# 17
# Modeling Neural Networks

This chapter is an introduction to the world of deep learning, whose methods make it possible to achieve *state-of-the-art* performance in many classification and regression fields often considered extremely difficult to manage (such as image segmentation, automatic translation, voice synthesis, and so on). The goal is to provide the reader with the basic instruments to understand the structure of a fully connected neural network using Keras (employing modern techniques to speed up the training process and prevent overfitting).

In particular, the topics covered in the chapter are as follows:

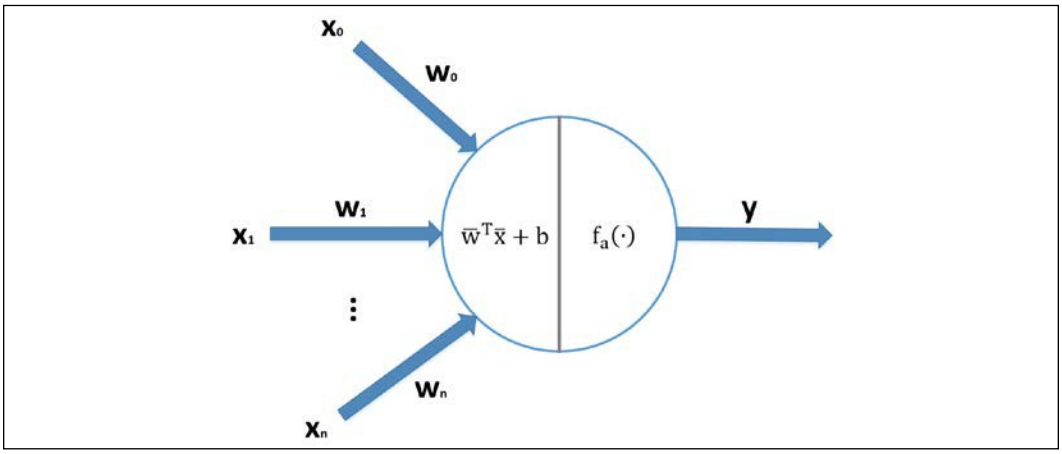
- The structure of a basic artificial neuron
- Perceptrons, linear classifiers, and their limitations
- Multilayer perceptrons with the most important activation functions (such as ReLU)
- Back-propagation algorithms based on the **stochastic gradient descent** (**SGD**) optimization method

Let's start this exploration with a formal definition of the computational unit that characterizes every neural network, the artificial neuron.

## The basic artificial neuron

The building block of a neural network is an abstraction of a biological neuron, a quite simplistic but powerful computational unit that was proposed for the first time by F. Rosenblatt in 1957 to make up the simplest neural architecture, called a perceptron, which we are going to analyze in the next section. Contrary to Hebbian learning, which is more biologically plausible but has some strong limitations, the artificial neuron was designed with a pragmatic viewpoint and only its structure is based on a few of the elements that characterize a biological neuron.

However, recent deep learning research activities have unveiled the enormous power of this kind of architecture. Even though there are more complex and specialized computational cells, the basic artificial neuron can be summarized as the conjunction of two blocks, which are clearly shown in the following diagram:




Structure of a generic artificial neuron

The input of a neuron is a real-valued vector $x\bar{x} \in \mathbb{R}^{nn}$, while the output is a scalar $yy \in \mathbb{R}$. The first operation is linear:

$$zz = w\bar{w}^{TT} \cdot x\bar{x} + bb$$

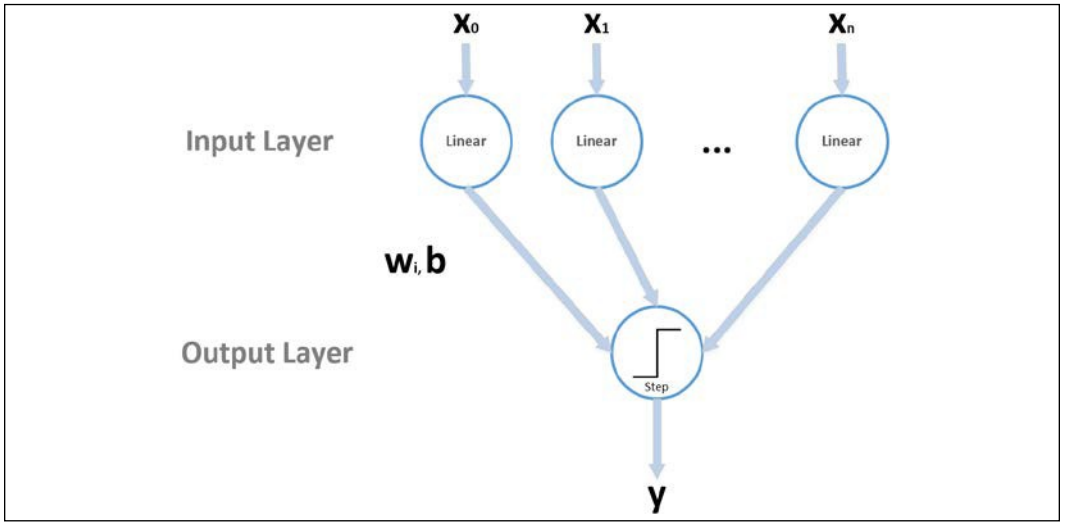
The vector $w\bar{w} \in \mathbb{R}^{nn}$ is called a weight vector (or synaptic weight vector, because, analogously to a biological neuron, it reweights the input values), while the scalar term $bb \in \mathbb{R}$ is a constant called bias. In many cases, it's easier to consider only the weight vector. It's possible to get rid of the bias by adding an extra input feature equal to 1 and a corresponding weight:

$$\bar{xx}^{*} = (xx_1, xx_2, \ldots, xx_{nn}, 1)$$

In this way, the only element that must be learned is the weight vector. The following block is called an activation function, and it's responsible for remapping the input into a different subset. If the function is $f_{aa}(zz) = zz$, the neuron is called linear and the transformation can be omitted. The first experiments were based on linear neurons that are much less powerful than non-linear ones, and this was a reason that led many researchers to consider the perceptron as a failure; but, at the same time, this limitation opened the door for a new architecture instead, that had the chance to show its excellent abilities. Let's now start our analysis with the first neural network ever proposed.

# The perceptron

The perceptron was the name that Frank Rosenblatt gave to the first neural model in 1957. A perceptron is a neural network with a single layer of input linear neurons, followed by an output unit based on the *sign*(*x*) function (alternatively, it's possible to consider a bipolar unit whose output is -1 and 1). The architecture of a perceptron is shown in the following diagram:




Structure of a perceptron

Even though the diagram might appear quite complex, a perceptron can be summarized by the following equation:

$$yy_{ii} = sssssss(\bar{w}^{T} \cdot \bar{xx}_{ii} + bb) \quad wwheeeeee \bar{}ww\bar{xx}_{ii} \in \mathbb{R}^{m} \quad aassaa \; yy_{ii} \in \{0,1\}$$

All the vectors are conventionally column-vectors; therefore, the dot product $w\bar{w}^{TT} \cdot x\bar{x}_{ii}$ transforms the input into a scalar, then the bias is added, and the binary output is obtained using the step function, which outputs 1 when $z > 0$ and 0 otherwise. At this point, a reader could object that the step function is non-linear; however, a non-linearity applied to the output layer is only a filtering operation that has no effect on the actual computation.

Indeed, the output is already decided by the linear block, while the step function is employed only to impose a binary threshold (transforming the continuous output into a discrete one). Moreover, in this analysis, we're considering only single-value outputs (even if there are multi-class variants) because our goal is to show the dynamics and also the limitations, before moving on to more generic architectures that can be used to solve extremely complex problems.

