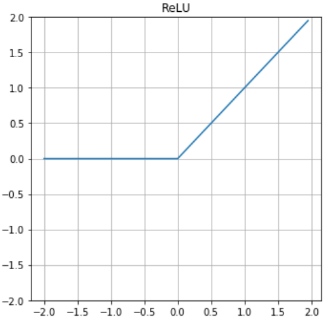


Remarks

- Similar to sigmoid

Activation: rectified linear unit

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

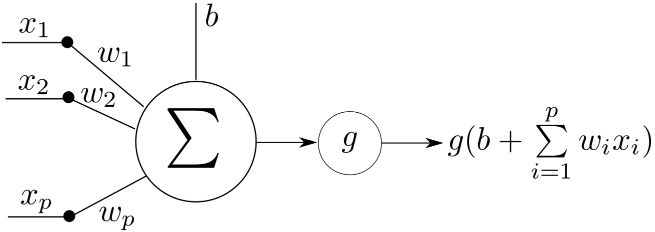


Remarks

+ Usable gradient when activated
+ Fast to compute
+ High abstraction

Contents

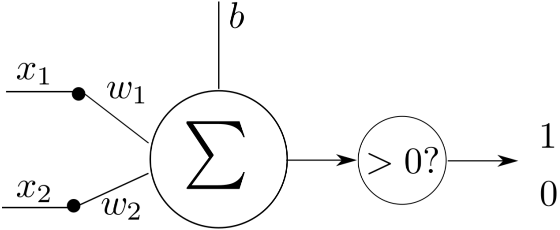
What can an artificial neuron compute?



In $\mathbb{R}^p$, $b + \sum_{i=0}^{p} w_i x_i = 0$ corresponds to a hyperplane. For a given point $\mathbf{x} = \{x_0, \dots, x_p\}$, decisions are made according to the side of the hyperplane it belongs to.

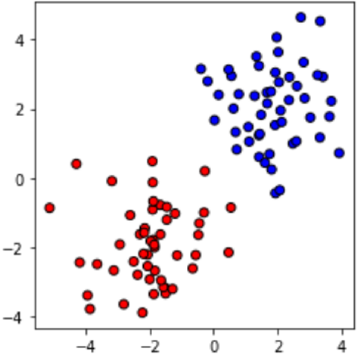
When the activation function is binary, we obtain a perceptron

Example of what we can do with a neuron

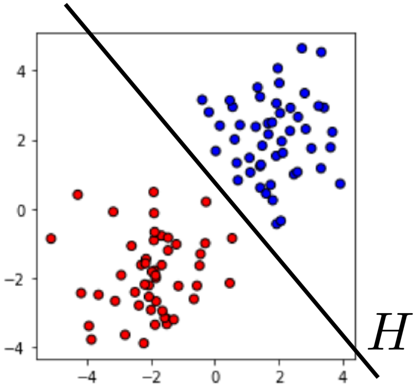


- $p = 2$: 2-dimensional inputs (can be represented on a screen!)
- Activation: binary
- Classification problem

