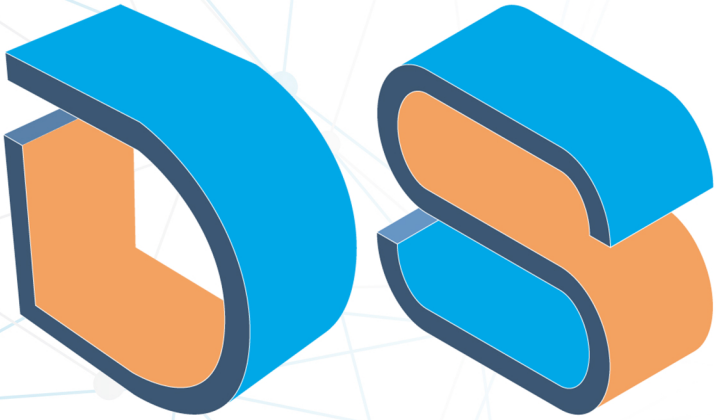


POLIMI
DATA SCIENTISTS

Soft Computing

Neural Networks Course Notes

Edited by:
Marco Varrone

These notes have been made thanks to the effort of Polimi Data Scientists staff.

Are you interested in Data Science activities?

Follow PoliMi Data Scientists on Facebook!

Polimi Data Scientist is a community of students and Alumni of Politecnico di Milano.

We organize events and activities related to Artificial Intelligence and Machine Learning, our aim is to create a strong and passionate community about Data Science at Politecnico di Milano.

Do you want to learn more?
Visit our website and join our Telegram Group! !

Credits

The following notes have been written by the Polimi Data Scientists student association by combining Prof. Matteucci's lectures, slides and multiple online resources, such as:

- https://blog.paperspace.com/vanishing-gradients-activation-function
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

Contents

Chapter 1

Introduction to Machine Learning

A computer program is said to learn from experience E with respect to some class of task T and a performance measure P, if its performance at task in T, as measured by P, improves because of experience E.
Machine Learning can be defined as the study or development of models and algorithms that make systems automatically improve their performance during execution. We call them Adaptive Models.

A model is typically a function f that receives an **input** (also called features, observations or independent variables) and returns an **output** (also called classes or dependent variables).
The shape of the function can be determined either with a top-down approach (Fuzzy Logic, Genetic Algorithms) or with a bottom-up approach (Machine Learning, Neural Networks). In the latter approach, the system observes a set of examples and builds a model based on these data.

1.1 Learning Paradigms

It is possible to distinguish different learning paradigms based on the type of samples used for training and the purpose of the model:

- Supervised Learning $< x_i, t_i >$: for each input x_i, given the desired output t_i, learn to produce the correct output y given new a input x. The model must be able to perform generalization: provide output for data it has never seen before.

- Unsupervised Learning $< x_i >$: exploits regularities in data to build a representation that can be used for reasoning or prediction. For example identify groups of samples with similar characteristics.

- Reinforcement Learning $< x_i >, \{a_1, a_2, ...\}, r_i \in \mathbb{R}$: producing actions $a_1, a_2, ...$ which affect the environment, and receiving rewards $r_1, r_2, ...$ Learn to act in a way that maximizes rewards in the long term.

We will focus mainly on Supervised Learning.

1.2 Supervised Learning

- Task T: extract from a finite set of examples a model of the observed phenomenon to be used in the future for prediction of decision making about it

- Experience E: a set of examples for the desired behavior pre-processed by an expert (the supervisor) as pairs input/output

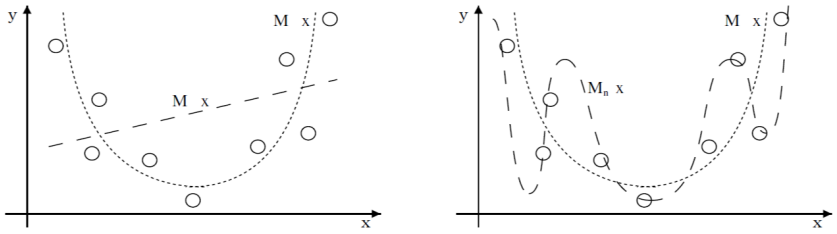
- Performance P: the measure of the distance between the desired output for new examples and the output provided by the model

1.2.1 Terminology

Before further analysis, some terms have to be defined:
Classification: the desired outputs t_i are discrete class labels (i.e. $t_i \in \{\Omega_0, \Omega_1, ..., \Omega_k\}$) . The goal is to classify new input correctly.
Regression: the desired outputs t_i are continuous values (i.e. $t_i \in \mathbb{R}$). The goal is to predict the output accurately for new inputs.
Inductive Hypothesis: a solution that approximates the target function over a sufficiently large set of training examples and will also approximate the target function over unobserved examples.
Model Complexity: the number of different target functions a model can approximate (e.g. linear, quadratic, etc.)

1.2.2 Choosing model complexity

Suppose that the training dataset is taken from a 2^{nd} order polynomial (but we don't know that, otherwise we would already know the right complexity), with some noise in it.



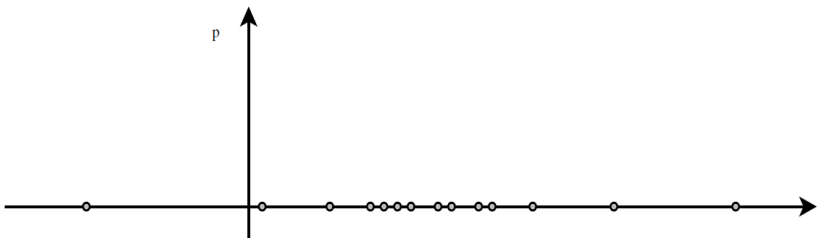
If the model is a 1^{st} order polynomial, as in the first picture, it results in poor performance because it does not fit the input data (**underfitting**).
If the model has a higher order, the trained function will perfectly fit the data, but when the output for an unseen input is required, it will result in poor performance because the function doesn't approximate correctly the real function. Hence it is not able to generalize (**overfitting**).

1.3 Maximum Likelihood Approach

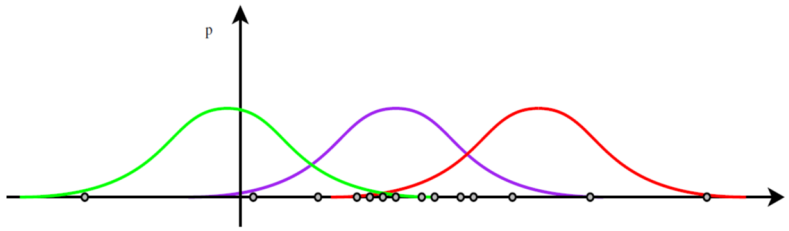
Suppose we observe some i.i.d. samples coming from a Gaussian distribution with known σ^2.

$$x_1, x_2, ..., x_K \sim \mathcal{N}(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Given three different possible models for these samples, we have to determine which one is the best to



represent the data.
In order to do that, we can use the maximum likelihood approach, in which the chosen parameters are the one that maximize the data probability.
Intuitively we should not choose the green or the red distribution because there are too many data points in a place where the probability with that distribution is very low. So, we need to choose the model to which the data are more likely.



1.4 Maximum Likelihood Estimation (MLE)

Suppose $\theta = (\theta_1, \theta_2, ..., \theta_n)^T$ is a vector of parameters and we want to find the MLE for θ assuming known form for $p(Data|\theta)$:

1. Write the likelihood $L = P(Data|\theta)$ for the data.

$$L(\mu) = p(x_1, x_2, ..., x_N|\mu, \sigma^2) = \prod_{n=1}^{N} p(x_n|\mu, \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}$$

2. Take the logarithm of the likelihood $\mathcal{L} = \ln P(Data|\theta)$ (optional but useful because it transforms the products into sums)

$$\begin{aligned}
\mathcal{L} &= \ln \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \\
&= \sum_{n=1}^{N} \ln \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}} \\
&= N\left(\ln \frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (x_n - \mu)^2
\end{aligned}$$

3. Obtain $\partial\mathcal{L}/\partial\theta$ using calculus

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mu} &= \frac{\partial}{\partial \mu} N\left(\ln \frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (x_n - \mu)^2 \\
&= -\frac{1}{2\sigma^2} \frac{\partial}{\partial \mu} \sum_{n=1}^{N} (x_n - \mu)^2 \\
&= \frac{1}{2\sigma^2} \sum_{n=1}^{N} 2(x_n - \mu) \\
&= \frac{1}{\sigma^2} \sum_{n=1}^{N} (x_n - \mu)
\end{aligned}$$

4. Solve the unconstrained equations $\partial \mathcal{L}/\partial \theta_i = 0$

$$\begin{aligned}
\frac{1}{\sigma^2}\sum_{n=1}^{N}(x_n - \mu) &= 0 \\
\sum_{n=1}^{N}(x_n - \mu) &= 0 \\
\sum_{n=1}^{N} x_n &= \sum_{n=1}^{N} \mu \\
\sum_{n=1}^{N} x_n &= N\mu \\
\mu^{\text{MLE}} &= \frac{1}{N}\sum_{n=1}^{N} x_n
\end{aligned}$$

So, the best MLE of the mean of a gaussian distribution is the mean of the samples. But this is not valid for other estimators that can be possibly used.