A perceptron can be trained with an online algorithm (even if the dataset is finite) but it's also possible to employ an offline approach that repeats for a fixed number of iterations or until the total error becomes smaller than a predefined threshold. The procedure is based on the squared error loss function (remember that, conventionally, the term loss is applied to single samples, while the term cost refers to the sum/average of every single loss):

$$LL(\bar{xx}_{ii}, yy_{ii}; \bar{ww}, bb) = \frac{1}{2}(\bar{ww}^{TT} \cdot xx_{ii} + bb - yy_{ii})^2 = \frac{1}{2}(ww_1 xx_{ii}^{(1)} + ww_2 xx_{ii}^{(2)} + \cdots + ww_{mm} xx_{ii}^{(mm)} + bb - yy_{ii})$$

When a sample is presented, the output is computed, and if it is wrong, a weight correction is applied (otherwise the step is skipped). For simplicity, we don't consider the bias, as it doesn't affect the procedure. Our goal is to correct the weights so as to minimize the loss. This can be achieved by computing the partial derivatives with respect to $w_i$:

$$\frac{\partial \partial \partial}{\partial \partial ww_{jj}} = (ww_{jj} xx_{ii}^{(jj)} - yy_{ii}) xx_{ii}^{(jj)}$$

Let's suppose that $w^{(0)} = (0, 0)$ (ignoring the bias) and the data point, $x\bar{x} = (1, 1)$, has the label $y = 1$. The perceptron misclassifies the sample because $sssssssss(\bar{ww}^{TT} \cdot \bar{xx}) = 0$ (or -1 if the representation $sssssssss(xx) \in \{-1, 1\}$ is employed). The partial derivatives are both equal to -1; therefore, if we subtract them from the current weights, we obtain $w^{(1)} = (1, 1)$ and now the sample is correctly classified because $sssssssss(\bar{ww}^{TT} \cdot \bar{xx}) = 0$. Therefore, including a learning rate $\eta\eta$, the weight update rule becomes as follows:

$$ww_{ii}^{(tt+1)} = \begin{cases} ww_{ii}^{(tt)} - \eta\eta\,(ww_{jj} xx_{ii}^{(jj)} - yy_{ii}) xx_{ii}^{(jj)} & iiii\ \ ssiissssss(\bar{ww}^{TT} \cdot \bar{xx}) \neq yy_{ii} \\ ww_{ii}^{(tt)} & oooohheeeewwiissee \end{cases}$$

When a sample is misclassified, the weights are corrected proportionally to the difference between the actual linear output and the true label. This is a variant of a learning rule called the delta rule, which represents the first step toward the most famous training algorithm, employed in almost any supervised deep learning scenario (we're going to discuss it in the next sections). The algorithm has been proven to converge to a stable solution in a finite number of states as the dataset is linearly separable. The formal proof is quite tedious and very technical, but those readers who are interested can find it in Minsky M. L., Papert S. A., *Perceptrons*, The MIT Press, 1969.

In this chapter, the role of the learning rate becomes more and more important, in particular when the update is performed after the evaluation of a single sample (as in a perceptron) or a small batch. In this case, a high learning rate (that is, one greater than 1.0) can cause instability in the convergence process because of the magnitude of single corrections.

When working with neural networks, it's usually better to use a small learning rate and repeat the training session for a fixed number of epochs. In this way, single corrections are limited, and they can only become stable if they're confirmed by the majority of samples/batches, which drives the network to converge to an optimal solution. If instead, the correction is the consequence of an outlier, a small learning rate can limit its action, avoiding destabilization of the whole network for just a few noisy samples. We'll discuss this problem further across the next few sections.

Now, we can describe the full perceptron algorithm and close this section with some important considerations:

1. Select a value for the learning rate $\eta\eta$ (such as $\eta\eta = 0.1$). As usual, smaller values allow more precise modifications but increase the computational cost, while larger values speed up the training phase but reduce the learning accuracy.
2. Append a constant column (set equal to 1.0) to the sample vector $X$. Therefore, the resulting vector will be $XX_{bb} \in \mathbb{R}^{MM \times (mm+1)}$.
3. Initialize the weight vector $w\bar{w} \in \mathbb{R}^{nn+1}$ with random values sampled from a normal distribution with a small variance (such as $\sigma\sigma^2 = 0.05$).
4. Set an error threshold $Thr$ (such as $Thr = 0.0001$).
5. Set a maximum number of iterations $N_{max}$.
6. Set $i = 0$.
7. Set $e = 1$.
8. While $i < N_{max}$ and $e > Thr$:
    a. Set $e = 0$.
    b. For $k = 1$ to $M$:
        i Compute the linear output $ll_{kk} = w\bar{w}^{TT} \cdot x\bar{x}_{kk}$ and the thresholded one $t_k = sign(l_k)$.
        ii. If $tt_{kk} \neq yy_{kk}$:
            1. Compute $\Delta ww_{jj} = \eta\eta(ll_{kk} - yy_{kk})xx_{kk}^{(jj)}$.
            2. Update the weight vector.

i. Set $e = e + (l_k - y_k)^2$ (alternatively, it's possible to use the absolute value $e = e + |l_k - y_k|$).

c. Set $ee = {}^{ee}/_{MM}$.

The algorithm is very simple, and you should have noticed an analogy with logistic regression. Indeed, this method is based on a structure that can be considered as a perceptron with a sigmoid output activation function (that outputs a real value that can be considered as a probability). The main difference is the training strategy—in logistic regression, the correction is performed after the evaluation of a cost function based on the negative log likelihood:

$$\begin{aligned} LL(XX, YY; \bar{w}bb) &= -\log \prod_{ii=1}^{MM} pp(yy_{ii} | x\bar{x}_{ii}; \bar{w}bb) \\ &= -\sum_{ii=1}^{MM} \log pp(yy_{ii} | x\bar{x}_{ii}; \bar{w}bb) = \\ &- \sum_{ii=1}^{MM} [yy_{ii} \log \sigma\sigma(\bar{W}^T \cdot x\bar{x}_{ii} + bb) + (1 - yy_{ii}) \log(1 - \sigma\sigma(\bar{W}^T \cdot x\bar{x}_{ii} + bb))] \end{aligned}$$

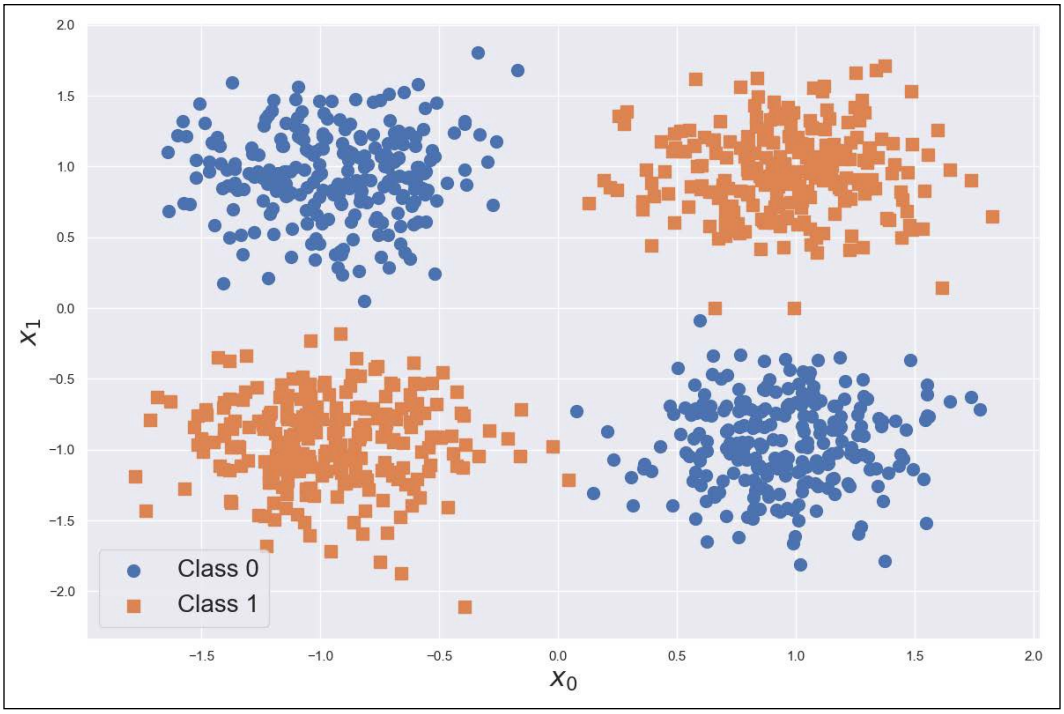
This cost function is the well-known cross-entropy, and in *Chapter 2*, *Loss functions and Regularization*, we showed that minimizing it is equivalent to reducing the Kullback-Leibler divergence between the true and predicted distribution. In almost all deep learning classification tasks, we're going to employ cross-entropy, thanks to its robustness and convexity (convexity is a convergence guarantee in logistic regression, but unfortunately, the property is normally lost in more complex architectures where the cost function is non-convex).