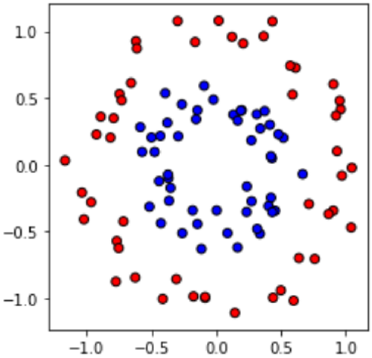
Gaussian clouds



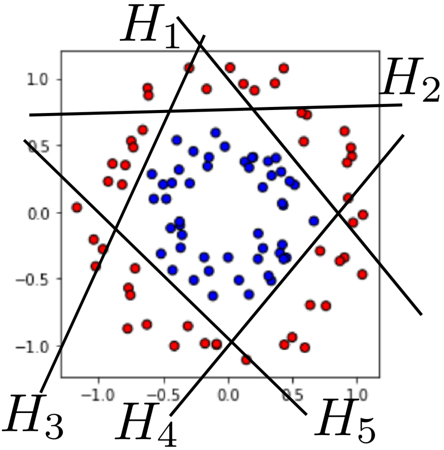
Gaussian clouds



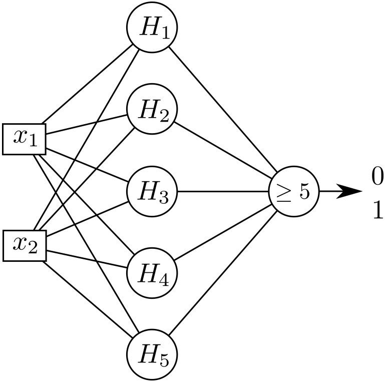
Circles



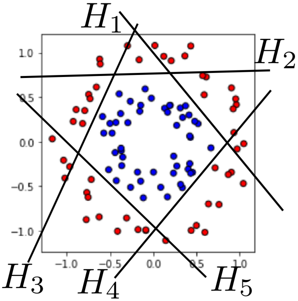
Circles



Solution

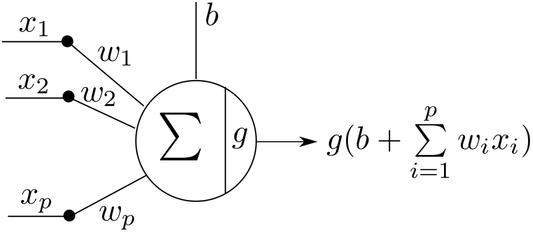


H_1
H_2
H_3
H_4
H_5
x_1
x_2
≥ 5
0
1



H_1
H_2
H_3
H_4
H_5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

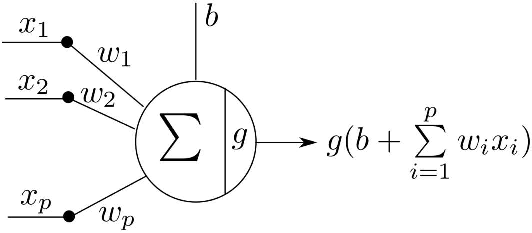
Artificial neuron compact representation



Contents

Contents

Notations



With

$$\mathbf{w} = (w_1, \ldots, w_p)^T$$
$$\mathbf{x} = (x_1, \ldots, x_p)^T$$

We can simply write:

$$\mathrm{g}(b + \sum_{i=1}^{p} w_i x_i) = \mathrm{g}(b + \mathbf{w}^T \mathbf{x})$$

Computation graph

Definition

A computation graph is a directly acyclic graph such that:

- A node is a mathematical operator
- To each edge is associated a value
- Each node can compute the values of its output edges from the values of its input edges
  - Nodes without input edges are *input nodes*. They represent the input values of the graph.
  - Similarly, output values can be held in the *output nodes*.

Computing a *forward pass* through the graph means choosing its values, and then progressively computing the values of all edges.

Computation graph example

We will compute:

$$\sigma(w_1 x + w_2 y + b)$$

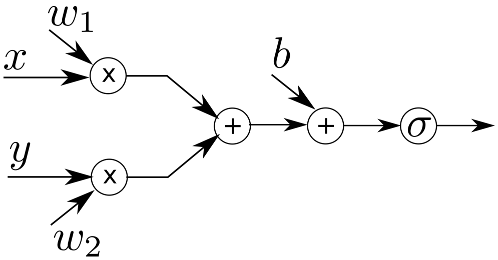
where σ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

Computation graph example

We will compute:

$$\sigma(w_1 x + w_2 y + b)$$

where σ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$



Neural network (NN)

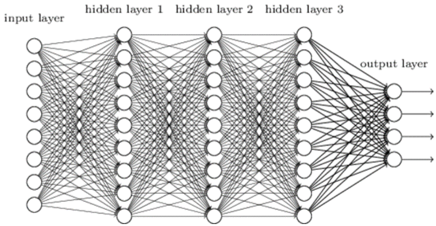
Definitions

- An artificial neural network is a computation graph, where the nodes are artificial neurons
- The input layer is the set of neurons without incoming edges.
- The output layer is the set of neurons without outgoing edges.

Feed-forward neural networks

Definition

- A feed-forward neural networks is a NN without cycles
- Neurons are organized in layers
  - A neuron belongs to layer q if the longest path in the graph between the input layer and the neuron is of length q.
- Any layers other than input and output layers are called hidden layers



(from http://www.jtoy.net)

Feed-forward neural networks

In the following of this course, except when otherwise specified, all NNs will be feed-forward. Indeed, this is the preferred type of NN for image processing.

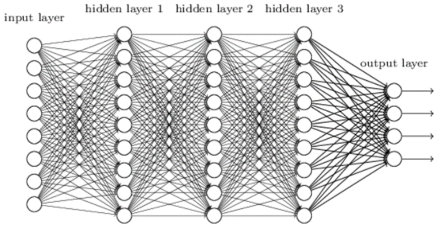
What about other architectures?