Chapter 2

The Perceptron

1956 is considered the date of birth of Artificial Intelligence, during the Darthmouth Conference,. At that time computers and algorithms were doing precisely what the programmer programs them and where able to do arithmetics very fast, but at the same time they were fragile, because noise was a problem and there was the need to deal with uncertainty. So, the necessity to model the way in which humans think and learn to replicate it into machines, started to develop.
The imitation of the human brain has been considered as a solution because it is:

- able to learn
- fault tolerant
- highly parallel
- able to deal with noise

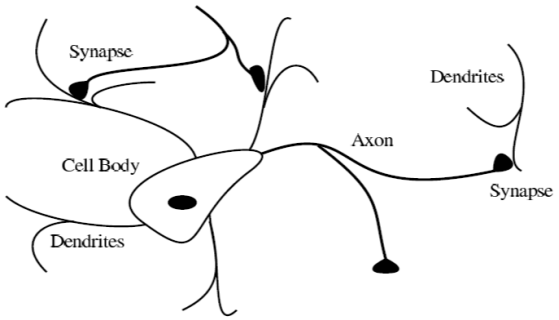
2.1 The biological neuron

In the brain we have:

- 10^{11} neurons
- 10^4 synapses per neuron

The computational model of the brain is:

- distributed among simple units called neurons
- intrinsically parallel
- redundant and thus fault tolerant: cells die and are born but computation is (almost) not affected

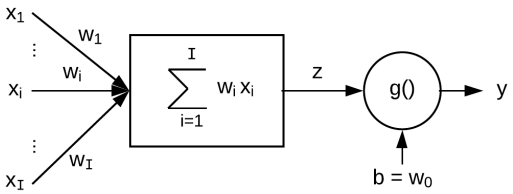


The neuron has a main connection with the rest of the network, called axon, and other minor connections called dendrites.
Neurons are connected with each other through the synapses, that are the terminations of the dendrites and neurons exchange charge through them. There are two types of synapses:

- Inhibitory synapse that makes the receiver neuron less likely to activate
- Excitatory synapse that makes the receiver neuron more likely to activate

The impact on the receiver is different between synapses, so they have different weights.
A neuron cumulates charges and when it exceeds a threshold level for the membrane potential it gets released through the axon (firing).
The computation is, hence, a highly non-linear phenomenon, because the neuron is almost inactive until it suddenly spikes.

2.2 The artificial neuron



To model the neuron we need to replicate its cumulation effect and firing effect.
In this simple model of a neuron we can identify:

- the (synaptic) weights w_i
- the activation value $\sum_i w_i x_i$
- the activation threshold or bias b
- the activation function $g(.)$

The output of the perceptron depends on if the value of the activation value exceeds the bias, based on the activation function:

$$y = g(z) = g\left(\sum_{1}^{I} w_i x_i - b\right)$$

We can define $b \doteq -w_0 \cdot 1$ and $x_0 = 1$ (notice that the indices of the inputs start from 1)

$$y = g\left(\sum_{1}^{I} w_i x_i + w_0 \cdot 1\right) = g\left(\sum_{1}^{I} w_i x_i + w_0 x_0\right) = g\left(\sum_{0}^{I} w_i x_i\right)$$

This version is not only simpler, but it also expresses the fact that also the bias can be learned, because it is a weight equivalent to the others.
It is also possible to express the formula as vectors:

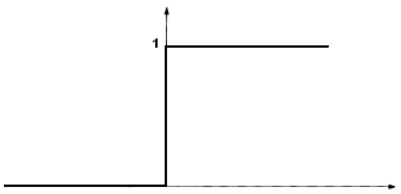
$$y = g(\mathbf{w}^T \mathbf{x})$$

where $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_I \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_I \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_I \end{bmatrix}$

Typical activation functions are:

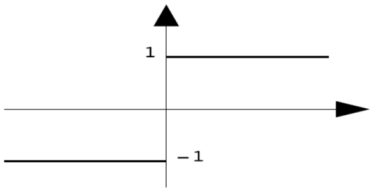
Step function

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$



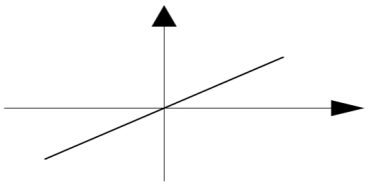
Sign function

$$g(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0 \end{cases}$$



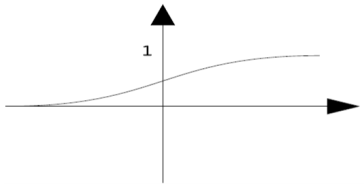
Linear function

$$g(z) = z$$



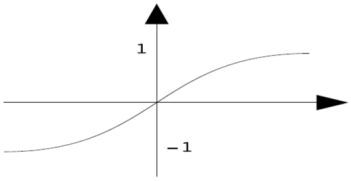
Sigmoid function

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



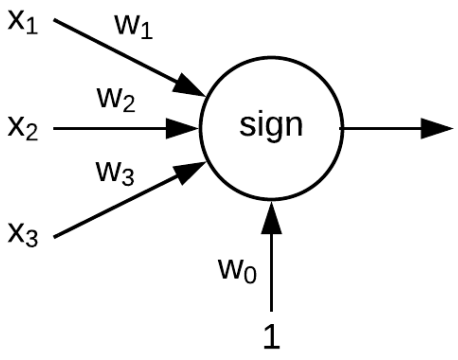
Hyperbolic tangent

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



2.3 The Perceptron

The first model of neuron proposed was the Perceptron.



It uses the sign function, which is usually preferred to the step function because it is easier to train. To perform learning we would like to change weights by providing examples.
We can notice that by tuning the values of the weights we can obtain the behavior of different logic operators:

2.3.1 Perceptron as a logic AND

with $w_1 = \frac{3}{2}$, $w_2 = 1$ and $w_0 = -2$:

x_0	x_1	x_2	y
1	0	0	$sign(-2) = -1$
1	0	1	$sign(-1) = -1$
1	1	0	$sign(-\frac{1}{2}) = -1$
1	1	1	$sign(\frac{1}{2}) = 1$

2.3.2 Perceptron as a logic OR

with $w_1 = 1$, $w_2 = 1$ and $w_0 = -\frac{1}{2}$:

x_0	x_1	x_2	y
1	0	0	$sign(-\frac{1}{2}) = -1$
1	0	1	$sign(\frac{1}{2}) = 1$
1	1	0	$sign(\frac{1}{2}) = 1$
1	1	1	$sign(\frac{3}{2}) = 1$

How to manage when the inputs values are different from just 0 and 1? Just change the weight accordingly to obtain the desired result.
But having found ways of programming by setting weights doesn't mean that it is easy to do, Hebb Learning tries to overcome this problem.

2.3.3 Hebbian Learning (1949)

Donald Olding Hebb studied the learning mechanism of the neural cell. The learning mechanism is the one that makes the triggering behavior of the neural cell change the more they experience some input and output.
He observed that the strength of a synapse increases according to the simultaneous activation of the relative input and the desired target.
Hebbian learning can be summarized by the following rule:

$$w_i^{k+1} = w_i^k + \Delta w_i$$
$$\Delta w_i = \eta \cdot t \cdot x_i$$

where:

- η: learning rate
- x_i: the i^{th} perceptron's input
- t: the desired output

Notice that if t_i and x_i have the same sign (they are simultaneously activated), the increment is positive (the strength increases).
The steps for the learning process can be summarized as follows:

1. Start from random weights
2. Show one record (i.e. set x_i values to the ones of the record)

   2.a. If wrong $w_i = w_i + \Delta w_i$
3. Go to next record (point 2)

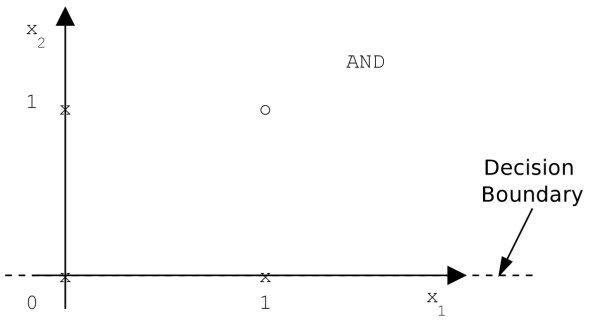
Hebbian Learning of the AND operator

Let's make an example of trying to learn the AND operator through the Perceptron.

x_0	x_1	x_2	y
1	0	0	-1
1	0	1	-1
1	1	0	-1
1	1	1	1

Suppose we start from a random initialization of $w_0 = 0, w_1 = 0$ and $w_2 = 1$, using a learning rate $\eta = \frac{1}{2}$.

The crosses represent the -1 values, while the only circle corresponds the only case in which the AND



operator returns a positive value.

The decision boundary of the Perceptron can be computed as:

$$\begin{aligned} 0 &= w_0 + x_1 \cdot w_1 + x_2 \cdot w_2 \\ x_2 \cdot w_2 &= -x_1 \cdot w_1 - w_0 \\ x_2 &= -\frac{w_1}{w_2} \cdot x_1 - \frac{w_0}{w_2} \end{aligned}$$

So it can be represented as a line (in a 2-dimensional space, in 3-D is a plane and an hyperplane in spaces with more than 3 dimensions). Based on the current weights the decision boundary corresponds to the x_1 axis.

Now for each epoch the four records in the table will be provided to the Perceptron, which will update the weights until the decision boundary will be able to separate the negative values from the positive one.

Epoch 1

- Record 1: $sign(w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2) = sign(0 \cdot 1 + 0 \cdot 0 + 1 \cdot 0) = 0 \neq -1$ (update)
  - $w_0 = w_0 + \eta \cdot t \cdot x_0 = 0 + \frac{1}{2} \cdot (-1) \cdot 1 = -\frac{1}{2}$
  - $w_1 = w_1 + \eta \cdot t \cdot x_1 = 0 + \frac{1}{2} \cdot (-1) \cdot 0 = 0$
  - $w_2 = w_2 + \eta \cdot t \cdot x_2 = 1 + \frac{1}{2} \cdot (-1) \cdot 0 = 1$
- Record 2: $sign(-\frac{1}{2} \cdot 1 + 0 \cdot 0 + 1 \cdot 1) = 1 \neq -1$ (update)
  - $w_0 = -\frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = -1$
  - $w_1 = 0 + \frac{1}{2} \cdot (-1) \cdot 0 = 0$
  - $w_2 = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$
- Record 3: $sign(-1 \cdot 1 + 0 \cdot 1 + \frac{1}{2} \cdot 0) = -1$ (don't update)
- Record 4: $sign(-1 \cdot 1 + 0 \cdot 1 + \frac{1}{2} \cdot 1) = -1 \neq 1$ (update)
  - $w_0 = -1 + \frac{1}{2} \cdot 1 \cdot 1 = -\frac{1}{2}$

- $w_1 = 0 + \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$
- $w_2 = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$

Not all the records have been predicted correctly, so a new epoch is computed.