# Example of a Perceptron with scikit-learn

Even if the algorithm is very simple to implement from scratch, I prefer to employ the scikit-learn implementation Perceptron, to focus our attention on the limitations that led to non-linear neural networks. The historical problem that showed the main weakness of the perceptron was based on the XOR dataset. It's easier to build this first, and visualize the structure, before we explain it:

```
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.utils import shuffle

np.random.seed(1000)

nb_samples = 1000
nsb = int(nb_samples / 4)

X = np.zeros((nb_samples, 2))
Y = np.zeros((nb_samples, ))

X[0:nsb, :] = np.random.multivariate_normal(
        [1.0, -1.0], np.diag([0.1, 0.1]), size=nsb)
Y[0:nsb] = 0.0

X[nsb:(2 * nsb), :] = np.random.multivariate_normal(
        [1.0, 1.0], np.diag([0.1, 0.1]), size=nsb)
Y[nsb:(2 * nsb)] = 1.0

X[(2 * nsb):(3 * nsb), :] = \
        np.random.multivariate_normal(
        [-1.0, 1.0], np.diag([0.1, 0.1]), size=nsb)
Y[(2 * nsb):(3 * nsb)] = 0.0

X[(3 * nsb):, :] = np.random.multivariate_normal(
        [-1.0, -1.0], np.diag([0.1, 0.1]), size=nsb)
Y[(3 * nsb):] = 1.0

ss = StandardScaler()
X = ss.fit_transform(X)

X, Y = shuffle(X, Y, random_state=1000)
```

The plot showing the true labels is shown here:




Example of the XOR dataset

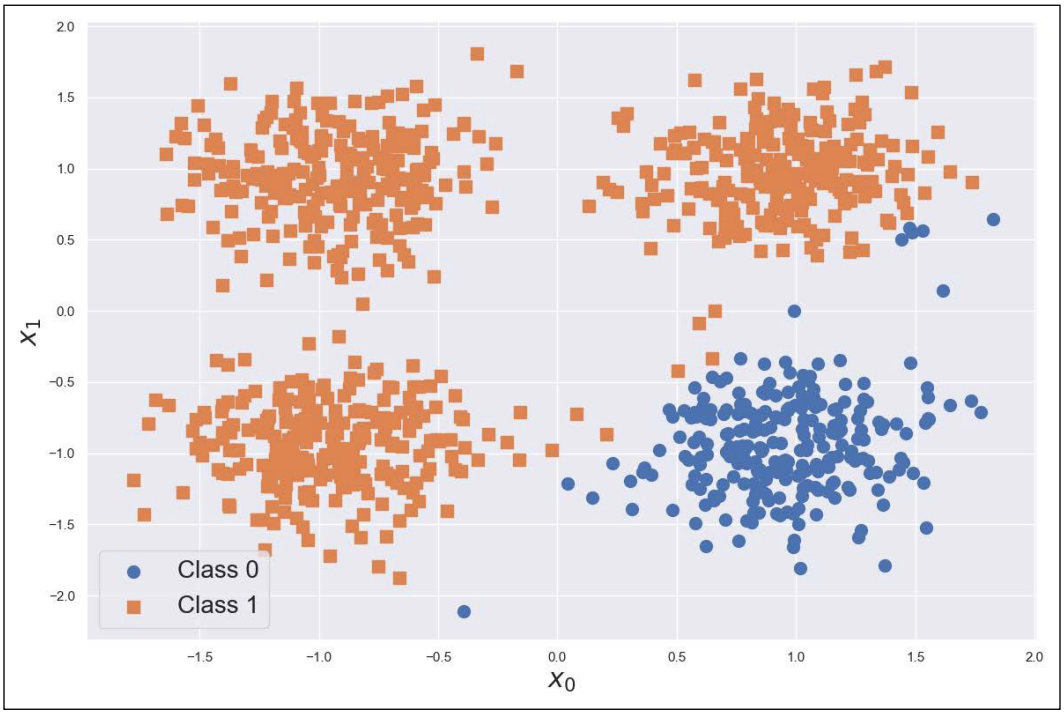
As it's possible to see, the dataset is split into four blocks that are organized as the output of a logical XOR operator. Considering that the separation hypersurface of a two-dimensional perceptron (as well as the separation hypersurface of logistic regression) is a line; it's easy to understand that any possible final configuration can achieve an accuracy that is about 50% (a random guess). To get confirmation, let's try to solve this problem:

```
from sklearn.linear_model import Perceptron
from sklearn.model_selection import cross_val_score

pc = Perceptron(penalty='l2', alpha=0.1,
                    n_jobs=-1, random_state=1000)
print("Perceptron Avg. CV score: {:.3f}".
      format(np.mean(cross_val_score(pc, X, Y, cv=10))))
```

The output of the previous snippet is:

```
Perceptron Avg. CV score: 0.504
```

This value confirms that, in this case, a perceptron is approximately equal to a random guess, therefore, it doesn't offer any advantages and there's no way to overcome this limitation. Conversely, for linearly separable scenarios, a scikit-learn implementation offers the possibility to add a regularization term (see *Chapter 2*, *Loss functions and Regularization*) through the parameter penalty (it can be `'l1'`, `'l2'`, or `'elasticnet'`) to avoid overfitting, induce sparsity, and improve the convergence speed (the strength can be specified using the parameter `alpha`). This is not always necessary, but as the algorithm is offered in a production-ready package, the designers decided to add this feature. Nevertheless, the average cross-validation accuracy is slightly higher than 0.5 (you're invited to test any other possible hyperparameter configuration). The corresponding plot (which can change with different random states or subsequent experiments) is shown here:



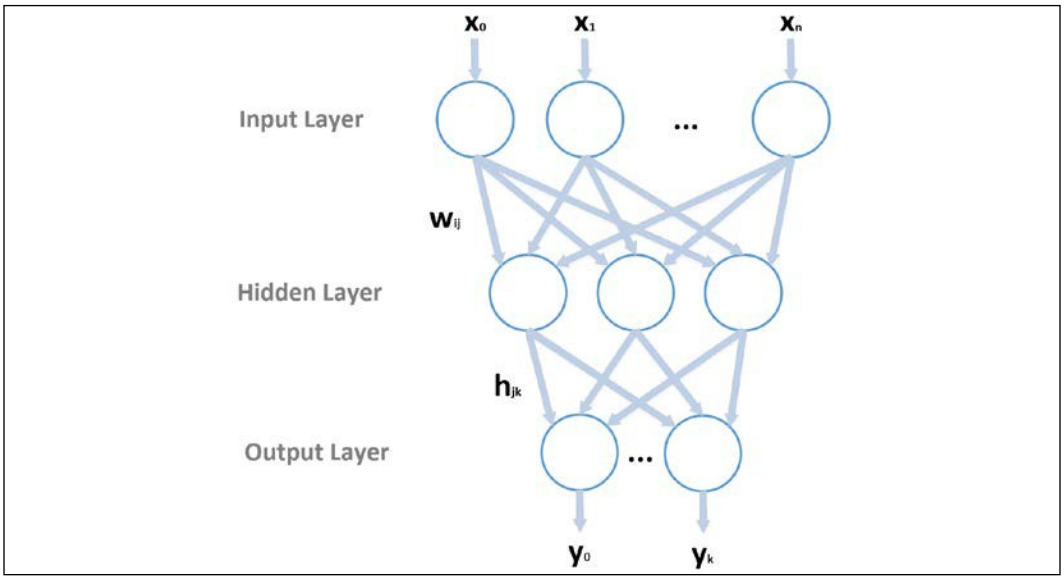

XOR dataset labeled using a Perceptron

It's obvious that a perceptron is another linear model without specific peculiarities, and its employment is discouraged in favor of other algorithms, such as logistic regression or SVM. After 1957, for a few years, many researchers didn't hide their disillusionment and considered the neural network as a promise never fulfilled. It was necessary to wait until a simple modification to the architecture, together with a powerful learning algorithm, officially opened the door to a new fascinating machine learning branch (later called deep learning).

In scikit-learn > 0.19, the class `Perceptron` allows adding `max_iter` or `tol` (tolerance) parameters. If not specified, a warning will be issued to inform you about future behavior. This piece of information doesn't affect the actual results.

# Multilayer Perceptrons (MLPs)

The main limitation of a perceptron is its linearity. How is it possible to exploit this kind of architecture by removing such a constraint? The solution is easier than you might speculate. Adding at least one non-linear layer between the input and output leads to a highly non-linear combination, parametrized with a larger number of variables. The resulting architecture, called a **Multilayer Perceptron** (**MLP**) and containing a single (just for simplicity) hidden layer, is shown in the following diagram:




Structure of a generic Multilayer Perceptron with a single hidden layer

This is a so-called feed-forward network, meaning that the flow of information begins in the first layer, always proceeds in the same direction, and ends at the output layer. Architectures that allow partial feedback (for example, in order to implement local memory) are called recurrent networks and will be analyzed in the next chapter.

In this case, there are two weight matrices, $W$ and $H$, and two corresponding bias vectors, $b$ and $c$. If there are m hidden neurons, $x\bar{x}_{ii} \in \mathbb{R}^{nn\times 1}$ (column vector), and $y\bar{y}_{ii} \in \mathbb{R}^{kk\times 1}$, the dynamics are defined by the following transformations:

$$\begin{cases} \bar{zz} = ff_h(WW^{TT}\bar{xx} + \bar{b}) \quad wwheeeeee \quad WW \in \mathbb{R}^{mn\times mm} \quad aaaaaa \quad \bar{b} \in \mathbb{R}^{mm\times 1} \\ \bar{y} = ff_{ha}(HH^{TT}\bar{zz} + \bar{cc}) \quad wwheeeeee \quad HH \in \mathbb{R}^{mm\times kk} \quad aaaaaa \quad \bar{cc} \in \mathbb{R}^{kk\times 1} \end{cases}$$

A fundamental condition for any MLP is that at least one hidden-layer activation function $ff_h(\bar{xx})$ is non-linear. It's straightforward to prove that $m$ linear hidden layers are equivalent to a single linear network and, hence, an MLP falls back into the case of a standard perceptron. In general, all hidden layers have non-linear activations, while the last one can also have a linear output to represent, for example, unbounded quantities (for example, in regression tasks). Conventionally, the activation function is fixed for a given layer, but there are no limitations on their combinations. In particular, the output activation is normally chosen to meet a precise requirement (such as multi-label classification, regression, image reconstruction, and so on). That's why the first step of this analysis concerns the most common activation functions and their features.