- Recurrent neural networks (RNN)
- Long short-term memory networks (LSTM)

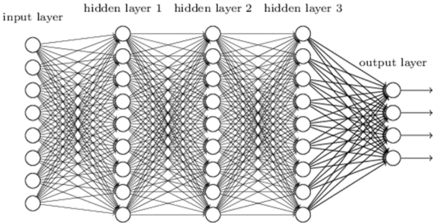
+ More powerful than feed-forward NNs
- Complex dynamics; more difficult to train
- Mainly used for processing temporal data

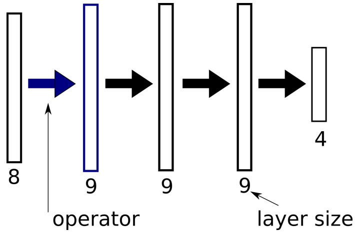
Fully-connected network

- A layer is said to be fully-connected (FC) if each of its neurons is connected to all the neurons of the previous and following layers
- If a FC layer contains r neurons, and the previous layer q, then its weights are 2D dimensional array (a matrix) of size $q \times r$
- A NN is said to be fully connected if all its hidden layers are fully connected



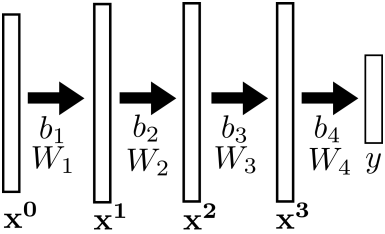
Graphical representation of NNs





- Data is organized into arrays, linked with operators
- A layer corresponds to an operator between arrays (and often an activation) as well as the resulting array.

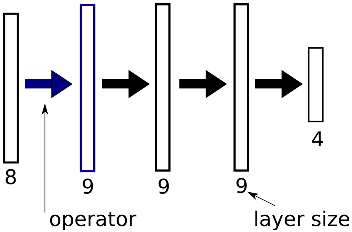
The equations of a fully connected neural network



$$\mathbf{x}^i = \mathbf{g}_i(\mathbf{x}^{i-1}\mathbf{W}_i + \mathbf{b}_i),\ i = 1, 2, 3$$

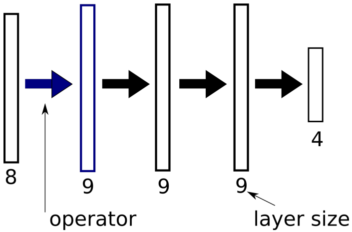
$$y = \mathbf{g}_4(\mathbf{x}^4\mathbf{W}_4 + \mathbf{b}_4)$$

Number of parameters



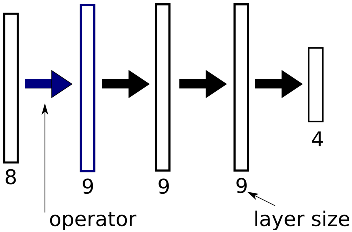
- How many parameters does the above network contain?

Number of parameters



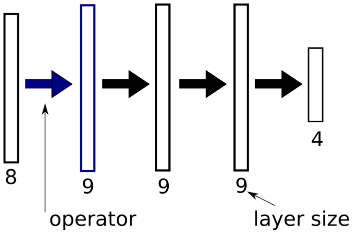
- How many parameters does the above network contain?
- First hidden layer:

Number of parameters



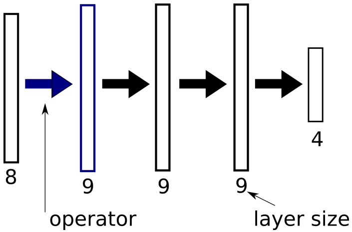
- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$

Number of parameters



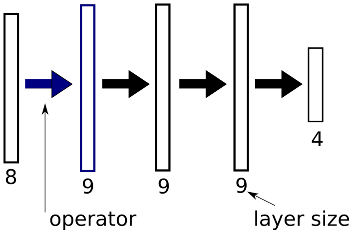
- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers:

Number of parameters



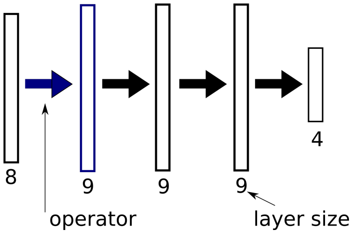
- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$

Number of parameters



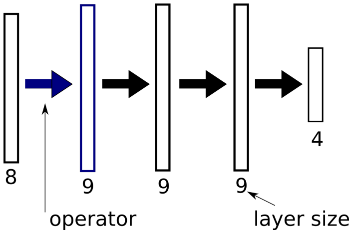
- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer:

Number of parameters



- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer: $4 \times 9 + 4$

Number of parameters



- How many parameters does the above network contain?
- First hidden layer:
  - 9 neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer: $4 \times 9 + 4$
- Total: 305 parameters

Batch processing

In a training context, our learning set contains n samples of vectors of length p, that can be grouped into a matrix X of size $n \times p$. The n corresponding outputs y_i can also be grouped into a vector $\mathbf{y}$ of length n. The resulting equations are:

$$\mathbf{X}^i = \mathrm{g}_i(\mathbf{X}^{i-1}\mathbf{W}_i + \mathbf{b}_i),\ i = 1, 2, 3$$

$$\mathbf{y} = \mathrm{g}_4(\mathbf{X}^4\mathbf{W}_4 + \mathbf{b}_4)$$

Mini-batch processing

- When dealing with large databases (large n and sometimes large p) for practical reasons the network cannot process the whole set in a single pass.