Epoch 2

- Record 1: $sign(-\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 + 1 \cdot 0) = -1$ (don't update)
- Record 2: $sign(-\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 + 1 \cdot 1) = \frac{1}{2} \neq -1$ (update)
  - $w_0 = -\frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = -1$
  - $w_1 = \frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 0 = \frac{1}{2}$
  - $w_2 = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$
- Record 3: $sign(-1 \cdot 1 + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0) = -1$ (don't update)
- Record 4: $sign(-1 \cdot 1 + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1) = 0 \neq 1$ (update)
  - $w_0 = -1 + \frac{1}{2} \cdot 1 \cdot 1 = -\frac{1}{2}$
  - $w_1 = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$
  - $w_2 = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$

Epoch 3

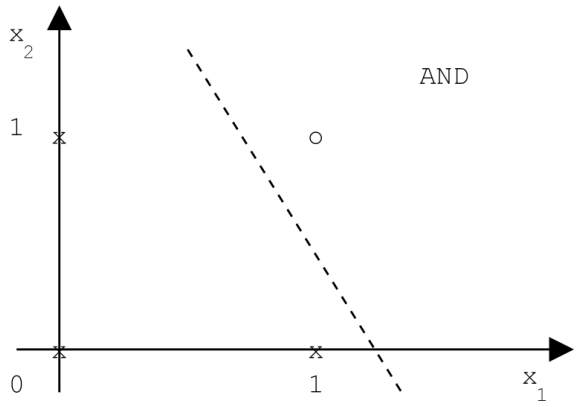
- Record 1: $sign(-\frac{1}{2} \cdot 1 + 1 \cdot 0 + 1 \cdot 0) = -1$ (don't update)
- Record 2: $sign(-\frac{1}{2} \cdot 1 + 1 \cdot 0 + 1 \cdot 1) = 1 \neq -1$ (update)
  - $w_0 = -\frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = -1$
  - $w_1 = 1 + \frac{1}{2} \cdot (-1) \cdot 0 = 1$
  - $w_2 = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$
- Record 3: $sign(-1 \cdot 1 + 1 \cdot 1 + \frac{1}{2} \cdot 0) = 0 \neq -1$ (update)
  - $w_0 = -1 + \frac{1}{2} \cdot (-1) \cdot 1 = -\frac{3}{2}$
  - $w_1 = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$
  - $w_2 = \frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 0 = \frac{1}{2}$
- Record 4: $sign(-\frac{3}{2} \cdot 1 + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1) = -1 \neq 1$ (update)
  - $w_0 = -\frac{3}{2} + \frac{1}{2} \cdot 1 \cdot 1 = -1$
  - $w_1 = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$
  - $w_2 = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$

...

Epoch 8

- Record 1: $sign(-\frac{3}{2} \cdot 1 + \frac{3}{2} \cdot 0 + \frac{3}{2} \cdot 0) = -1$ (don't update)
- Record 2: $sign(-\frac{3}{2} \cdot 1 + \frac{3}{2} \cdot 0 + \frac{3}{2} \cdot 1) = 0 \neq -1$ (update)
  - $w_0 = -\frac{3}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = -2$
  - $w_1 = \frac{3}{2} + \frac{1}{2} \cdot (-1) \cdot 0 = \frac{3}{2}$
  - $w_2 = \frac{3}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = 1$
- Record 3: $sign(-2 \cdot 1 + \frac{3}{2} \cdot 1 + 1 \cdot 0) = -1$ (don't update)
- Record 4: $sign(-2 \cdot 1 + \frac{3}{2} \cdot 1 + 1 \cdot 1) = 1$ (don't update)

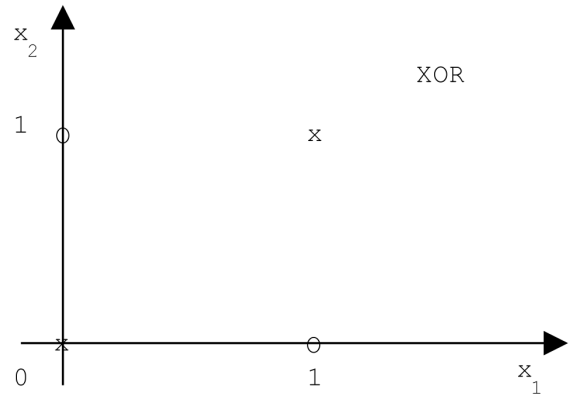
Since the last three records return a correct prediction, to check if the learning converged, it is sufficient to check only Record 1: $sign(-2 \cdot 1 + \frac{3}{2} \cdot 0 + 1 \cdot 0) = -1$.
All the records are correctly predicted, thus the learning stops.



It may happen to obtain different weights as a solution, that's because the same points may be separable by more than one decision boundary and the final result depend on the initialization values.
It can be proved that if the learning converges, it also converges in a finite number of epochs, but does it always converge?
The answer is no, particularly in two situations:

1. if η is large, it is less likely to converge, because the change of weights is too big. So, when the weights get updated because of a record, the Perceptron's output may change too much, causing another record to become wrongly predicted.

2. if it is not possible to learn that specific truth table. One example of non linearly separable problem is the XOR problem:



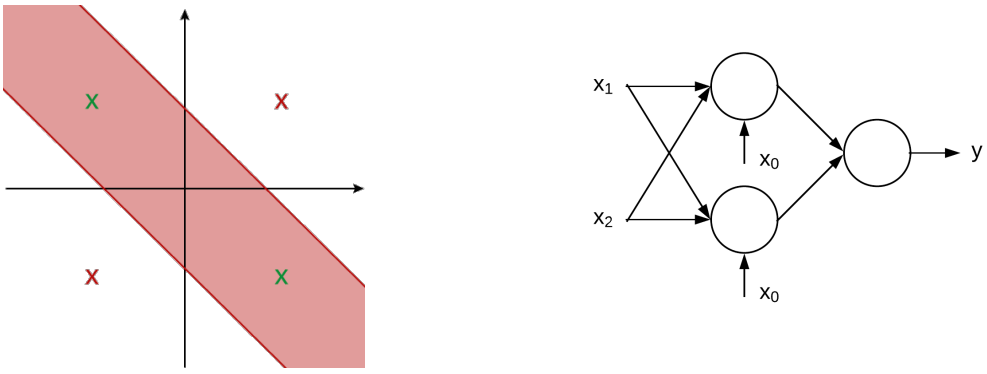
This function cannot be learned by a Perceptron, even if we increase the number of dimensions.

Before going further some definitions are necessary:

- Online learning: every time a record is evaluated, the weights are modified and the learning process continues with the new model. It is less used now but it may be useful in case of data coming as a stream, and the model is updated as soon as a new record arrives.

- Batch learning: it uses the same model for all the data and updates the model only from one epoch to the next one. The advantage is that this approach averages the contributions of the single records, obtaining a smoother behavior and a better convergence. Nowadays it is very common to use miniBatch, that are chunks of data instead of the whole set, because the latter does not fit in memory.

It is be possible to solve the XOR problem by combining neurons.



The neuron on the top in the first layer has the decision boundary which is higher in the plot: all the points above it are 0 and all the points below it are 1; the neurons on the bottom in the first layer has the bottom decision boundary: all the points above it are 1 and all the points below it are 0; the last neurons is simply an AND operator of the outputs of the previous two. This model is called **Multilayer Perceptron**.
Notice that the process of learning in this case is not trivial: the update of weights is $\Delta w = \eta \cdot t \cdot x_i$, so for the weight from the first neuron to the third one the value x_i comes as output of the first neuron and it is not known; for the weight from the first input to the first neuron, t is not known.

Chapter 3

Feedforward Neural Networks

Artificial Neural Network: a set of neurons connected according to a topology.

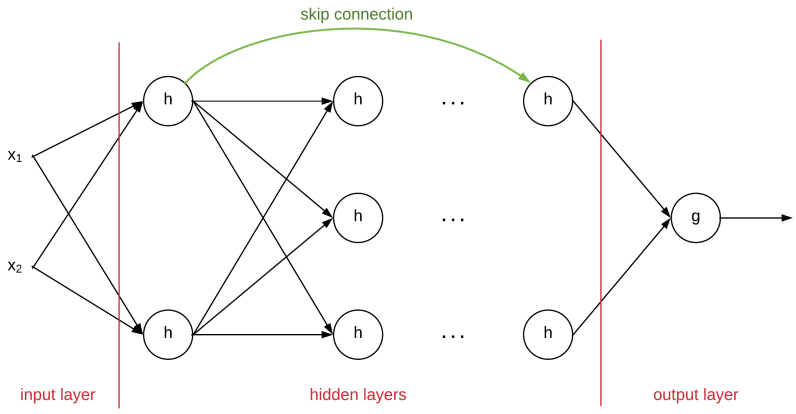
Layer: neurons at the same distance from the input neurons.

Input layer: layer of neurons that receives as input the data to process.

Output layer: layer of neurons that gives the final result of the network.

Hidden layer: layer of neurons that processes data from other neurons to be processed by other neurons.

An artificial neural network is a non-linear model characterized by the number of neurons, their topology, activation functions and the values of synaptic weights and biases.



Where I is the number of input neurons, J is the number of hidden neurons (in case of single hidden layer) and O is the number of output neurons.
In an artificial neural network, information can flow along different paths, but always forward.
Usually each neuron of one layer is connected to all the neurons of the following layer and only to those (fully connected FFNN). The act of setting a weight to zero is equivalent to removing the connection.
It is also possible to have skip connections, which are connections that end on a neuron that is not in the immediately subsequent layer. But this connection can be modeled by a fully connected NN by adding a new neurons between the two that just passes the value unchanged. Hence, a fully connected NN is a superset of a NN with skip connections.
Both the input layer and the output layer depend on the problem, while the shape of the hidden layer, in terms of number of neurons per hidden layer and the number of hidden layers, is a design parameter.

With the step and sign activation functions it is difficult to relate the output of the Perceptron to its input, this means that the same output (e.g. 1) can derive from an infinite number of inputs (e.g.

from 0 to ∞). One of the solutions is to introduce the continuous approximation of these two functions, which are the already seen sigmoid and hyperbolic tangent (tanh). In 1991 it has been proven that a FFNN, with S-shaped activation functions and only 1 hidden layer, can approximate any non-linear continuous function with arbitrary precision, provided enough neurons (universal approximator).
There are some situations in which a universal approximator is not enough, e.g. a classification problem, which may be discontinuous.

3.1 Regression

In a regression problem the goal is to approximate a target function t given a finite set of N observations. Hence $y \in \mathbb{R}$, but the output of the sigmoid or tanh $\notin \mathbb{R}$. For this reason $g()$ is usually a linear activation function $z = \sum_{i=0}^{I} w_i x_i$. It is still a non-linear model because it is a linear combination of non-linear functions.