We can now have a deeper insight into the most common activation functions, discussing their features and their limitations.

# Activation functions

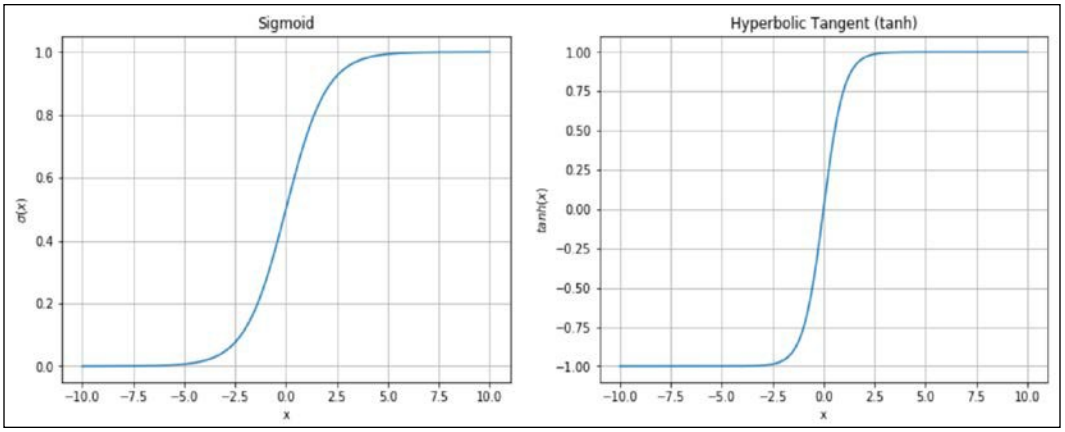
In general, any continuous (also step-wise) and differentiable function could be employed as an activation function. The continuity property allows us to take all values in the domain $D$ (generally $DD = \mathbb{R}$, so $f(x)$ is defined for any $x$), while the differentiability is a fundamental condition to optimize neural networks. Even so, some functions have particular properties that allow us to achieve good accuracy while improving the learning process speed. They're commonly used in state-of-the-art models, and it's important to understand their properties in order to make the most reasonable choice.

## Sigmoid and Hyperbolic Tangent

These two activations are very similar, with a very simple but important difference. Let's start defining them:

$$ff_{sssssssssssss}(xx) = \sigma\sigma(xx) = \frac{1}{1 + ee^{-xx}} \quad aaaaaa \quad ff_{ttttth}(xx) = \tanh xx = \frac{ee^{xx} - ee^{-xx}}{ee^{xx} + ee^{-xx}}$$

The corresponding plots are shown here:




Sigmoid and hyperbolic tangent plots

A sigmoid $\sigma(x)$ is bounded between 0 and 1, with two asymptotes ($\sigma(x) \to 0$ when $x \to -\infty$ and $\sigma(x) \to 1$ when $x \to +\infty$). Similarly, the hyperbolic tangent (tanh) is bounded between -1 and 1 with two asymptotes corresponding to the extreme values. Analyzing the two plots, we can discover that both functions are almost linear in a short range (about (-2, 2)), and they become almost flat immediately after. This means that the gradient is high and about constant when $x$ has small values around 0 and it falls down to about 0 for larger absolute values. A sigmoid perfectly represents a probability or a set of weights that must be bounded between 0 and 1, and therefore, it can be a good choice for some output layers.

However, the hyperbolic tangent is completely symmetric, and it's preferable for optimization purposes because its performance is normally superior. This activation function is often employed in intermediate layers, whenever the input is normally small. The reason will be clear when the back-propagation algorithm is analyzed; however, it's obvious that large absolute inputs lead to almost constant outputs, and since the gradient is about 0, the weight correction can become extremely slow (this problem is formally known as vanishing gradient). For this reason, in many real-world applications, the next family of activation functions is often employed.

# Rectifier activation functions

These functions are all linear (or quasi-linear for Swish) when $x > 0$, while they differ when $x < 0$. Even if some of them are not differentiable when $x = 0$, the derivative is always set equal to 0 in this case. The most common functions are as follows:

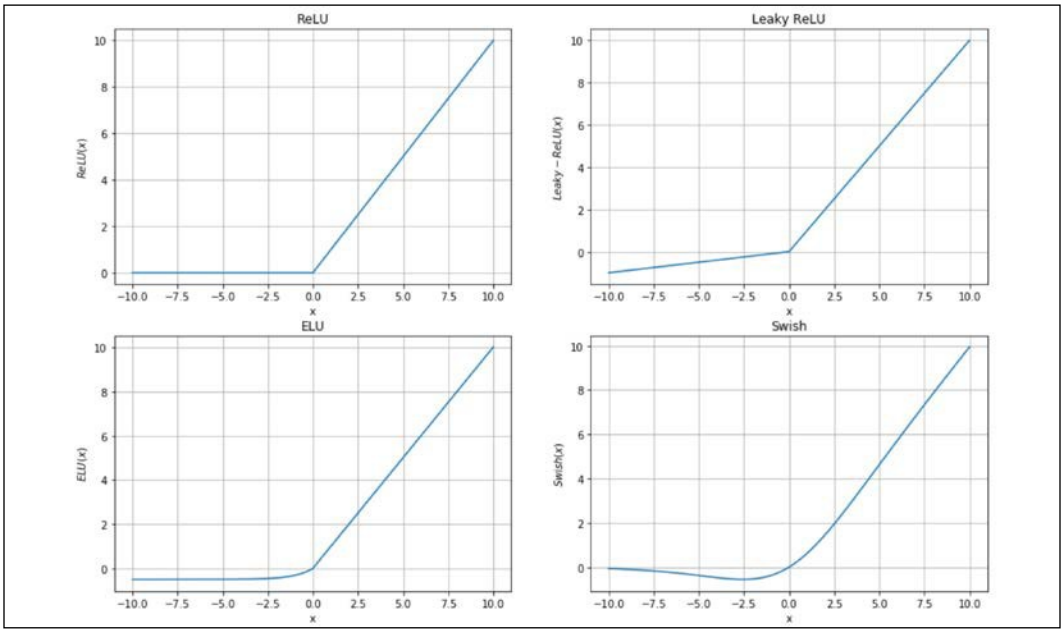
$$ff_{RRRRRRRR}(xx) = \max(0, xx)$$

$$ff_{RRRRLLLLLLRRRRRRRR}(xx) = \max(0, \alpha\alpha xx) \; wwwwh \; \alpha\alpha \leq 1$$

$$ff_{EERRRR}(xx) = \begin{cases} xx & wff \; xx > 0 \\ \alpha\alpha(ee^{xx} - 1) & oowwheeeewwwweeee \end{cases}$$

$$\{^{ff_{SSSSSSSSh}}(xx) = \frac{xx}{1 + ee^{-\alpha\alpha xx}} = xxxx(\alpha\alpha xx)$$

The corresponding plots are shown here:




ReLU (top-left), Leaky ReLU (top-right), ELU (bottom-left), and Swish (bottom-right)

The most basic function (and also the most commonly employed) is the ReLU, which has a constant gradient when $x > 0$, while it is null for $x < 0$. This function is very often employed in visual processing when the input is normally greater than 0 and has the extraordinary advantage of mitigating the vanishing gradient problem, as a correction based on the gradient is always possible. On the other hand, ReLU is null (together with its first derivative) when $x < 0$, therefore every negative input doesn't allow any modification. In general, this is not an issue, but there are some deep networks that perform much better when a small negative gradient is allowed. This consideration drove the creation of the other variants, which are characterized by the presence of the hyperparameter $\alpha\alpha$, which controls the strength of the negative tail. Common values between 0.01 and 0.1 allow behavior that is almost identical to ReLU, but with the possibility of a small weight update when $x < 0$.

The last function, called Swish and proposed in Ramachandran P., Zoph P., Le V. L., *Searching for Activation Functions*, arXiv:1710.05941 [cs.NE], is based on the sigmoid and offers the extra advantage of converging to 0 when $xx \to 0$, so the non-null effect is limited to a short region bounded between (-*b*, 0) with $b > 0$. This function can improve the performance of some particular visual processing deep networks, as discussed in the aforementioned paper. However, I always suggest starting the analysis with ReLU (which is very robust and computationally inexpensive) and switching to an alternative only if no other techniques can improve the performance of a model.