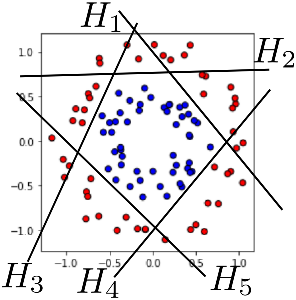
Mini-batch processing

- When dealing with large databases (large n and sometimes large p) for practical reasons the network cannot process the whole set in a single pass.
- One can also separate the training databases into subsets containing m samples ($m < n$), called *mini-batches*.

Contents

Universal approximation theorem

- We have previously seen that a neuron can be used as a linear classifier and that combining several of them one can build complex classifiers
- We will see that this observation can be generalized



Universal approximation theorem

Let f be a continuous real-valued function of $[0, 1]^p$ ($p \in \mathbb{N}^*$) and ϵ a strictly positive real. Let g be a non-constant, increasing, bounded real function (*the activation function*).
Then there exists an integer n, real vectors $\{\mathbf{w}_i\}_{1 \leq i \leq n}$ of $\mathbb{R}^p$, and reals $\{b_i\}_{1 \leq i \leq n}$ and $\{v_i\}_{1 \leq i \leq n}$ such that for all $\mathbf{x}$ in $[0, 1]^p$:

$$\left| f(\mathbf{x}) - \sum_{i=1}^{n} v_i \mathbf{g}(\mathbf{w}_i^T \mathbf{x} + b_i) \right| < \epsilon$$

A first version of this theorem, using sigmoidal activation functions, was proposed by [Cybenko, 1989]. The version above was demonstrated by [Hornik, 1991].

Universal approximation theorem: what does it mean?

$$\left| f(\mathbf{x}) - \sum_{i=1}^{n} v_i \mathbf{g}(\mathbf{w}_i^T \mathbf{x} + b_i) \right| < \epsilon$$

This means that function f can be approximated with a neural network containing:

- an input layer of size p;
- a hidden layer containing n neurons with activation function g, weights $\mathbf{w}_i$ and biases b_i;
- an output layer containing a single neuron, with weights v_i (and an identity activation function).

Universal approximation theorem in practice

- The number of neurons increases very rapidly with the complexity of the function
- Empirical evidence has shown that multi-layer architectures give better results

Universal approximation theorem in practice

- The number of neurons increases very rapidly with the complexity of the function
- Empirical evidence has shown that multi-layer architectures give better results

A NN can potentially have a lot of parameters. How can we set them?

Contents

Introduction

- We have seen that NNs have a lot of potential. However, how can the parameters $\boldsymbol{\theta} = (\mathbf{W}_i, \mathbf{b}_i)$ be set?
- What is our objective ?
- A very general solution, that is also the mostly used, is gradient descent

Learning problem

We recall that our training set contains n samples:

$$(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R}$$

We choose a family $f_{\boldsymbol{\theta}}$ of functions from $\mathbb{R}^p$ into $\mathbb{R}$, depending on our set of parameters $\boldsymbol{\theta}$, and find the value of $\boldsymbol{\theta}$ that minimizes a chosen loss function L:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}}(L(\boldsymbol{\theta}) + \mathcal{R}(\boldsymbol{\theta}))$$

where $\mathcal{R}(\boldsymbol{\theta})$ is a regularization term.

For the time being, for the sake of simplicity, we will drop the regularization term until further notice

Loss function

A general form of the loss function is:

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

where d is some disparity function (the more similar its parameters, the smaller its value).

Loss function: examples

Squared error

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2$$

This loss function is mainly used in regression problems.

Binary cross-entropy

In this case, $y_i \in \{0, 1\}$:

$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \Big(y_i \log(f(\mathbf{x}_i, \boldsymbol{\theta})) + (1 - y_i) \log(1 - f(\mathbf{x}_i, \boldsymbol{\theta})) \Big)$$

This loss function is used in binary classification problems, where the network's output can be interpreted as a probability of belonging to a class.

Gradient descent

Definition

Gradient descent is an optimization algorithm. For a derivable function L, a positive real γ (the learning rate) and a starting point $\boldsymbol{\theta}_0$, it computes a sequence of values:

$$\forall e \in \mathbb{N} : \boldsymbol{\theta}_{e+1} = \boldsymbol{\theta}_e - \gamma \nabla L(\boldsymbol{\theta}_e)$$

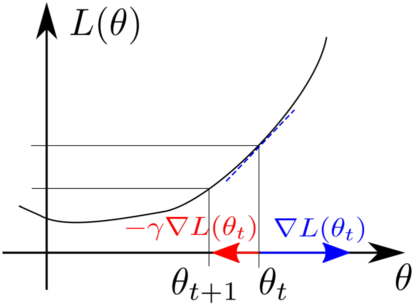
Property

If γ is small enough, then:

$$L(\boldsymbol{\theta}_{i+1}) \leq L(\boldsymbol{\theta}_i)$$

Gradient descent is an essential tool in optimization.