3.2 Classification

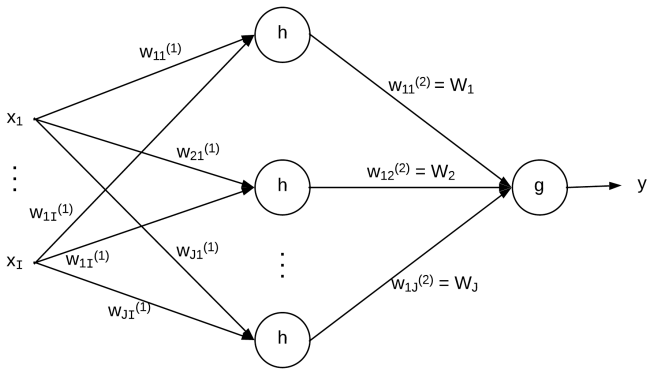
In a classification problem the goal is to separate two (or more) classes $y \in \{\Omega_0, \Omega_1\}$, according to the posterior probability. If we want to encode the classs in [+1, -1] we can use tanh, while if we want to encode them in [0, 1] we can use the sigmoid.
If the classes are more than one ($y \in \{\Omega_0, \Omega_1, \Omega_2\}$) they should not be encoded in [1, 2, 3] because it will imply and order and a distance between classes, which are usually meaningless concepts in a classification problem. In this case One Hot Encoding can be used: there are three binary outputs which refer respectively to the first, second and third class; the sample belongs to the i^{th} class if the i^{th} output is equal to 1 and it does not belong to it if the output is 0.
With outputs between 0 and 1, they can be interpreted as the probability of that sample to belong to that class. But in this case the outputs have to sum up to 1. Multiple solutions have been proposed, e.g. take the output according to a majority vote, or normalize and take the maximum value(but it involves the use of a non continuous function), or use a softmax function as output activation function:

$$g = \frac{e^{z_j}}{\sum_{j=0}^{J} e^{z_j}}$$

The softmax function has the following properties:

- sums up to 1
- is continuous and differentiable, so it is always possible to compute the derivative of the output with respect to the input
- "winner takes all": the exponential makes the biggest value even bigger

3.3 Learning in multilayer perceptrons



The output of the network is

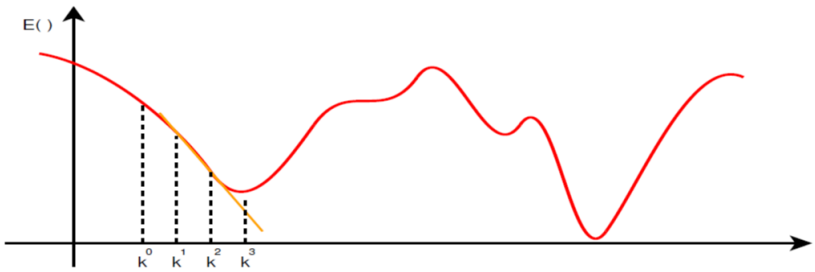
$$y = g\left(\sum_{j=0}^{J} W_j \cdot h\left(\sum_{i=0}^{I} w_{ji} \cdot x_i\right)\right)$$

and it has to be as close as possible to the target function: $y_n \approx t_n$.
Hence, we want to minimize the error function:

$$E = \sum_{n=0}^{N} (t_n - y_n)^2 = \sum_{n} \left(t_n - g\left(\sum_{j=0}^{J} W_j \cdot h\left(\sum_{i=0}^{I} w_{ji} \cdot x_i\right)\right)\right)^2$$

3.3.1 Backpropagation and Gradient Descent

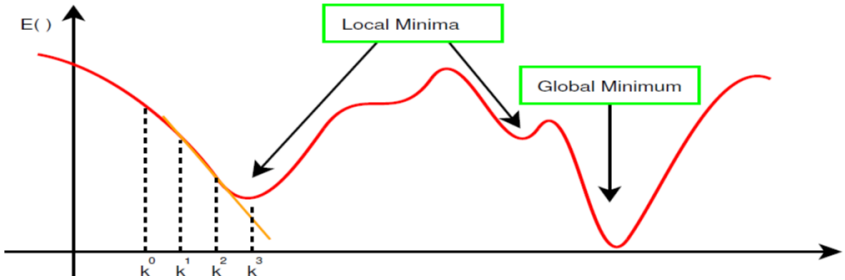
Sometimes it is not easy to minimize or maximize analytically a function, so we need to apply a numerical approach: the gradient descent.
It performs the following steps:



1. Pick up a possible solution $\mathbf{w}^0$ (at random)
2. Compute the function derivative $\frac{\partial E}{\partial \mathbf{w}}\big|_k$
3. Update the solution
$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \left.\frac{\partial E}{\partial \mathbf{w}}\right|_k$$
4. Repeat 2 and 3 until convergence

The possible problems (with corresponding solution) are:

- local minima → multiple random initializations or use momentum
- slow convergence → use second order derivative
- no convergence at all → reduce η during minimization



In case of gradient descent with momentum the update rule is:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \left.\frac{\partial E}{\partial \mathbf{w}}\right|_k - \alpha \left.\frac{\partial E}{\partial \mathbf{w}}\right|_{k-1}$$

The process of learning in a Feedforward Neural Network is called **Backpropagation**, which is the use of gradient descent to iteratively minimize the network error.

Gradient for the output weights

Let's now compute the derivative of the error function w.r.t. a specific weight from the hidden neurons to the output one. Using a specific weight as reference will help easing the reading process instead of using a generic weight W_j. For example let's consider the weight W_7.

$$\begin{aligned}\frac{\partial E}{\partial W_7} &= \frac{\partial \sum_n^N (t_n - y_n)^2}{\partial W_7} \\ &= \sum_n^N \frac{\partial (t_n - y_n)^2}{\partial W_7} \\ &= \sum_n^N 2(t_n - y_n)\frac{\partial(-y_n)}{W_7} \\ &= -2\sum_n^N (t_n - y_n)\frac{\partial y_n}{\partial W_7}\end{aligned}$$

Then, we compute the derivative of the output w.r.t. the weight.

$$\begin{aligned}\frac{\partial y_n}{\partial W_7} &= \frac{\partial}{\partial W_7} g\left(\sum_j^J W_j h_j \left(\sum_i^I w_{ji} x_i\right)\right) \\ &= g'(\cdot)\frac{\partial}{\partial W_7}\left(\sum_j^J W_j h_j \left(\sum_i^I w_{ji} x_i\right)\right) \\ &= g'(\cdot) h_7 \left(\sum_i^I w_{7i} x_i\right)\end{aligned}$$

For the last equality note that:

$$\frac{\partial}{\partial W_7}\left(\sum_j^J W_j h_j \left(\sum_{i=0}^I w_{ji} x_i\right)\right) =$$

$$= \frac{\partial}{\partial W_7}\left(W_0 h_0\left(\sum_i w_{0i}x_i\right) + ... + W_7 h_7\left(\sum_i w_{7i}x_i\right) + ... + W_J h_J\left(\sum_i w_{Ji}x_i\right)\right) =$$

$$= h_7\left(\sum_i^I w_{7i} x_i\right)$$

Putting everything together we can finally say that, for a generic weight W_j:

$$\frac{\partial E}{\partial W_j} = -2\sum_n^N (t_n - y_n) g'(\cdot) h_j \left(\sum_i^I w_{ji} x_i\right)$$

Gradient for the input weights

Let's repeat the process for the weights w_{ji}, considering in this case w_{35}.

$$\begin{aligned}\frac{\partial E}{\partial w_{35}} &= \frac{\partial \sum_n^N (t_n - y_n)^2}{\partial w_{35}} \\ &= -2\sum_n^N (t_n - y_n)\frac{\partial y_n}{\partial w_{35}}\end{aligned}$$

$$\frac{\partial y_n}{\partial w_{35}} = \frac{\partial}{\partial w_{35}} g\left(\sum_j^J W_j h_j \left(\sum_i^I w_{ji} x_i\right)\right)$$
$$= g'(\cdot)\frac{\partial}{\partial w_{35}} \sum_j^J W_j h_j \left(\sum_i^I w_{ji} x_i\right)$$
$$= g'(\cdot) W_3 h_3' \left(\sum_i^I w_{3i} x_i\right) x_5$$

In this case, too, the last passage is explained by:

$$\frac{\partial}{\partial w_{35}} \sum_j^J w_j h_j \left(\sum_i w_{ji} x_i\right) =$$
$$= \frac{\partial}{\partial w_{35}} \left(W_0 h_0 \left(\sum_i^I w_{0i} x_i\right) + ... + W_3 h_3 \left(\sum_i^I w_{3i} x_i\right) + ... + W_J h_J \left(\sum_i^I w_{Ji} x_i\right)\right) =$$
$$= W_3 \frac{\partial}{\partial w_{35}} h_3 \left(\sum_i^I w_{3i} x_i\right) =$$
$$= W_3 h_3' \left(\sum_i^I w_{3i} x_i\right) \frac{\partial}{\partial w_{35}} \sum_i^I w_{3i} x_i =$$
$$= W_3 h_3' \left(\sum_i^I w_{3i} x_I\right) x_5$$

Finally, the derivative of the error with respect to the input weights is:

$$\frac{\partial E}{\partial w_{ji}} = -2 \sum_n^N (t_n - y_n) g'(\cdot) W_j h_j' \left(\sum_{i'}^I w_{ji'} x_{i'}\right) x_i$$

So, according to the gradient descent rule the weights are updated as follows:

$$W_j^{k+1} = W_j^k - \eta \left.\frac{\partial E}{\partial W_j}\right|_{W_j^k}$$

$$w_{ji}^{k+1} = w_{ji}^k - \eta \left.\frac{\partial E}{\partial w_{ji}}\right|_{w_{ij}^k}$$

The backpropagation process is divided into two phases:

- forward pass: if computes the neurons' output values, which are expressed as a non-linear function of the input signal and the weights associated to that neuron.
- backward pass: starts at the end and recursively applies the chain rule to compute the weights all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.