## Softmax

This function characterizes the output layer of almost all classification networks, as it can immediately represent a discrete probability distribution. If there are *k* outputs, $y_i$, the softmax, is computed as follows:

$$\mathbb{f}_{\text{ssssssssssss}}^{(ii)}(xx) = \frac{ee^{yy_{ii}}}{\sum_{jj=1}^{kk} ee^{yy_{jj}}}$$

In this way, the output of a layer containing *k* neurons is normalized so that the sum is always 1. It goes without saying that, in this case, the best cost function is cross-entropy. In fact, if all true labels are represented with one-hot encoding, they implicitly become probability vectors with 1 corresponding to the true class. The goal of the classifier is hence to reduce the discrepancy between the training distribution of its output by minimizing the function (see *Chapter 2*, *Loss functions and Regularization*):

$$LL(YY\,;\hat{Y},\,\bar{\theta\theta}) = -\sum_{ii=1}^{kk} yy_{ii}\log\hat{y}_{ww\boldsymbol{h}eeeeee\hat{y}\,iiii\;aa\;ppeeeeppiippppiippppp}$$

We can now discuss the training approach employed in an MLP (and almost all other neural networks).

# The back-propagation algorithm

This algorithm is more of a methodology than an actual algorithm. In fact, it was designed to be flexible enough to adapt to any kind of neural architecture without any substantial changes. Therefore, in this section we'll define the main concepts without focusing on a particular case. Those who are interested in implementing it will be able to apply the same techniques to different kinds of networks with minimal effort (assuming that all requirements are met).

The goal of a training process using a deep learning model is normally achieved by minimizing a cost function. Let's suppose we have a network parameterized with a global vector $\theta\theta$. In that case, the cost function (using the same notation for loss and cost but with different parameters to disambiguate) is defined as follows:

$$JJ(\theta\theta) = \frac{1}{MM}\sum_{ii=1}^{MM} LL(xx^{(ii)}, y^{(ii)}; \theta\theta)$$

We've already explained that the minimization of the previous expression (which is the empirical risk) is a way to minimize the real expected risk and, therefore, maximize the accuracy. Our goal is to find an optimal parameter set so that the following applies:

$$\theta\theta_{oooooo} = \operatorname*{argmin}_{\theta\theta} JJ(\theta\theta)$$

If we consider a single loss function (associated with a data point $xx^{(ii)}$ and a true vectorial label $y$), we know that such a function can be expressed with an explicit dependence on the predicted value:

$$LL(xx^{(ii)}, y; \theta\theta) = LL(y, \hat{y})$$

In the previous expression, the parameters have been embedded in the prediction. From calculus (without the excessive mathematical rigor that can be found in many books about optimization techniques), we know that the gradient of $LL(y, \hat{y})$, a scalar function, computed at any point (we are assuming the $L$ is differentiable) as a vector with components:

$$\nabla_{\theta} LL = \left(\frac{\partial LL}{\partial \theta_1}, \frac{\partial LL}{\partial \theta_2}, \dots, \frac{\partial LL}{\partial \theta_{TT}}\right)^{TT}$$

As $\nabla_{\theta\theta} LL$ always points in the direction of the closest maximum, so the negative gradient points in the direction of the closest minimum. Hence, if we compute the gradient of $L$, we have a ready-to-use piece of information that can be used to minimize the cost function. Before proceeding, it's useful to expose an important mathematical property called the chain rule of derivatives:

$$\frac{\partial f_1(f_2(\dots f_m(xx)\dots))}{\partial xx} = \frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}\cdots\frac{\partial f_m}{\partial xx}$$

Now, let's consider a single step in an MLP (starting from the bottom) and let's exploit the chain rule:

$$\bar{y} = f_{aa}(HH^{TT}\bar{zz} + \bar{cc})$$

Each component of the vector $y$ is independent of the others, so we can simplify the example by considering only an output value:

$$\hat{y}_{ll} = f_{aa}\left(\sum_{jj=1}^{kk} h_{jjii} z_{jj} + c_{ii}\right)$$

In the previous expression (discarding the bias), there are two important elements—the weights, $h_j$ (which are the columns of $H$), and the expression, $z\bar{z}_{jj}$, which is a function of the previous weights. As $L$ is, in turn, a function of all predictions $\hat{y}_{ll}$, applying the chain rule (using the variable $t$ as the generic argument of the activation functions), we get the following:

$$\frac{\partial\partial\partial}{\partial\partial h_{iiii}} = \frac{\partial\partial\partial}{\partial\partial\hat{y}_{ll}}\frac{\partial\partial\hat{y}_{ll}}{\partial\partial\partial}\frac{\partial\partial\partial}{\partial\partial h_{iiii}} = \frac{\partial\partial\partial}{\partial\partial\hat{y}_{ll}}\frac{\partial\partial\hat{y}_{ll}}{\partial\partial\partial} z_{ii} = \delta\delta_{ii}\, z_{ii}$$

As we normally cope with vectorial functions, it's easier to express this concept using the gradient operator. Simplifying the transformations performed by a generic layer, we can express the relations (with respect to a row of $H$, so to a weight vector $\bar{h}_{ii}$ corresponding to a hidden unit, $z\bar{z}_{ii}$) as follows:

$$\begin{cases} LL = pp(\hat{y}) \;\; ww\boldsymbol{h}eeeeee\; \bar{y} \in \mathbb{R}^{kk\times 1} \\ \bar{y} = \bar{q}(h_{ii}) \;\; ww\boldsymbol{h}eeeeee\; h_{ii} \in \mathbb{R}^{mm\times 1} \end{cases}$$

Employing the gradient and considering the vectorial output $\bar{y}$ can be written as $\bar{y} = (y_1, y_2, \ldots, y_m)$, we can derive the following expression:

$$\nabla_{h_{ii}} LL = J_{h_{ii}}(\hat{y})^{TT}\nabla_{\hat{y}} LL = \begin{pmatrix} \frac{\partial\partial\hat{y}_{ll}}{\partial\partial h_1} & \cdots & \frac{\partial\partial\hat{y}_{kk}}{\partial\partial h_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial\partial\hat{y}_{ll}}{\partial\partial h_{mm}} & \cdots & \frac{\partial\partial\hat{y}_{kk}}{\partial\partial h_{mm}} \end{pmatrix} \begin{pmatrix} \frac{\partial\partial LL}{\partial\partial\hat{y}_{ll}} \\ \vdots \\ \frac{\partial\partial LL}{\partial\partial\hat{y}_{kk}} \end{pmatrix} = \begin{pmatrix} \frac{\partial\partial LL}{\partial\partial h_1} \\ \vdots \\ \frac{\partial\partial LL}{\partial\partial h_{mm}} \end{pmatrix}$$

In this way, we get all the components of the gradient of $L$ computed with respect to the weight vectors, $\bar{h}_{ii}$. If we move back, we can derive the expression of $\bar{z}_{jj}$:

$$zz_{jj} = ff_h\left(\sum_{pp=1}^{mm} ww_{ppjj} xx_{pp} + bb_{jj}\right)$$

Reapplying the chain rule, we can compute the partial derivative of $L$ with respect to $w_{pj}$ (to avoid confusion, the argument of the prediction $y\hat{y}_{ii}$ is called $t_1$, while the argument of $zz_{jj}$ is called $t_2$):

$$\frac{\partial \partial \partial \partial}{\partial \partial ww_{pppp}} = \frac{\partial \partial \partial \partial}{\partial \partial \hat{yy}_{ii}} \frac{\partial \partial yy_{ii}}{\partial \partial tt_1} \frac{\partial \partial tt_1}{\partial \partial zz_{pp}} \frac{\partial \partial zz_{pp}}{\partial \partial tt_2} \frac{\partial \partial tt_2}{\partial \partial ww_{pppp}} = \delta\delta_{ii} hh_{mpii} \frac{\partial \partial zz_{pp}}{\partial \partial tt_2} xx_{pp}$$

Observing this expression (which can easily be rewritten using the gradient) and comparing it with the previous one, it's possible to understand the philosophy of the back-propagation algorithm, presented for the first time in Rumelhart D. E., Hinton G. E., Williams R. J., *Learning representations by back-propagating errors*, Nature 323, 1986. The data points are fed into the network and the cost function is computed. At this point, the process starts from the bottom, computing the gradients with respect to the closest weights and reusing a part of the calculation $\delta\delta_{ii}$ (proportional to the error) to move back until the first layer is reached. The correction is indeed propagated from the source (the cost function) to the origin (the input layer), and the effect is proportional to the responsibility of each different weight (and bias). Considering all the possible different architectures, writing all the equations for a single example would be useless.