Gradient descent in the scalar case



$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla L(\boldsymbol{\theta}_t)$$

Gradient descent: stopping criteria

In practice:

$$\forall e \in [0, \dots, E-1]: \quad \boldsymbol{\theta}_{e+1} = \boldsymbol{\theta}_e - \gamma \nabla L(\boldsymbol{\theta}_e)$$

- Choose E (the number of epochs) based on experience
- Track the quality of the model using a validation dataset and stop when the validation loss does not improve

Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

- If n is very large, its computation is not feasible.

Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

- If n is very large, its computation is not feasible.
- A computation on the whole training set leads to a single update of the model parameters - convergence can therefore be slow.

Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample i.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample i.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed through backpropagation and

Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample i.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed through backpropagation and
- finally the parameters are updated:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla L(\boldsymbol{\theta}_t)$$

Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample i.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = d(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed through backpropagation and
- finally the parameters are updated:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla L(\boldsymbol{\theta}_t)$$

- Note that the learning rate γ can have a different value.

Gradient descent applied to neural networks

In the case of neural networks, the loss L depends on each parameter $\boldsymbol{\theta}_i$ via the composition of several simple functions. In order to compute the gradient $\nabla_{\boldsymbol{\theta}} L$ we will make extensive use of the chain rule theorem.

Chain rule theorem

Let f_1 and f_2 be two derivable real functions ($\mathbb{R} \rightarrow \mathbb{R}$). Then for all x in $\mathbb{R}$: :

$$(f_2 \circ f_1)'(x) = f_2'(f_1(x)).f_1'(x)$$

Leibniz notation

Let us introduce variables x, y and z:

$$x \xrightarrow{f_1} y \xrightarrow{f_2} z$$

Then:

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

The backpropagation algorithm

- The backpropagation algorithm is used in a neural network to efficiently compute the partial derivative of the loss with respect to each parameter of the network.
- One can trace the origins of the method to the sixties
- It was first applied to NN in the eighties [Werbos, 1982, LeCun, 1985]

Simple backpropagation example

Simple backpropagation example

$$x \xrightarrow{\;\frac{\partial y}{\partial x}\;} y \longrightarrow \qquad \longrightarrow$$

Simple backpropagation example

$$x \xrightarrow{\frac{\partial y}{\partial x}} y \xrightarrow{\frac{\partial z}{\partial y}} z \xrightarrow{\qquad}$$

Simple backpropagation example

$$x \xrightarrow{\frac{\partial y}{\partial x}} y \xrightarrow{\frac{\partial z}{\partial y}} z \xrightarrow{\frac{\partial l}{\partial z}} l$$

Simple backpropagation example

$$x \xrightarrow{\frac{\partial y}{\partial x}} y \xrightarrow{\frac{\partial z}{\partial y}} z \xrightarrow{\frac{\partial l}{\partial z}} l$$

$$\frac{\partial l}{\partial l} = 1$$

Simple backpropagation example

$$x \xrightarrow{\frac{\partial y}{\partial x}} y \xrightarrow{\frac{\partial z}{\partial y}} z \xrightarrow{\frac{\partial l}{\partial z}} l$$

$$\frac{\partial l}{\partial z} = \frac{\partial l}{\partial l}\frac{\partial l}{\partial z} \qquad \frac{\partial l}{\partial l} = 1$$

Simple backpropagation example

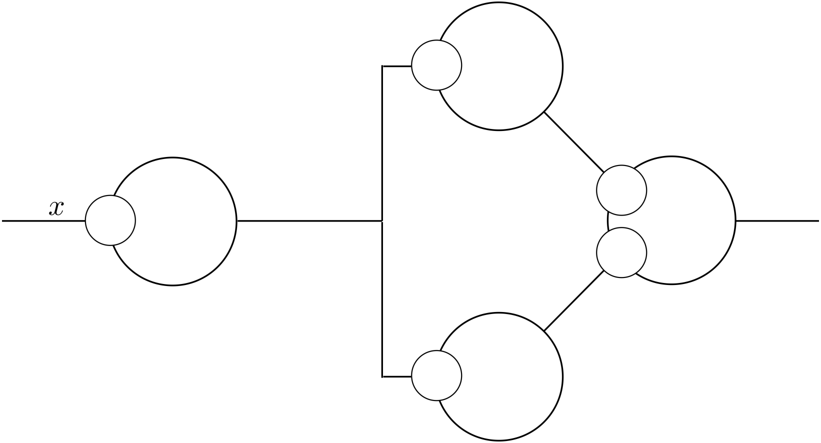
$$
\begin{tikzcd}
x \arrow[r, "\frac{\partial y}{\partial x}"] & y \arrow[r, "\frac{\partial z}{\partial y}"] & z \arrow[r, "\frac{\partial l}{\partial z}"] & l \\
 & \frac{\partial l}{\partial y} = \frac{\partial l}{\partial z} \frac{\partial z}{\partial y} & \frac{\partial l}{\partial z} = \frac{\partial l}{\partial l} \frac{\partial l}{\partial z} & \frac{\partial l}{\partial l} = 1
\end{tikzcd}
$$