3.4 Error function in regression

Now one may ask, why using $E = \sum_n (t_n - y_n)^2$ as an error function? Is it always correct to use it?
If the samples come from a target function as $t_n = f(x) + \epsilon$ and we want to approximate it using our model, as $t_n = y_n + \epsilon$, suppose that $\epsilon \sim \mathcal{N}(0, \sigma^2)$, hence $t_n \sim \mathcal{N}(y, \sigma^2)$.
This means that in this case, the machine learning problem reduces to estimating the mean of the gaussian from the set of samples.
Similarly to Section 1.4, in which we have shown that the maximum likelihood estimator of the mean of a gaussian function is the mean of the sample, now we will find the best set of weights to estimate the

mean of the gaussian function.
The likelihood function is

$$L(\mathbf{w}) = p(t_1, ..., t_N|\mathbf{w}) = \prod_n p(t_n|\mathbf{w}) = \prod_n \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(t_n - y_n)^2}$$

Maximizing the log likelihood

$$\begin{aligned}
\text{argmax}_{\mathbf{w}} \ln(L) &= \text{argmax}_{\mathbf{w}} \sum_n \left(\ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}(t_n - y_n)^2 \right) \\
&= \text{argmax}_{\mathbf{w}} - \sum_n \frac{1}{2\sigma^2}(t_n - y_n)^2 \\
&= \text{argmin}_{\mathbf{w}} \frac{1}{2\sigma^2} \sum_n (t_n - y_n)^2 \\
&= \text{argmin}_{\mathbf{w}} \sum_n (t_n - y_n)^2
\end{aligned}$$

This shows that the estimating the maximum likelihood is equivalent to minimizing the mean square error.

3.5 Error function in classification

There are some cases in which $t_n \neq f(x) + \epsilon$ or $\epsilon \nsim \mathcal{N}(0, \sigma^2)$.
For example when $t_n = f(x) \cdot \epsilon$ or in the case of classification, for which $t_n \in \{0, 1\}$.
In classification $t_n \sim Be(y_n)$, so $p(t_n|\mathbf{x}) = y_n^{t_n}(1 - y_n)^{1-t_n}$. Hence,

$$L(\mathbf{w}) = p(t_1, ..., t_N|\mathbf{w}) = \prod_n p(t_n|\mathbf{w}) = \prod_n y_n^{t_n}(1 - y_n)^{1-t_n}$$

$$\mathcal{L}(\mathbf{w}) = \log(L(\mathbf{w})) = \sum_n t_n \log y_n + (1 - t_n) \log(1 - y_n)$$

$$\begin{aligned}
\text{argmax}_{\mathbf{w}} \mathcal{L}(\mathbf{w}) &= \text{argmin}_{\mathbf{w}} - \mathcal{L} \\
&= \text{argmin}_{\mathbf{w}} - \left(\sum_n t_n \log y_n + (1 - t_n) \log (1 - y_n) \right)
\end{aligned}$$

This expression is called **cross entropy** and its minimization is equivalent to the minimization of the Kullback-Leibler divergence of the network output and the target distribution.

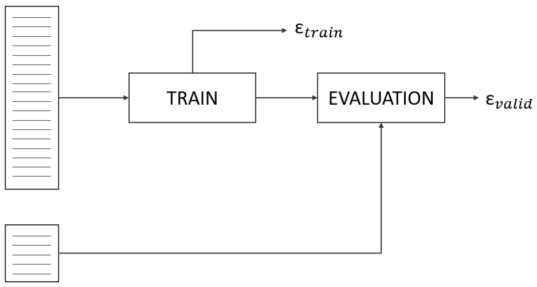
3.6 Generalization

A model is said to be able to generalize if after learning on a training set is able to produce good results also on new unseen samples.
Overfitting, as explained in Section 1.2.2, is when the model fits perfectly the training set, but it does not generalize on new samples, so it just memorizes the training set and its noise.
To evaluate generalization/overfitting, the initial dataset is split into training set, validation set and test set. The model is actually trained only using the first set; the second one is used to estimate how well the current model predicts unseen data, and to select the model with the lowest validation error; the last set is used as a final evaluation of the model. It is very important that the model does not come in contact with the test set during the training phase, otherwise the measure of generalization would be highly overestimated.
This methods reduces the size of the dataset used for training and in case of small datasets it may be a problem. Alternatives that allow to train the model on a set that has the size of the training set and the validation set combined are:

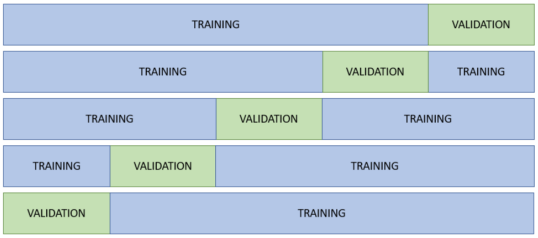
- k-fold crossvalidation: divide the whole dataset into training set and test set and further split the training set into k sets of the same size. Then training and validation are repeated k times: each time one of the k sets is used for validation while the other k-1 are used for training. The error estimation is averaged between all the k runs. In this way every set is used for training at least once.

- Leave-One-Out method: k-fold crossvalidation with k=N. So the model is trained using all but one sample and that sample is then used for validation. This is repeated for each sample of the training set. The N errors are then averaged.

This two methods require more computation because the model has to be trained respectively k and N times, which can often be prohibitive.



(a) Validation

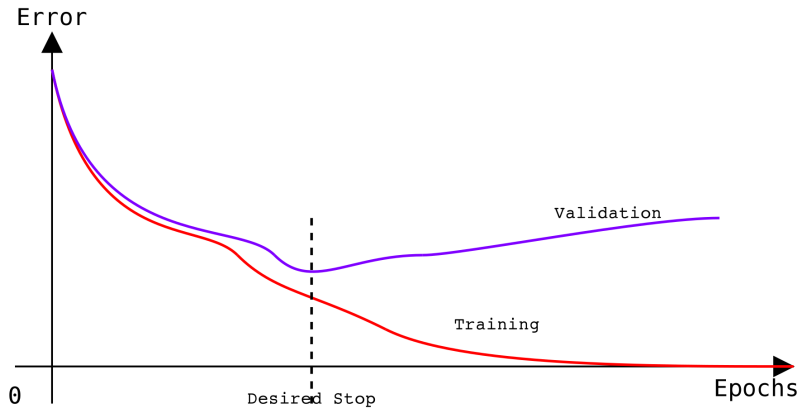


(b) 5-fold crossvalidation

There are multiple ways to avoid overfitting in Neural Networks.

3.6.1 Early stopping

It consists in stopping the learning process before the model starts to fit the noise in the data.



This method is used to find out how many neurons (the complexity) are needed in the hidden layer by comparing different topologies w.r.t. the validation error.
Multiple strategies are possible for the stopping criteria:

- train all epochs and choose manually the epoch in which to stop

- train until the validation error increases: it is more efficient but the error is noisy, so it may not be monotonic, resulting in a premature stop

- train until the validation error doesn't decrease for N epochs

- ...

3.6.2 Weight decay (regularization)

Up to now, Maximum Likelihood Estimation has been used to obtain the value of the weights:

$$\hat{\mathbf{w}} = \text{argmax}_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})$$

This approach is called frequentist, which relies exclusively on the experience. It is opposed to the bayesian approach, in which there is an initial belief on the values of the weights (called prior distribution) and the experience is used to change it.
It is called Maximum A-Posteriori (MAP):

$$\hat{\mathbf{w}} = \text{argmax}_{\mathbf{w}} P(\mathbf{w}|\mathcal{D}) \propto \text{argmax}_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(\mathbf{w})$$

where $P(\mathbf{w}|\mathcal{D})$ is called posterior distribution, $P(\mathcal{D}|\mathbf{w})$ is the likelihood and $P(\mathbf{w})$ is the prior distribution. As the data goes to infinite the two approaches give the same result.

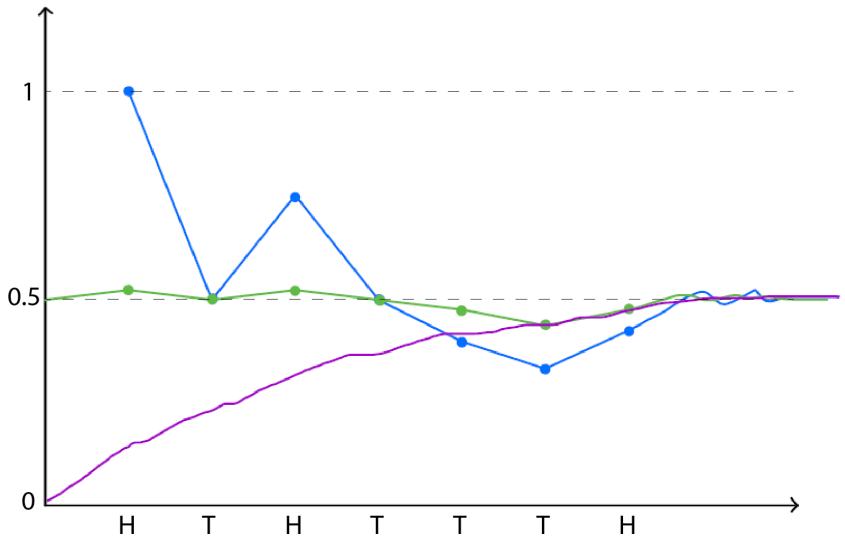


Figure 3.2: Frequentist and bayesian approach

Figure 3.2 is a qualitative example and it has the purpose of showing the intuition around the difference between the frequentist and bayesian approach. It shows the predicted probability of obtaining head of a regular coin.
The blue line represents the probability modeled with a frequentist approach. The probability before flipping the coin the first time is not defined, but as soon as the first flip is performed, the probability is defined to 1 for head (because the first toss gave head) and then it is updated for each flip, based on the experience.
The green line represents the probability modeled with a bayesian approach that has the correct prior distribution. We can see that each flip slightly modifies the probability value.
The purple line shoes the bayesian approach with a wrong prior distribution. The probability is believed to be 0, but thanks to the experience acquired by tossing the coin, the posterior distribution is iteratively updated until it converges to the right value. The prior distribution is difficult to determine, because it requires domain knowledge. Conjugate priors are often used to get an "easy" posterior distribution.