A very important phenomenon that is worth considering was already outlined in the previous section and now it should be clearer: the chain rule is based on multiplications, and therefore, when the gradients start to become smaller than 1, the multiplication effect forces the last values to be close to 0. This problem is known as vanishing gradient and can really stop the training process of very deep models that use saturating activation functions (such as sigmoid or tanh). Rectifier units provide a good solution for many specific issues, but sometimes when functions such as hyperbolic tangent are necessary, other methods such as normalization must be employed to mitigate the phenomenon. We are going to discuss some specific techniques in this chapter and in the next one, but a generic best practice is to always work with normalized datasets and, if necessary, to also test the effect of whitening.

# Stochastic gradient descent (SGD)

Once the gradients have been computed, the cost function can be moved in the direction of its minimum. However, in practice, it is better to perform an update after the evaluation of a fixed number of training samples (a batch).

Indeed, the algorithms that are normally employed don't compute the global cost for the whole dataset, because this operation could be very computationally expensive. An approximation is obtained with partial steps, limited to the experience accumulated with the evaluation of a small subset. According to some literature, the expression stochastic gradient descent (SGD) should be used only when the update is performed after every single sample. When this operation is carried out on every $k$ points, the algorithm is also known as mini-batch gradient descent; however, conventionally, SGD is referred to as all batches containing $kk \geq 1$ data points, and we are going to use this expression from now on.

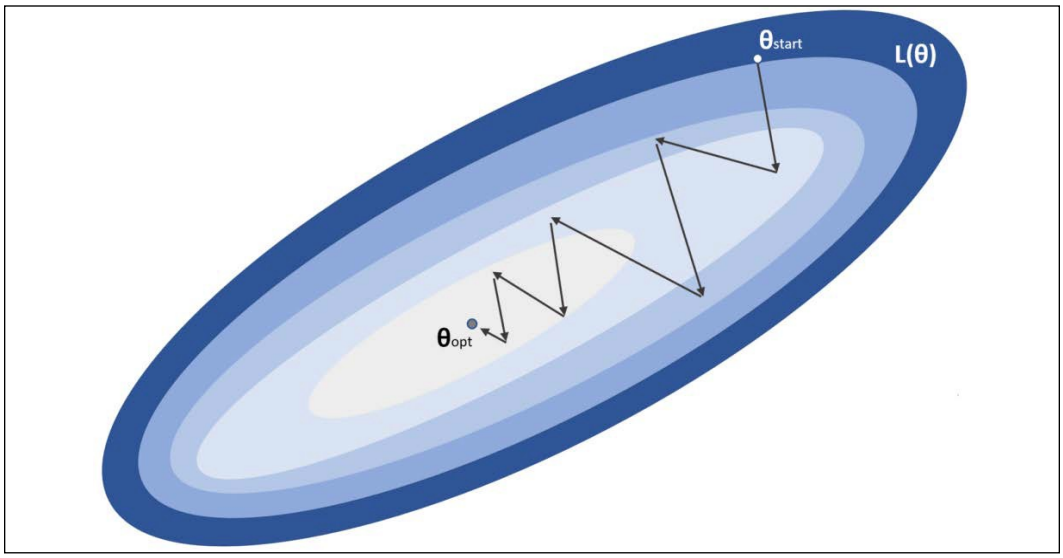
The process can be expressed considering a partial cost function computed using a batch containing k data points:

$$LL(\bar{\theta\theta}) = \frac{1}{kk} \sum_{ii=1}^{kk} LL(\bar{xx}_{ii}, \bar{y}_{ii}; \bar{\theta\theta})$$

The algorithm performs gradient descent by updating the weights according to the following rule:

$$\bar{\theta\theta}^{(tt+1)} = \bar{\theta\theta}^{(tt)} - \eta\eta \nabla_{\theta} L$$

If we start from an initial configuration $\bar{\theta\theta}_{sssssssss}$ and a target $\bar{\theta\theta}_{oooooo}$, the stochastic gradient descent process can be imagined like the path shown in the following diagram:




Graphical representation of the optimization process based on SGD

The weights are moved toward the minimum $\bar{\theta}_{000000}$, with many subsequent corrections that could also be wrong considering the whole dataset. For this reason, the process must be repeated several times (epochs), until the validation accuracy reaches its maximum. In a perfect scenario, with a convex cost function *L*, this simple procedure converges to the optimal configuration.

Unfortunately, a deep network is a very complex and non-convex function where plateaus and saddle points are quite common (see *Chapter 1*, *Machine Learning Model Fundamentals*). In such a scenario, a vanilla SGD wouldn't be able to find the global optimum and, in many cases, would not even find a close point. For example, in flat regions, the gradients can become so small (also considering the numerical imprecisions) as to slow down the training process until no change is possible (so $\theta\theta^{(tt+1)} \approx \theta\theta^{(tt)}$). In the next section, we are going to present some common and powerful algorithms that have been developed to mitigate this problem and dramatically accelerate the convergence of deep models.

Before moving on, it's important to mark two important elements. The first one concerns the learning rate, $\eta\eta$. This hyperparameter plays a fundamental role in the learning process. As is also shown in the figure, the algorithm proceeds, jumping from one point to another one (which is not necessarily closer to the optimum). Together with the optimization algorithms, it's absolutely important to correctly tune up the learning rate. A high value (such as 1.0) can move the weights too rapidly, increasing the instability. In particular, if a batch contains a few outliers (or simply non-dominant samples), a large $\eta\eta$ will consider them as representative elements, correcting the weights so as to minimize the error. However, subsequent batches might better represent the data-generating process, and therefore, the algorithm must partially revert its modifications in order to compensate for the incorrect update. For this reason, the learning rate is usually quite small with common values bounded between 0.0001 and 0.01 (in some particular cases, $\eta\eta = 0.1$ can also be a valid choice).

On the other hand, a very small learning rate leads to minimal corrections, slowing down the training process. A good trade-off, which is often the best practice, is to let the learning rate decay as a function of the epoch. In the beginning, $\eta\eta$ can be higher, because the probability of being close to the optimum is almost null; so, larger jumps can be easily adjusted. While the training process goes on, the weights are progressively moved toward their final configuration and, hence, the corrections become smaller and smaller. In this case, large jumps should be avoided, preferring fine-tuning. That's why the learning rate is decayed. Common techniques include exponential decay or linear decay. In both cases, the initial and final values must be chosen according to the specific problem (testing different configurations) and the optimization algorithm. In many cases, the ratio between the start and end value is about 10, and sometimes even larger.

Another important hyperparameter is the batch size. There are no silver bullets that allow us to automatically make the right choice, but some considerations can be made. As SGD is an approximate algorithm, larger batches can result in corrections that are probably more similar to the ones obtained considering the whole dataset. However, when the number of samples is extremely high, we don't expect the deep model to map them with a one-to-one association, but instead, our efforts are directed to improving the generalization ability. This feature can be re-expressed by saying that the network must learn a smaller number of abstractions and reuse them in order to classify new samples.

A batch, if sampled correctly, contains a part of these abstract elements and part of the corrections automatically improve the evaluation of a subsequent batch. You can imagine a waterfall process, where a new training step never starts from scratch. However, the algorithm is also called mini-batch gradient descent, because the usual batch size normally ranges from 16 to 512 (larger sizes are uncommon, but always possible), which are values smaller than the number of total samples (in particular, in deep learning contexts). A reasonable default value could be 32 data points, but I always suggest testing larger values, comparing performance in terms of training speed and final accuracy.

When working with deep neural networks, all the values (the number of neurons in a layer, batch size, and so on) are normally powers of two. This is not a constraint, but only an optimization tip (above all, when using GPUs), as the memory can be more efficiently filled when the blocks are based on $2^N$ elements. However, this is only a suggestion, whose benefits could also be negligible; so, don't be afraid to test architectures with different values. For example, in many papers, the batch size is 100 and some layers have 1,000 neurons.

# Weight initialization

A very important element is the initial configuration of a neural network. How should the weights be initialized? Let's imagine we that have set them all to zero. As all neurons in a layer receive the same input, if the weights are 0 (or any other common, constant number), the output will be equal. When applying the gradient correction, all neurons will be treated in the same way; so, the network is equivalent to a sequence of single-neuron layers. It's clear that the initial weights must be different to achieve a goal called *symmetry breaking*, but which is the best choice?

If we knew (also approximately) the final configuration, we could set them to easily reach the optimal point in a few iterations, but unfortunately, we have no idea where the minimum is located.

Therefore, some empirical strategies have been developed and tested, with the goal of minimizing the training time (obtaining *state-of-the-art* accuracy). A general rule of thumb is that the weights should be small (compared to the input sample variance). Large values lead to large outputs that negatively impact on saturating functions (such as tanh and sigmoid), while small values can be more easily optimized because the corresponding gradients are relatively larger, and the corrections have a stronger effect. The same is also true for rectifier units because maximum efficiency is achieved by working in a segment crossing the origin (where the non-linearity is actually located). For example, when coping with images, if the values are positive and large, a ReLU neuron becomes almost a linear unit, losing a lot of its advantages (that's why images are normalized, so as to bound each pixel value between 0 and 1 or -1 and 1).