Simple backpropagation example

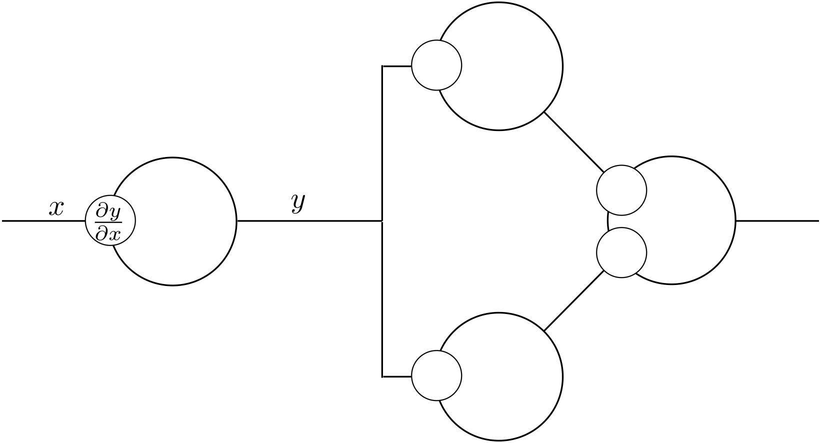
$$x \xrightarrow{\frac{\partial y}{\partial x}} y \xrightarrow{\frac{\partial z}{\partial y}} z \xrightarrow{\frac{\partial l}{\partial z}} l$$

$$\frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} \qquad \frac{\partial l}{\partial y} = \frac{\partial l}{\partial z}\frac{\partial z}{\partial y} \qquad \frac{\partial l}{\partial z} = \frac{\partial l}{\partial l}\frac{\partial l}{\partial z} \qquad \frac{\partial l}{\partial l} = 1$$