Usually the derivatives of an overfitting model are greater in absolute value than the ones of a model that is able to generalize, which are smoother. This means that the weights of the former are bigger.
If we choose a prior distribution as follows:

$$\mathbf{w} \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2)$$

$$\begin{aligned}\hat{\mathbf{w}} &= \text{argmax}_{\mathbf{w}} P(\mathbf{w}|\mathcal{D}) \\ &= \text{argmax}_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(\mathbf{w}) \\ &= \text{argmax}_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(w_1, w_2, ..., w_m)\end{aligned}$$

If we assume the weights independent and identically distributed

$$\begin{aligned}\hat{\mathbf{w}} &= \text{argmax}_{\mathbf{w}} P(\mathcal{D}|\mathbf{w})P(w_1)P(w_2)...P(w_m) \\ &= \text{argmax}_{\mathbf{w}} \prod_n \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(t_n - y_n)^2} \cdot \prod_m \frac{1}{\sqrt{2\pi}\sigma_m} e^{-\frac{1}{2\sigma_m^2}(w_m - 0)^2}\end{aligned}$$

We can take the logarithm.

$$\begin{aligned}\hat{\mathbf{w}} &= \text{argmax}_{\mathbf{w}} -\frac{1}{2\sigma^2}\sum_n (t_n - y_n)^2 - \frac{1}{2\sigma_{\mathbf{w}}^2}\sum_m w_m^2 \\ &= \text{argmin}_{\mathbf{w}} \sum_n (t_n - y_n)^2 + \frac{\sigma^2}{\sigma_{\mathbf{w}}^2}\sum_m w_m^2\end{aligned}$$

Setting $\gamma = \frac{\sigma^2}{\sigma_{\mathbf{w}}^2}$ we obtain

$$\hat{\mathbf{w}} = \text{argmin}_{\mathbf{w}} \sum_n (t_n - y_n)^2 + \gamma \sum_m w_m^2$$

The first term is the usual minimization of the mean square error; the second term is a penalization based on the size of the weights and the coefficient γ. In this way, smoother solutions will be preferred. This type of regularization is called Ridge. Notice that if $\gamma = 0$ there is no penalization and the model is prone to overfitting; if $\gamma \to \infty$ only the weights are minimized so the output of the model is 0 for any input. To find the right value of γ it is necessary to train multiple models with different γ values and select the one that has the minimum validation error.
There is another type of regularization, called Lasso:

$$\hat{\mathbf{w}} = \text{argmin}_{\mathbf{w}} \sum_n (t_n - y_n)^2 + \gamma \sum_m |w_m|$$

The effect is similar because it tends to penalize models with big weights, but instead of reducing the size of all weights, it tends to put some of them directly to zero, obtaining a sparse model.

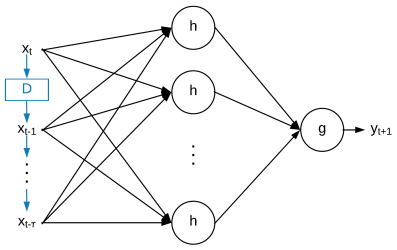
Chapter 4

Recurrent Neural Networks

Suppose we want to predict the next day stock market average $y(t+1)$ based on the previous days' economic indicators $x(t), x(t-1)$,... There are mainly three approaches using neural networks:

- Standard Feedforward NN: do a regression from $x(t)$ to $y(t+1)$, but it is not possible to capture dependencies on earlier values of x.
- Feedforward with delayed input: replicate a finite number of earlier values of x and apply regression from $[x(t), x(t-1), ..., x(t-q)]$, but there is the problem that the value of q is not known.
- Recurrent Neural Networks: add a new unit b to the hidden layer and a new input unit $c(t)$ to represent the value of b at time $(t-1)$. b thus can summarize information from earlier values of x arbitrarily distant in time.

4.1 Feedforward with delayed input



x(t)	y(t)
1	7
2	6
3	7
4	8
5	9
6	5
...	...

x(t-2)	x(t-1)	x(t)	y(t+1)
1	2	3	8
2	3	4	9
3	4	5	5
4	5	6	...
5	6	...	
6	...		
...			

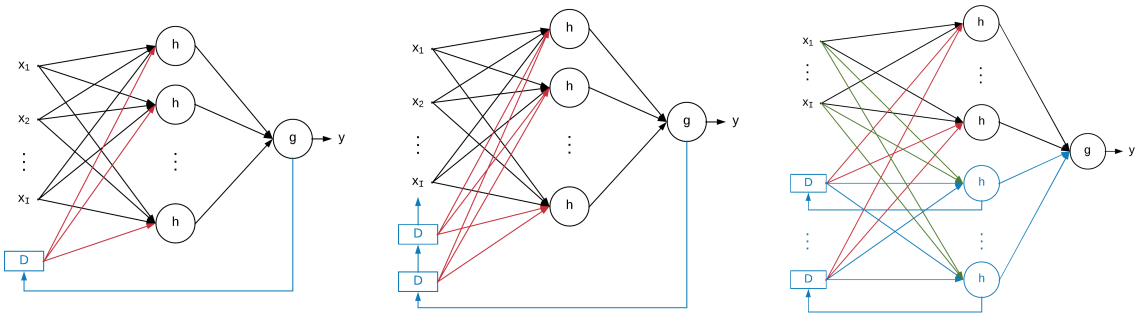
The weights of the network can be learned building a delayed dataset from the original one and then using the standard backpropagation learning algorithm

$$y(t+1) = g\left(\sum_j^J W_j \cdot h\left(\sum_{i=0}^T w_{ji} \cdot x(t-i)\right)\right)$$

The main issue is that the network does not contain a state, so it does not consider the history of the system.

4.2 Recurrent Artificial Neural Networks

There are different types of RNN

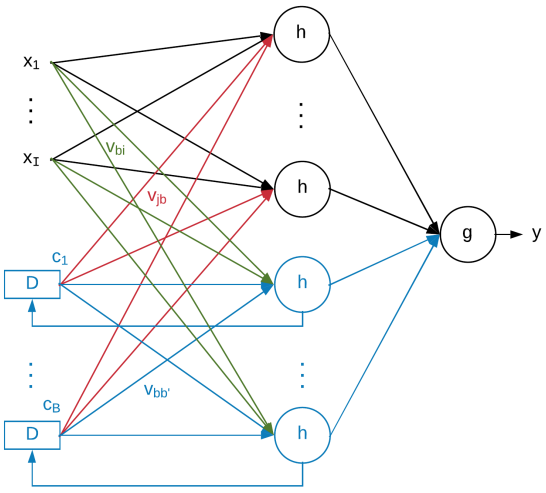


1. Recur the output as input
2. Recur the output (and its previous values) as input
3. Recur the internal (hidden) state of the network as input

Let's now analyze the last topology, called Elman topology.

4.3 Elman Networks

In this type of Recurrent Neural Network, a context network is added to act as a state.



In the context network B new neurons are added in the hidden layer and their output is not only connected to the output layer, but it is also delayed and connected again to the hidden layer (in the following time step).
Each delayed output at time step t has a coefficient associated to it c_b^t. Each connection from neuron b' to the other neuron b of the following time step is weighted by $v_{bb'}$.
Hence, the network's output is now

$$y^t = g\left(\sum_j^J W_j \cdot h\left(\sum_b^B v_{jb} c_b^{t-1} + \sum_i^I w_{ji} x_i^t\right) + \sum_b^B W_b h\left(\sum_{b'}^B v_{bb'} c_{b'}^{t-1} + \sum_i^I v_{bi} x_i^t\right)\right)$$

$$c_b^t = h\left(\sum_{b'}^B v_{bb'} \cdot c_{b'}^{t-1} + \sum_i^I v_{bi} \cdot x_i^t\right)$$

In this network the most difficult part to train is the context because it is not a feedforward NN. The learning technique used is called Backpropagation Through Time.

4.3.1 Backpropagation Through Time

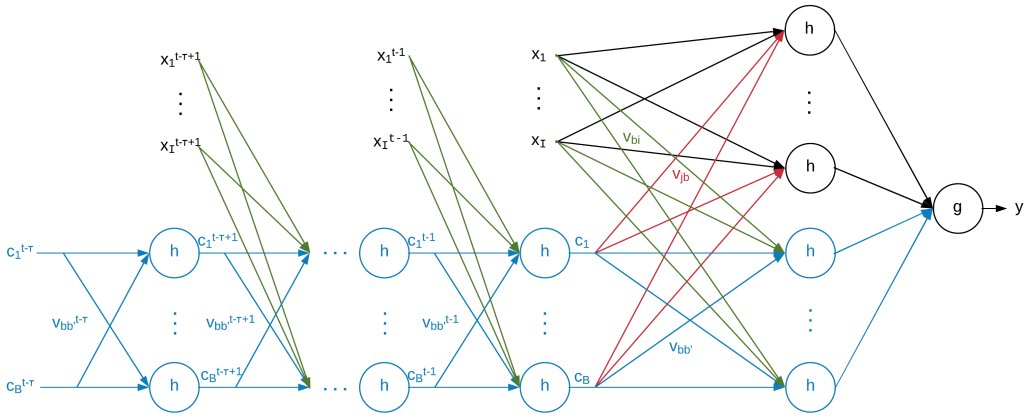
It is an extension of the standard Backpropagation algorithm. It performs:

1. Unfolding of the RNN for U time steps, obtaining a FF NN.
2. Backpropagation is applied to the new network

Network unfolding

Let N denote a recurrent network required to learn a temporal task, starting from time $t-u$ all the way up to time t. Let N^* denote the feedforward NN that results from the unfolding of N:

1. For each time step in the interval $(t-u, t]$ the network N^* has a layer containing k neurons, where k is the number of neurons contained in N.
2. In every layer of N^* there is a copy of each neuron in N.
3. For each time-step $\tau \in (t-u, t]$ the synaptic weight from neuron i in layer τ to neuron j in layer $\tau+1$ of N^* is a copy of the synaptic connection from neuron i to neuron j in N.