At the same time, ideally, the activation variances should remain almost constant throughout the network, as well as the weight variances after every back-propagation step. These two conditions are fundamental in order to improve the convergence process and to avoid the vanishing and exploding gradient problems (the latter, which is the opposite of vanishing gradient, will be discussed in *Chapter 19*, *Deep Convolutional Networks*).

A very common strategy considers the number of neurons in a layer and initializes the weights as follows:

$$ww_{iiii} \sim NN\left(0, \frac{1}{nn}\right)$$

This method is called variance scaling and can be applied using the number of input units (Fan-In), the number of output units (Fan-Out), or their average. The idea is very intuitive: if the number of incoming or outgoing connections is large, the weights must be smaller, so as to avoid large outputs. In the degenerate case of a single neuron, the variance is set to 1.0, which is the maximum value allowed (in general, all methods keep the initial values for biases equal to 0.0 because it's not necessary to initialize them with a random value).

Other variations have been proposed, even if they all share the same basic ideas. LeCun proposed initializing the weights as follows:

$$ww_{iiii} \sim UU\left(-\sqrt{\frac{3}{nn_{ffffff-iiff}}}, \sqrt{\frac{3}{nn_{ffffff-iiff}}}\right)$$

Another method, called Xavier initialization (presented in Glorot X., Bengio Y., *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, 2010), is similar to LeCun initialization, but it's based on the average of the number of units of two consecutive layers (to mark the sequentiality, we have substituted the terms Fan-In and Fan-Out with explicit indices):

$$ww_{iiii} \sim UU\left(-\sqrt{\frac{6}{nn_{kk} + nn_{kk+1}}}, \sqrt{\frac{6}{nn_{kk} + nn_{kk+1}}}\right)$$

This is a more robust variant, as it considers both the incoming connections and also the outgoing ones (which are in turn incoming connections). The goal (widely discussed by the authors in the aforementioned papers) is to try to meet the two previously presented requirements. The first one is to avoid oscillations in the variance of the activations of each layer (ideally, this condition can avoid saturation). The second one is strictly related to the back-propagation algorithm, and it's based on the observation that, when employing variance scaling (or an equivalent uniform distribution), the variance of a weight matrix is proportional to the reciprocal of $3n_k$. Therefore, the averages of Fan-In and Fan-Out are multiplied by three, trying to avoid large variations in the weights after the updates. Xavier initialization has been proven to be very effective in many deep architectures, and it's often the default choice.

Other methods are based on a different way to measure the variance during both the feed-forward and back-propagation phases and trying to correct the values to minimize residual oscillations in specific contexts. For example, He, Zhang, Ren, and Sun (in He K., Zhang X., Ren S., Sun J., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, arXiv:1502.01852 [cs.CV]) analyzed the initialization problem in the context of convolutional networks (we are going to discuss them in the next chapter) based on ReLU or variable Leaky-ReLU activations (also known as PReLU – parametric ReLU), deriving an optimal criterion (often called the He initializer), which is slightly different from the Xavier initializer:

$$ww_{iiii} \sim UU\left(-\sqrt{\frac{6}{nn_{ffffff-iiff}}}, \sqrt{\frac{6}{nn_{ffffff-iiff}}}\right)$$

All these methods share some common principles, and in many cases, they are interchangeable. As already mentioned, Xavier is one of the most robust and, in the majority of real-life problems, there's no need to look for other methods; however, you should always be aware that the complexity of deep models must often be faced using empirical methods based on sometimes simplistic mathematical assumptions. Only validation with a real dataset can confirm whether a hypothesis is correct or it's better to continue the investigation in another direction.

# Example of MLP with TensorFlow and Keras

Keras (`https://keras.io`) is a powerful Python toolkit that allows modeling and training complex deep learning architectures with minimal effort. Thanks to its flexibility, it has been incorporated into TensorFlow, which has become its predefined backend. Therefore, from now on, we are going to refer to TensorFlow 2.0 (for further details, I suggest the book Holdroyd T., *TensorFlow 2.0 Quick Start Guide*, Packt Publishing, 2019). When it's not necessary to use advanced features, we employ the Keras API through TensorFlow. If you want to use another backend, you'll have to install Keras separately and follow the instructions in the documentation to configure it properly.

Tensorflow can be installed using the command `pip -U install tensorflow` (or `tensorflow-gpu` for GPU support). All of the required documentation can be found on the official page at `https://www.tensorflow.org/`.

In this example, we want to build a small MLP with a single hidden layer to solve the XOR problem (the dataset is the same as was created in the previous example). The simplest and most common way is to instantiate the class `Sequential`, which defines an empty container for an indefinite model. In this initial part, the fundamental method is `add()`, which allows adding a layer to the model. For our example, we want to employ four hidden layers with hyperbolic tangent activation and two `softmax` output layers.

The following snippet defines the MLP:

```
import tensorflow as tf

model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(4, input_dim=2,
                              activation='tanh'),
        tf.keras.layers.Dense(2, activation='softmax')
])
```

The `Dense` class defines a fully connected layer (a classical MLP layer), and the first parameter is used to declare the number of desired units. The first layer must declare the `input_shape` or `input_dim`, which specify the dimensions (or the shape) of a single sample (the batch size is omitted as it's dynamically set by the framework). All the subsequent layers compute the dimensions automatically. One of the strengths of Keras is the possibility to avoid setting many parameters (such as weight initializers), as they will be automatically configured using the most appropriate default values (for example, the default weight initializer is Xavier).

In the next examples, we're going to explicitly set some of them, but I suggest you check the official documentation to get acquainted with all the possibilities and features. The other layer involved in this experiment is **activation**, which specifies the desired activation function (it's also possible to declare it using the parameter activation implemented by almost all layers, but I prefer to decouple the operations to emphasize the single roles, and also because some techniques—such as batch normalization—are normally applied to the linear output before the activation).

At this point, we must ask Keras to compile the model (using the preferred backend):

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The parameter optimizer defines the stochastic gradient descent algorithm that we want to employ. Using `optimizer='sgd'`, it's possible to implement a standard version (as described in the previous paragraph). In this case, we're employing Adam (with the default parameters), which is a much more performant variant that will be discussed in the next chapter. The parameter `loss` is used to define the cost function (in this case, cross-entropy) and `metrics` is a list of all the evaluation scores we want to be computed (`'accuracy'` is enough for many classification tasks). Once the model is compiled, it's possible to train it:

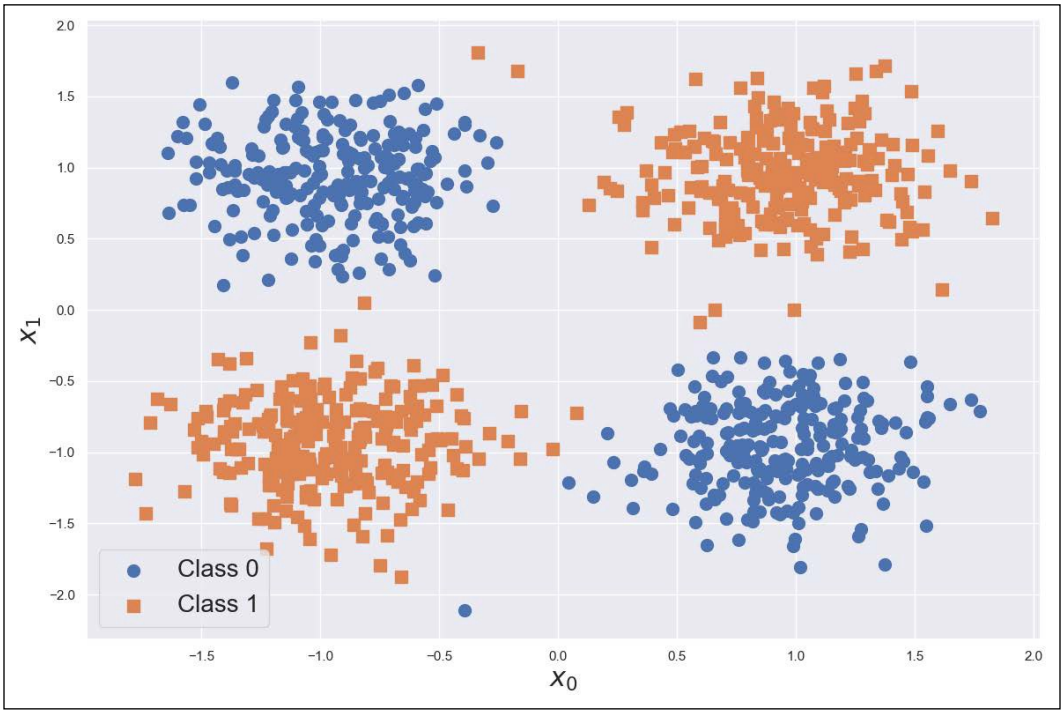
```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = \
        train_test_split(X, Y, test_size=0.3,
                         random_state=1000)

model.fit(X_train,
             tf.keras.utils.to_categorical(
          Y_train, num_classes=2),
             epochs=100,
             batch_size=32,
             validation_data=
             (X_test,
             tf.keras.utils.to_categorical(
                Y_test, num_classes=2)))
```