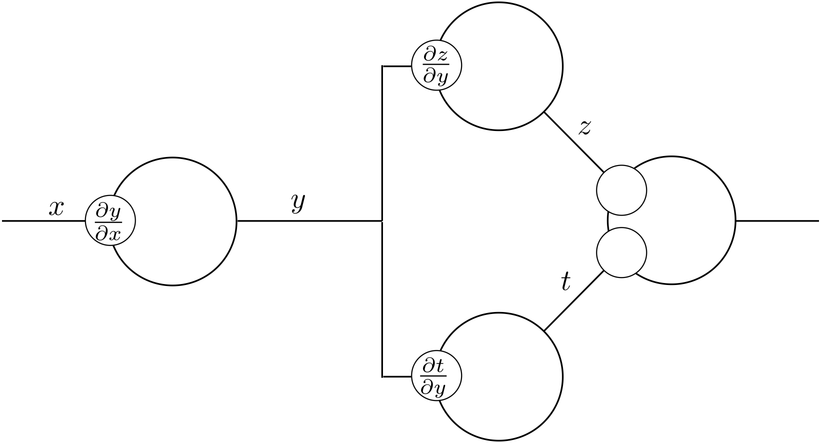
Backpropagation example



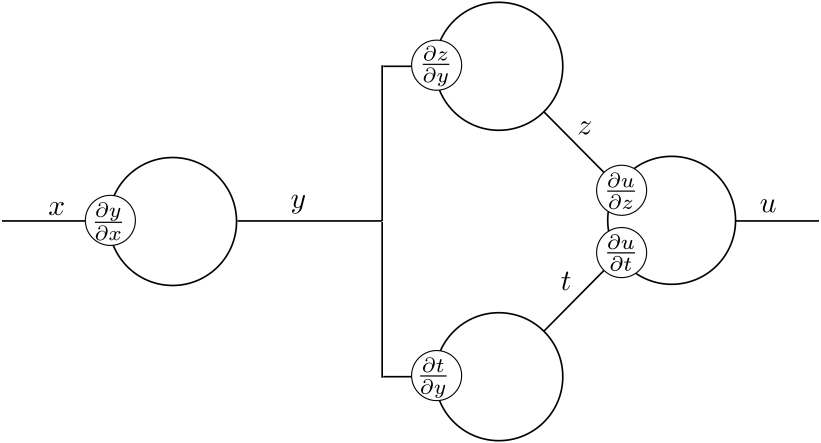
Backpropagation example



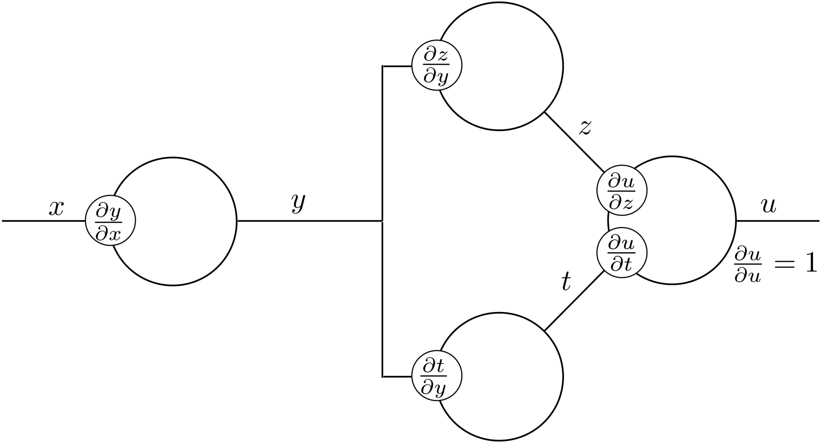
Backpropagation example



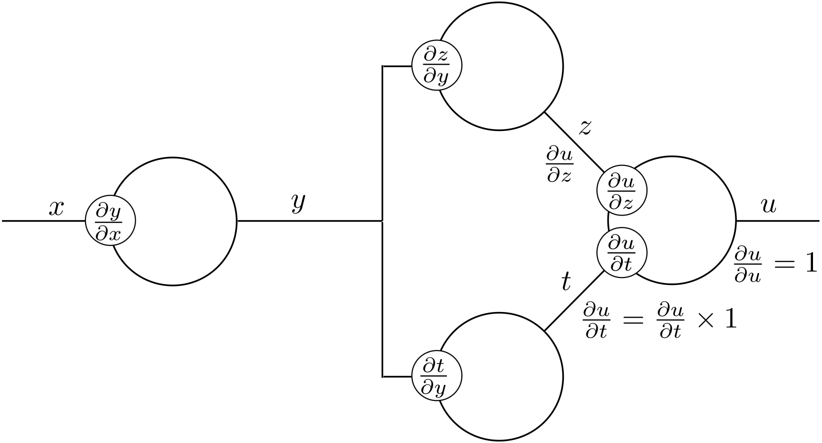
Backpropagation example



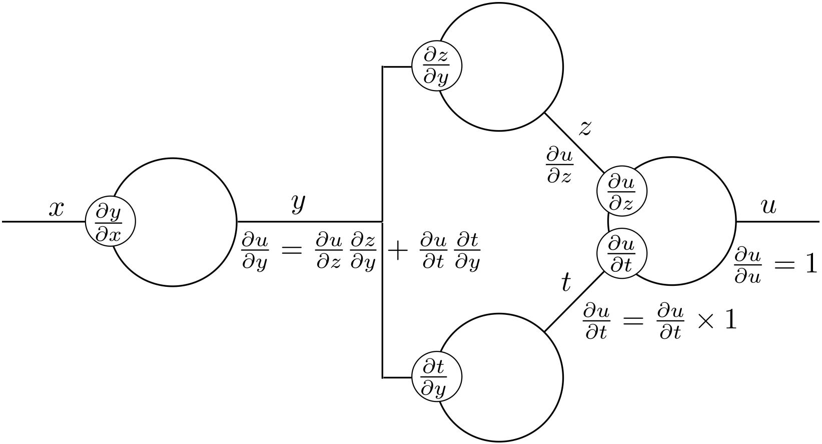
Backpropagation example



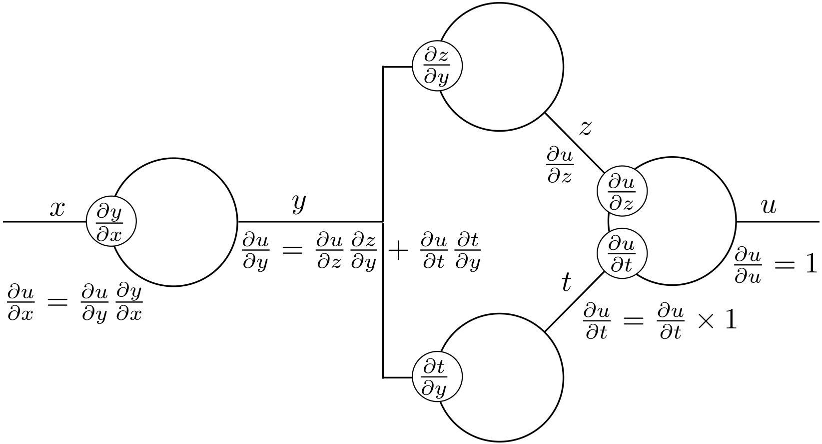
Backpropagation example



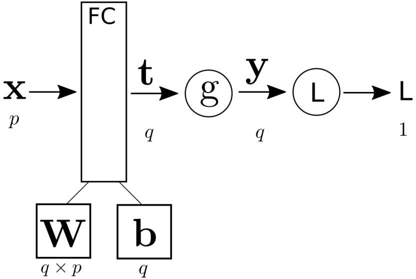
Backpropagation example



Backpropagation example



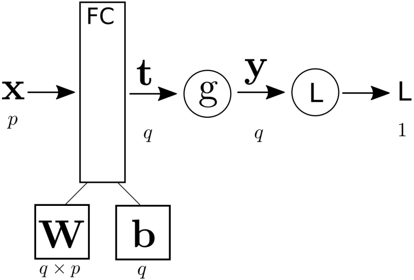
Backpropagation through a fully connected layer