Backpropagation

All the weights are trained with gradient descent, so considering a generic time step τ.

$$v_{bi}^{t-\tau} = v_{bi}^{t-\tau} - \eta \frac{\partial E}{\partial v_{bi}^{t-\tau}}$$

$$v_{bb'}^{t-\tau} = v_{bb'}^{t-\tau} - \eta \frac{\partial E}{\partial v_{bb'}^{t-\tau}}$$

$$v_{jb}^{t-\tau} = v_{jb}^{t-\tau} - \eta \frac{\partial E}{\partial v_{jb}^{t-\tau}}$$

Applying the usual method of update will lead to having different values for for the same weights in different time steps. This is not convenient because they actually are the same weights, so after updating the weights according to the gradient, they are averaged.

$$v_{bi} = \frac{1}{U+1} \sum_{\tau=0}^{U} v_{bi}^{t-\tau}$$

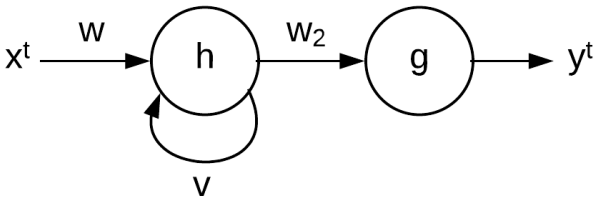
$$v_{bb'} = \frac{1}{U+1} \sum_{\tau=0}^{U} v_{bb'}^{t-\tau}$$

$$v_{jb} = \frac{1}{U+1} \sum_{\tau=0}^{U} v_{jb}^{t-\tau}$$

Instead of computing the backpropagation and averaging the weights, a possible alternative is to update the weights with the average gradient. The result will be the same, but the process will be slightly more efficient.

4.3.2 Vanishing Gradient

Unfortunately, the context network is not able to capture long term sequences, because of the vanishing gradient effect.
To explain it, a simplified version of the RNN will be used.



Where

$$c^t = h(vc^{t-1} + wx^t)$$

$$y = g(w_2 c^t)$$

To train v and w we need to unroll the network, and derive E w.r.t. them.

$$\begin{aligned}\frac{\partial E}{\partial w} &= \sum_{t=0}^{N} \frac{\partial E^t}{\partial w} \\ &= \sum_t \frac{\partial E^t}{\partial y} \cdot \frac{\partial y}{\partial c^t} \cdot \frac{\partial c^t}{\partial w} \\ &= \sum_t \frac{\partial E^t}{\partial y} \cdot \frac{\partial y}{\partial c^t} \cdot \frac{\partial c^t}{\partial c^{t-\tau}} \cdot \frac{\partial c^{t-\tau}}{\partial w}\end{aligned}$$

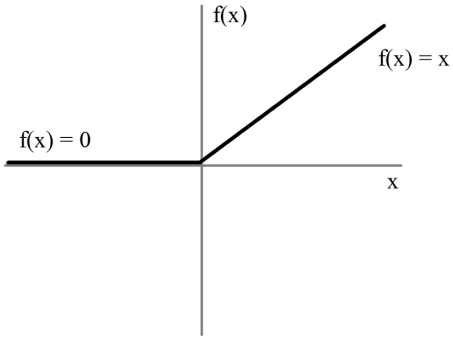
$$\frac{\partial c^t}{\partial c^{t-\tau}} = \prod_{i=t}^{t-\tau+1} \frac{\partial c^i}{\partial c^{i-1}} = \prod_{i=t}^{t-\tau+1} \frac{\partial h(vc^{i-1} + wx)}{\partial c^{i-1}} = \prod_{i=t}^{t-\tau+1} vh'(vc^{i-1} + wx) \simeq v^\tau h'^\tau$$

The equality is approximated because h' is not always the same.
Notice that h is often a sigmoid or an hyperbolic tangent, which have a derivative smaller than 1. By repeatedly multiplying them, after some steps, $\frac{\partial h^t}{\partial h^{t-\tau}}$ is almost zero and hence the gradient is almost zero as well. This means that the more the learning process goes into the past, the more the gradient will be close to zero, independently by the error (vanishing gradient).
Based on the same principles, if $h' > 1$ the gradient will increase dramatically, resulting in an gradient explosion that will affect the network's ability to learn. The following sections will describe two possible solutions for the vanishing gradient problem.

4.3.3 Rectified Linear Unit (ReLU)

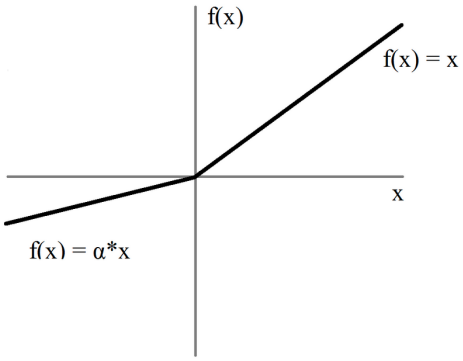
The ReLU function can be defined as follows:

$$R(x) = \max(0, x)$$



It has a derivative equal to 1 for $x > 0$ and equal to 0 for $x < 0$. In the former case, the gradient is backpropagated as it is, while in the latter no gradient is backpropagated from that point backwards. The use of ReLU has a disadvantage: if the weights learned are such that the x is negative for the entire domain of inputs, the neuron never learns. This is known as the dying ReLU problem.
The dying ReLU problem can be tackled by using a modified version of the activation function, called Leaky ReLU.

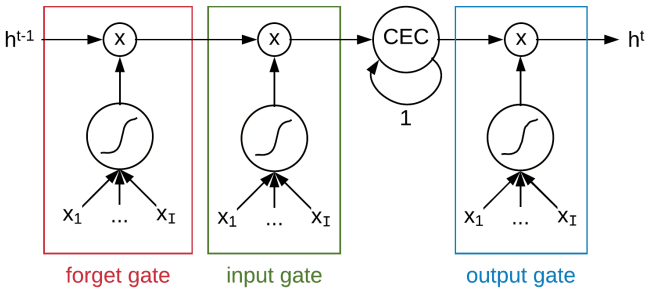
$$LR(x) = \max(0.01x, x)$$



The vanishing gradient can be also caused by the value of v. By putting $v = 1$ we obtain a type of neuron called Constant Error Carousel (CEC), which does not help though, because the fixed value 1 will have the effect of just accumulating the input, without the possibility of writing in input.

4.4 Long short-term memory (LSTM)

LSTM networks are special types of RNN that are able to learn very long sequences.
All recurrent neural networks have the form of a chain of repeating modules of neural network, but instead of having a single neural network layer, it has multiple, each with a specific role. In the following part a brief description of all of them is presented.



The horizontal line is the one through which the state is modified.
The first gate is called *forget gate*. The result of the gate is in $[0, 1]$ and its multiplication with the state leads, in the former case, to the complete deletion of the previous value or, in the latter case, the state to be left unchanged.
Then there is the *input* (or *write*) *gate*, because it controls the value to be added to the state cell depending on the input $(x_1, ..., x_I)$.
For this reason it is called forget gate: the value between 0 to 1 of the gate controls how much information of the previous value of the state can enter into the cell.
Then the state is passed through a CEC, and then through the last gate, the *output* (or *read*) *gate*, that controls what to output from the memory.
LSTM has been proved to be very effective to avoid the vanishing gradient problem. It can learn what should be written or read into memory and when it should. All the weights can be learned using backpropagation.

Chapter 5

Deep Learning

5.1 Hierarchical feature learning

5.1.1 "Classical" learning

The "classical" way of learning is usually by using a set of hand-crafted features and then applying a trainable classifier. One of the problems of this approach is that the selection of the features may require the need of an expert or may be difficult to perform and hence be suboptimal.
Some improvements have been brought by semi-supervised learning techniques, which use unsupervised learning to find features to describe unlabeled data. A possible example is to use clustering to find possible groups and for each labeled sample, a feature is the group it belongs to.
It is sometimes very difficult to find even unlabeled data of the same type of the labeled ones. Self-taught learning uses unlabeled data with a domain different from the labeled ones because it can still improve the performance of the classifier. For example, regarding the task of recognizing a specific type of object in images, it can still be useful to train on unlabeled data coming from random images because they still have features, such as lines and edges, that can be useful for the detection of the specific object. When the labeled dataset is small this technique can give a significant improvement.

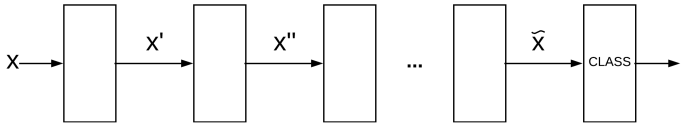
5.1.2 Deep Learning

Deep Learning is a class of machine learning algorithms in which the feature extractor is trained too, to learn the best representation of the data, so that the features are tailored to the specific problem it is required to solve.
It uses multiple hidden layers to extract low, mid and high-level features and then there is another layer with the role of classifier. Deep Learning assumes that it is possible to learn a hierarchy of descriptors with increasing abstraction.
A deep model have many parameters, so it is prone to overfitting. Some ways to cope with that are:

- regularization of backpropagation
- dropout technique for training, in which at each iteration some of the neurons are randomly switched off.
- having a huge amount of data (often difficult or expensive)

5.2 Neural Networks Autoencoders

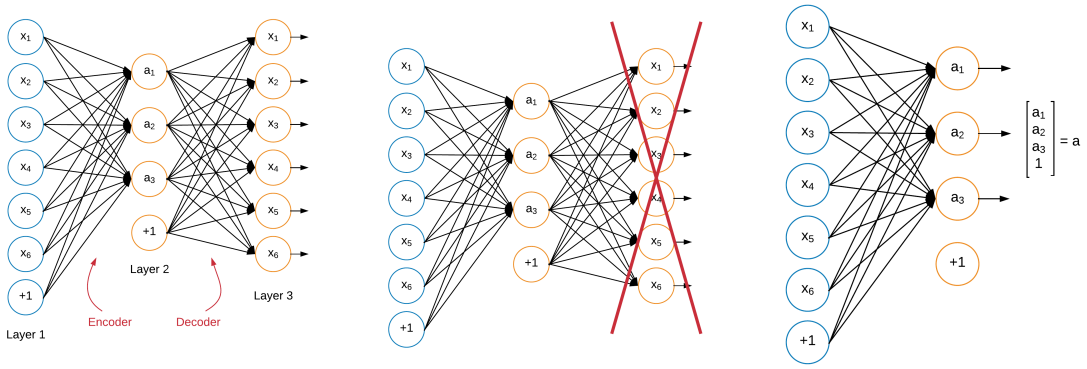
In Deep Learning we have some hidden layers that have the role of learning features, but because they perform unsupervised learning, they don't have labels.