The output of the previous snippet is:

```
Train on 700 samples, validate on 300 samples
Epoch 1/100
700/700 [==============================] - 1s 2ms/sample - loss:
0.7453 - accuracy: 0.5114 - val_loss: 0.7618 - val_accuracy: 0.4767
Epoch 2/100
700/700 [==============================] - 1s 1ms/sample - loss:
0.7304 - accuracy: 0.5129 - val_loss: 0.7465 - val_accuracy: 0.4833
Epoch 3/100
700/700 [==============================] - 1s 2ms/sample - loss:
0.7177 - accuracy: 0.5143 - val_loss: 0.7342 - val_accuracy: 0.4900
…
Epoch 99/100
700/700 [==============================] - 1s 1ms/sample - loss:
0.0995 - accuracy: 0.9914 - val_loss: 0.0897 - val_accuracy: 0.9967
Epoch 100/100
700/700 [==============================] - 1s 2ms/sample - loss:
0.0977 - accuracy: 0.9914 - val_loss: 0.0878 - val_accuracy: 0.9967
```

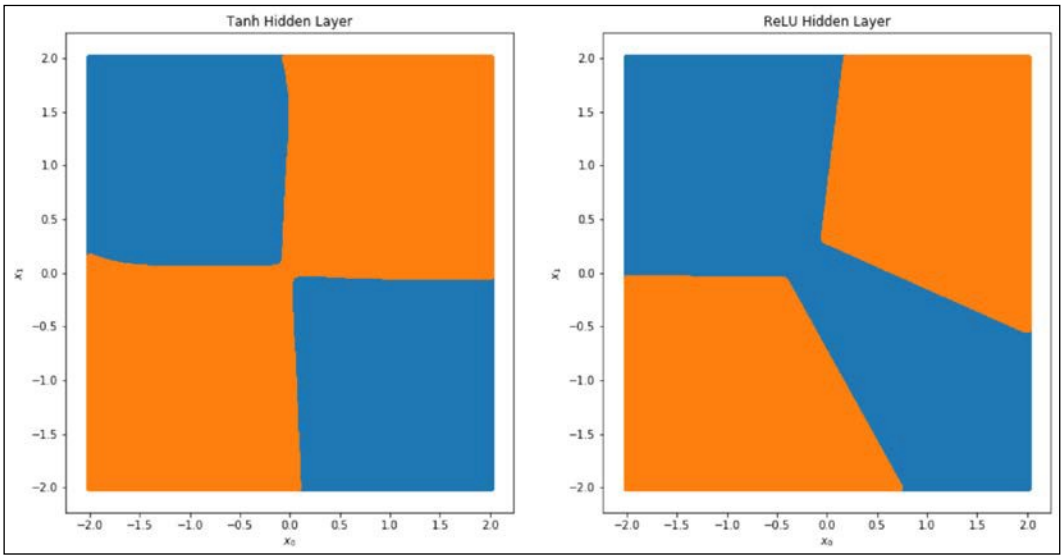
The operations are quite simple. We've split the dataset into training and test/validation sets (in deep learning, cross-validation is seldom employed), and then, we have trained the model setting `batch_size=32` and `epochs=100`. The dataset is automatically shuffled at the beginning of each epoch, unless setting `shuffle=False`. In order to convert the discrete labels into one-hot encoding, we have used the utility function `to_categorical`. In this case, the label 0 becomes (1, 0) and the label 1 (0, 1). The model converges before reaching 100 epochs; therefore, I invite you to optimize the parameters as an exercise. However, at the end of the process, both accuracies are extremely close to 1.

The final classification plot is shown here:




MLP classification of the XOR dataset

Only a few points (which can also be considered as outliers) have been misclassified, but it's clear that the MLP successfully separated the XOR dataset. To get confirmation of the generalization ability, we've plotted the decision surfaces for a hyperbolic tangent hidden layer and a ReLU one:




MLP decision surfaces with Tanh (left) and ReLU (right) hidden layer

In both cases, the MLPs delimited the areas in a reasonable way. However, while a hyperbolic tangent hidden layer seems to be overfitted (this is not true in our case, as the dataset represents exactly the data-generating process), the ReLU layer generates less smooth boundaries with an apparent lower variance (in particular, for considering the outliers of a class). We know that the final validation accuracies confirm an almost perfect fit, and the decision plots (which are easy to create with two dimensions) show acceptable boundaries in both cases, but this simple exercise is useful to understand the complexity and the sensitivity of a deep model and to gain a better comprehension about the best way to structure it. For these reasons, it's absolutely necessary to select a valid training set (representing the ground truth) and employ all possible techniques to avoid overfitting (as we're going to discuss later). The easiest way to detect such a situation is to check the validation loss. A good model should reduce both training and validation loss after each epoch, reaching a plateau for the latter. If, after $n$ epochs, the validation loss (and, consequently, the accuracy) begins to increase, while the training loss keeps decreasing, it means that the model is overfitting the training set.

Another generally valid empirical indicator that the training process is evolving correctly is that, at least at the beginning, the validation accuracy should be higher than the training accuracy. This might seem strange, but we need to consider that the validation set is slightly smaller and less complex than the training set; therefore, if the capacity of the model is not saturated with training samples, the probability of misclassification is higher for the training set than for the validation set.

When this trend is inverted, the model is very likely to overfit after a few epochs. The advice is generally correct, however it's important to remember that the complexity of these models sometimes leads to unpredictable behaviors. Hence, before aborting a training process, it's always helpful to wait for at least one third of the total epochs. To verify these concepts, I invite you to repeat the exercise using a large number of hidden neurons (so as to dramatically increase the capacity), but they will be clearer when working with much more complex and unstructured datasets.

# Summary

In this chapter, we started our exploration of the deep learning world by introducing the basic concepts that led the first researchers to improve algorithms until they achieved the top results we can achieve nowadays. The first part explained the structure of a basic artificial neuron, which combines a linear operation followed by an optional non-linear scalar function. A single layer of linear neurons was initially proposed as the first neural network, with the name of the perceptron.

Even though it was quite powerful for many problems, this model soon showed its limitations when working with non-linear separable datasets. A perceptron is not very different from logistic regression, and there's no concrete reason to employ it. Nevertheless, this model opened the doors to a family of extremely powerful models obtained by combining multiple non-linear layers. The multilayer perceptron, which has been proven to be a universal approximator, is able to manage almost any kind of dataset, achieving high-level performance when other methods fail.

In the next section, we analyzed the building blocks of an MLP. We started with the activation functions, describing their structures and features, and focused on the reasons they are the main choice for specific problems. Then, we discussed the training process, considering the basic idea behind the back-propagation algorithm and how it can be implemented using the stochastic gradient descent method. Even though this approach is quite effective, it can be slow when the complexity of the network is very high. For this reason, many optimization algorithms were proposed. In this chapter, we analyzed the role of momentum and how it's possible to manage adaptive corrections using RMSProp. Then, we combined momentum and RMSProp to derive a very powerful algorithm called Adam. In order to provide a complete picture, we also presented two slightly different adaptive algorithms, called AdaGrad and AdaDelta.

In the next chapter, we are going to discuss the most important neural network optimization strategies (including RMSProp and Adam) and how to use regularization and other techniques that improve the overall performances of the models both in terms of speed and accuracy.

# Further reading


- Minsky M. L., Papert S. A., *Perceptrons*, The MIT Press, 1969
- Ramachandran P., Zoph P., Le V. L., *Searching for Activation Functions*, arXiv:1710.05941 [cs.NE]
- Rumelhart D. E., Hinton G. E., Williams R. J., *Learning representations by back-propagating errors*, Nature 323, 1986
- Glorot X., Bengio Y., *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, 2010
- He K., Zhang X., Ren S., Sun J., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, arXiv:1502.01852 [cs.CV]
- Holdroyd T., *TensorFlow 2.0 Quick Start Guide*, Packt Publishing, 2019
- *Kingma D. P., Ba J., Adam: A Method for Stochastic Optimization, arXiv:1412.6980 [cs.LG]*
- Duchi J., Hazan E., Singer Y., *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research 12, 2011
- Zeiler M. D., *ADADELTA: An Adaptive Learning Rate Method*, arXiv:1212.5701 [cs.LG]
- Hornik K., *Approximation Capabilities of Multilayer Feedforward Networks*, Neural Networks, 4/2, 1991
- Cybenko G., *Approximations by Superpositions of Sigmoidal Functions*, Mathematics of Control, Signals, and Systems, 2 /4, 1989