Setup:

$$
\begin{aligned}
p, q &\in \mathbb{N}^* \\
\mathbf{x} &\in \mathbb{R}^p \\
\mathbf{W} &\in \mathbb{R}^q \times \mathbb{R}^p \\
\mathbf{b}, \mathbf{t}, \mathbf{y} &\in \mathbb{R}^q \\
L &\in \mathbb{R}
\end{aligned}
$$

Backpropagation through a fully connected layer



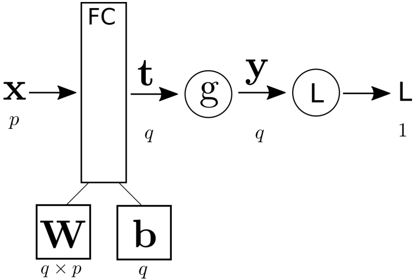
Local gradients:

Forward pass:

$$
\begin{aligned}
\mathbf{t} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\
\mathbf{y} &= \mathrm{g}(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
L &= L(\mathbf{y})
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathbf{t}}{\partial \mathbf{W}} &= \mathbf{x}^t \\
\frac{\partial \mathbf{t}}{\partial \mathbf{b}} &= 1 \\
\frac{\partial \mathbf{y}}{\partial \mathbf{t}} &= \mathrm{g}'
\end{aligned}
$$

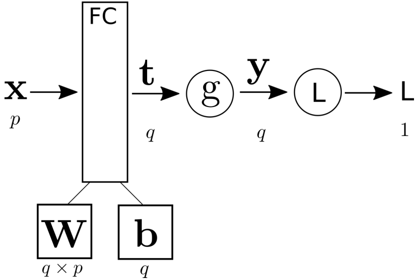
Backpropagation through a fully connected layer



Backpropagation:

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{t}} &= \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{t}} \\
&= \frac{\partial L}{\partial \mathbf{y}} \odot \mathbf{g}'(\mathbf{t})
\end{aligned}
$$

Backpropagation through a fully connected layer



Backpropagation:

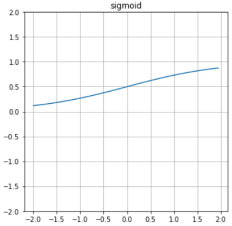
$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}} &= \frac{\partial L}{\partial \mathbf{t}} . \frac{\partial \mathbf{t}}{\partial \mathbf{W}} \\ &= \frac{\partial L}{\partial \mathbf{y}} \odot \mathrm{g}'(\mathbf{t}).\mathbf{x}^t\end{aligned}$$

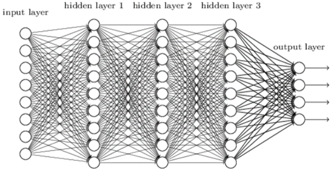
$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{y}} \odot \mathrm{g}'(\mathbf{t})$$

Network parameters initialization

General idea

Inputs of activation functions should be in an appropriate range (high gradient)





- If all parameters are initialized to zero, then in each layer the activations will remain equal – symmetry will never be broken
- Simple solution: random values from a normal or uniform distribution
- More advanced solutions exist: [LeCun et al., 1998, Glorot and Bengio, 2010, He et al., 2015]

Conclusion

We have seen:

- What is an artificial neuron and an artificial neural network (NN)
- The (potential) power of a NN
- The backpropagation algorithm
- NN learning basics

In the following, we will see how to process images using NNs.

References I

[Cybenko, 1989] Cybenko, G. (1989). Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192.

[Fukushima, 1979] Fukushima, K. (1979). Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position- Neocognitron. *ELECTRON. & COMMUN. JAPAN*, 62(10):11–18.

[Fukushima, 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*. arXiv: 1502.01852.

[Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

References II

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[LeCun, 1985] LeCun, Y. (1985). Une procedure d'apprentissage pour reseau a seuil asymmetrique (A learning scheme for asymmetric threshold networks). In *proceedings of Cognitiva 85*.

[LeCun et al., 1998] LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient BackProp. In Orr, G. B. and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pages 9–50. Springer.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

References III

[Werbos, 1982] Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kozin, F., editors, *System Modeling and Optimization*, Lecture Notes in Control and Information Sciences, pages 762–770. Springer Berlin Heidelberg.