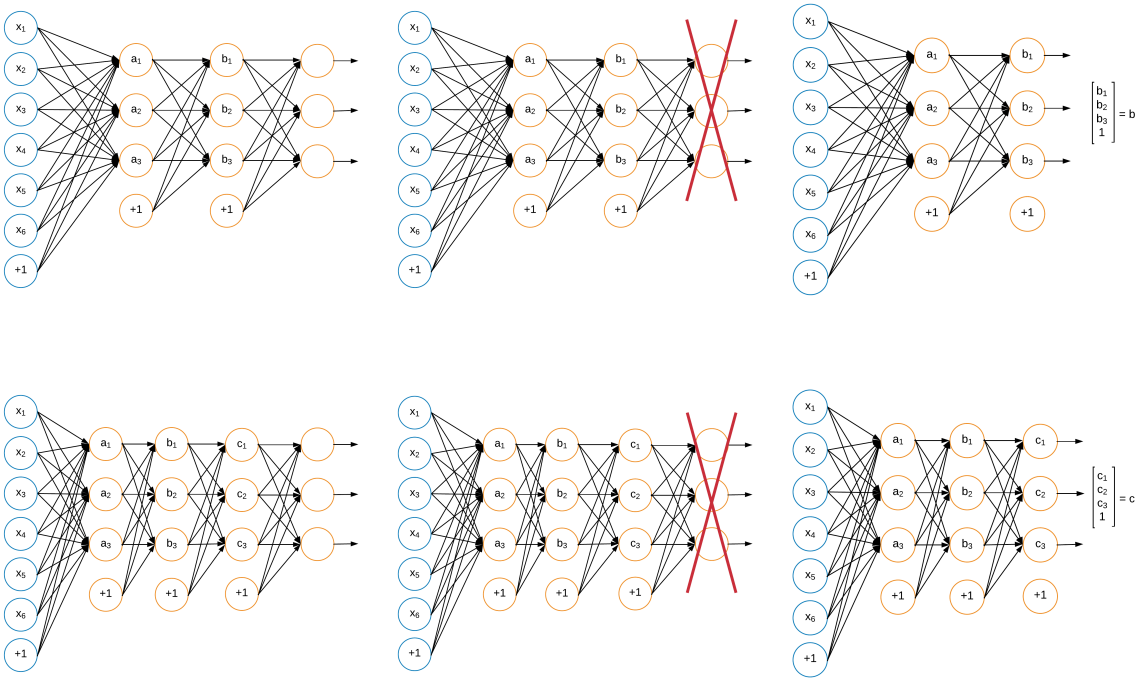
To deal with the problem, autoencoders are used, so that the network is trained to output the input itself (i.e. to learn the identity function). This is done by projecting in a smaller input space containing all the information. Thus, the second layer has a limited number of units (compressed representation) and has to be sparse (sparse representation).
The autoencoder is trained from a dataset of unlabeled training samples $x_1, x_2, ...$ using standard back-propagation through

$$\min_{\theta} \underbrace{||h_\theta(x_i) - y_i||^2}_{\text{reconstruction error term}} + \underbrace{\lambda \sum_i |a_i|}_{L_1 \text{ sparsity term}}$$

Then the third layer is removed and we obtain a new representation of the input.



The process is repeated multiple times, training each layer individually to avoid vanishing gradient, until the representation to feed to the supervised learning algorithm is obtained.



The autoencoders technique is used in many different fields, while the following technique, called Convolutional Neural Network, is mainly focused on image recognition.

5.3 Convolutional Neural Networks

5.3.1 Multi Layer Perceptrons

For this type of neural networks the case of recognizing an hand-written letter will be considered.
The prior art was to use multiple layers of perceptrons as it has been previously seen. The image is in a square of pixels, and there is one input neuron for each pixel (plus the bias); the number of output neurons is equal to the number of possible letters.
Multi Layer Perceptrons have some drawbacks: they are hard to train, sensitive to noise and have little, if any, invariance to shifting, scaling and other forms of distortion.

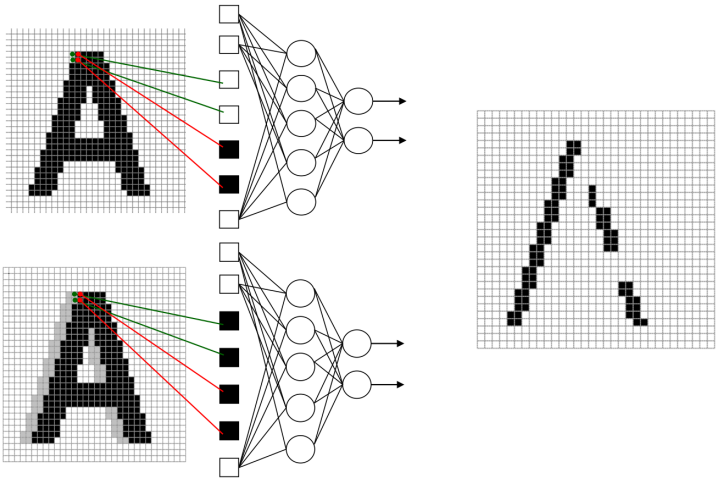
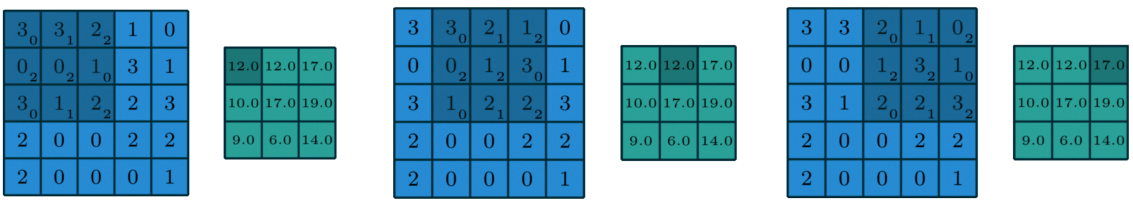


Figure 5.4: Example of shifting

A possible solution is to learn the encoding for every possible transformation, but it is clear that this approach is not the best possible, because what should be important is the configuration of pixels in an area, not where they are actually placed.

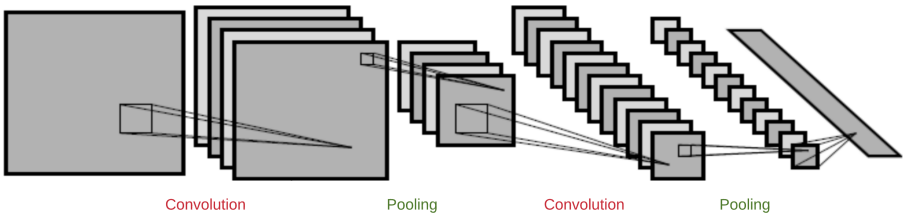
5.3.2 Convolutional Neural Networks

It is possible to represent an black and white image as a two-dimensional matrix in which each element contains the intensity of the pixel.
A very well known operation in image processing is called convolution, in which a kernel (a small matrix of weights) slides over the input feature map. At each kernel position, elementwise product is computed between the kernel and the overlapped input subset. The result is summed up and constitute the output feature map, which is another image.

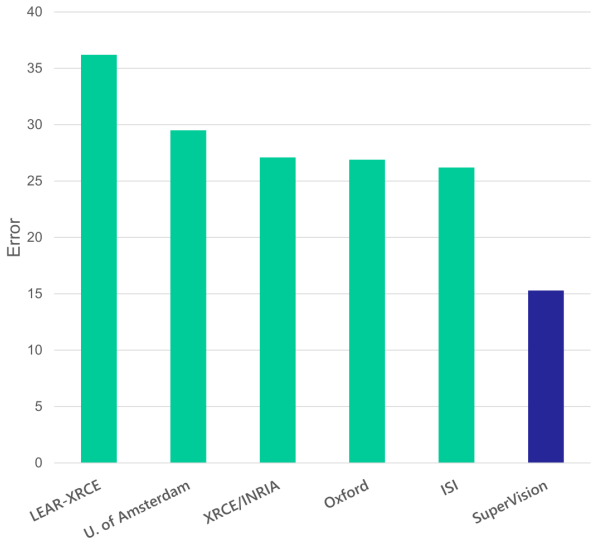


By using specific kernels we can obtain specific effects, such as an image that highlights the intensity of the edges. The idea behind CNN is to learn the kernel weights to obtain interesting features needed by the classifier. The task is feasible because the product function is derivable.

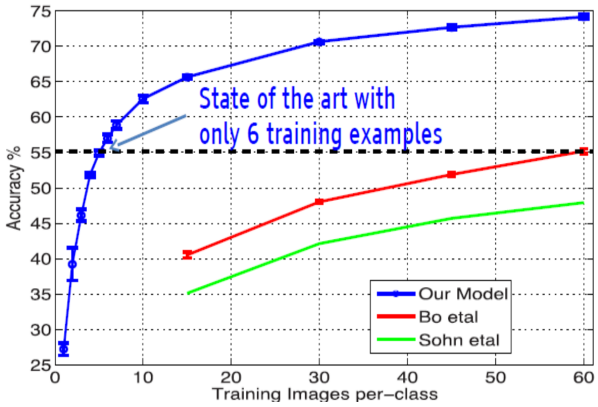
Pooling layers are inserted between convolution layers to reduce the number of parameters and hence control overfitting. The pooling layer usually performs the *max* operation over a square of 2x2 pixels, obtaining a scaled down image which keeps the convolution effect.
By iteratively applying convolution and pooling we obtain a set of very small images that are summarized by a vector that has a certain degree of shift/distortion invariance.



The translation and distortion independence is achieved through the pooling operation, because by reducing dramatically the size of the representation, images with shifted pixels will result in similar reduced representations.
But there is no way to incorporate rotation independence, which has to be learned through data augmentation, as previously explained for the Multi Layer Perceptrons. Data augmentation is performed also with zooming, noise adding, sharpening, fading, different resolution, etc.
All the weights can be trained with standard backpropagation, but given the many neurons in the network, the number of parameters is often huge.
This architecture has been shown to be very effective with respect to the rival techniques.



The huge number of parameters requires a huge dataset. But after training, the neural network has learned features to recognize not all possible objects but a significant amount of them. So, it is possible to use the same feature extractor for different recognition tasks: the only part that has to be trained again is the classifier at the end of the network, strongly reducing the computational effort.
In fact, it has been shown that by reusing the trained feature extraction network and training only the last layer, it is sufficient to reach the state of the art accuracy with only 6 samples per class.


State of the art with
only 6 training examples
Our Model
Bo etal
Sohn etal
Accuracy %
Training Images per−class
75
70
65
60
55
50
45
40
35
30
25
0
10
20
30
40
50
60