Documentation - multilayer_perceptron

Thibault Nguyen - thibnguy

August 2025

Contents

1 Machine Learning and Deep Learning Basis

1.1 Definition of the Machine Learning:

Machine learning is a field of **artificial intelligence**, which consists in programming a machine to learn how to perform tasks by studying examples of these tasks.
From a mathematical point of view, these examples are represented by **data** that the machine uses to develop a **model**. for example a function of the type $f(x) = ax + b$.
The goal of the **machine learning** is to find the **parameters** a and b that give the best possible **model**, that is, the **model** that best fits our **data**.
For this, we program in the machine an **optimization algorithm** that will test different values of a and b until we obtain the combination that **minimizes the distance** between the **model** and the **points**.

1.2 Definition of the Deep Learning:

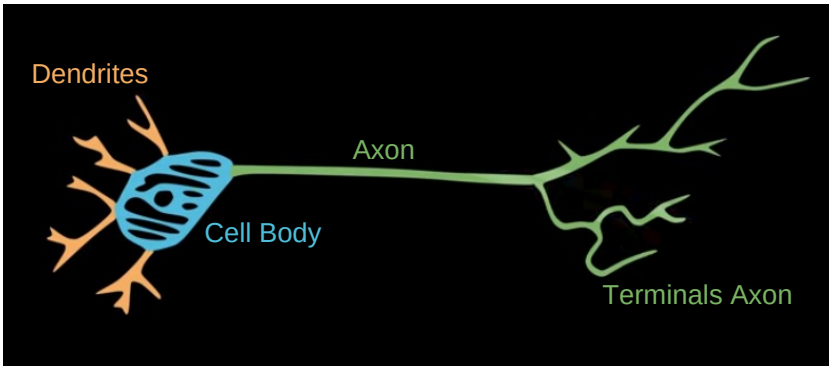
Deep Learning is a domain of **Machine Learning** in which we develop what we call **artificial neural networks**.
The principle remains exactly the same: we provide the machine with **data**, and it uses an **optimization algorithm** to adjust the **model** to these **data**.
But this time, our **model** is not a simple function of the type ~~$f(x) = ax + b$~~, but rather a network of interconnected functions: a **neural networks**.
The more the machine is able to learn to do complex tasks, like recognizing objects, identifying a person in a picture, driving a car, etc.

1.3 History of the Deep Learning

1.3.1 Functioning of a Biological Neuron

In 1943 by two mathematicians and neuroscientists named **Warren McCulloch** and **Walter Pitts** invented the **first neural networks** by studying the functioning of **biological neurons**.
In biology, neurons are excitable cells connected to each other and their role is

to transmit information in our nervous system.
Each neuron is composed of several **dendrites**, a **cell body**, and an **axon**.

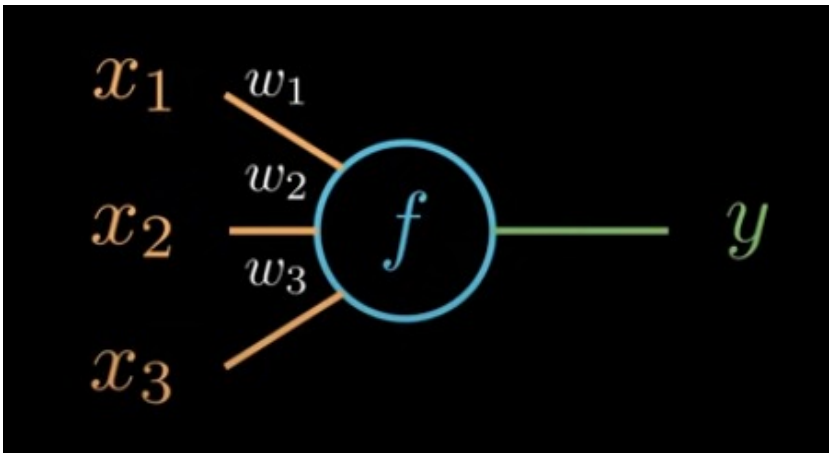


Dendrites are the **gateways** to a neuron. It is at this point, at the **synapse**, that the neuron receives signals from the preceding neurons. These signals can be **excitatory** or **inhibitory** (transmit signal or not, signal +1/-1). When the sum of these signals exceeds a certain **threshold**, the neuron **activates** and produces an electrical signal.
This signal travels along the axon to the endings and is sent to other neurons in our nervous system...

1.3.2 Functioning of an Artificial Neuron

What **Warren McCulloch** and **Walter Pitts** have tried to do is to **model** this operation, by considering that a **neuron** could be represented by a **transfer function**, which takes as **input** X signals and returns an **output** y.
Inside this function, there are 2 main steps.
The first one is an **aggregation** step. We make the sum of all the **inputs** of the **neuron**, by multiplying each input by a **coefficient** W. This coefficient represents the **synaptic activity** (whether the signal is excitatory, in which case w is $+1$, or inhibitory, in which case it is -1).
In this **aggregation** phase, we obtain an expression of the form

$$f = w1.x1 + w2.x2 + w3.x3 + \cdots + wn.xn$$



Once this step has been completed, we move on to the **activation** phase. We look at the result of the calculation made previously, and if it exceeds a certain **threshold**, usually 0, then the neuron is activated and returns an output y = 1. Otherwise, it remains at 0.

2 The Perceptron

2.1 History of the Perceptron

In 1957, the psychologist **Franck Rosenblatt** created the first **learning algorithm** in the history of **Deep Learning**.
The model of **perceptron** is an artificial neuron, which is activated when the **weighted sum** of its inputs exceeds a certain **threshold**, usually 0.
But with this, the perceiver also has a **learning algorithm** allowing it to find the values of its **parameters** w in order to obtain the **outputs** y that we want.
To develop this algorithm, **Frank Rosenblatt** was inspired by **Hebb's theory**.
This theory suggests that when two biological neurons are jointly **excited**, then they strengthen their **synaptic links** that is, they strengthen the connections between them. In neuroscience, this is called **synaptic plasticity**, and this

is what allows our brain to build memory, to learn new things, to make new associations.

2.2 Application of the Perceptron

So from this idea, **Frank Rosenblatt** developed a **learning algorithm**, which consists in **training** an artificial neuron on reference **data** (X, y) so that it **reinforces** its **parameters** w each time an **input** X is **activated** at the same time as the **output** y present in these data.
To do this, he devised the following formula, in which the **parameters** w are **updated** by calculating the difference between the **reference output** and the **output produced** by the neuron, and multiplying this difference by the value of each **input** X, as well as by a **positive learning step**.
This way, if our neuron produces a different **output** than the one it is supposed to produce, for example if it outputs $y = 0$, while we would like to have $y = 1$, then our formula will give us

$$W = W + \alpha(y_{true} - y)X$$

So, for the **inputs** x that are worth 1, the **coefficient** w will be increased by a small **alpha step**. It will be reinforced which will cause an increase of the function $w1.x1 + w2.x2$, which will bring our neuron closer to its **activation threshold**.
As long as we are below this **threshold**, that is to say as long as the neuron produces a bad **output**, then the **coefficient** w will continue to increase thanks to our formula, until $y_{true} == y$ and at that moment our formula will give $w = w + 0$. This means that our **parameters** will stop **evolving**.

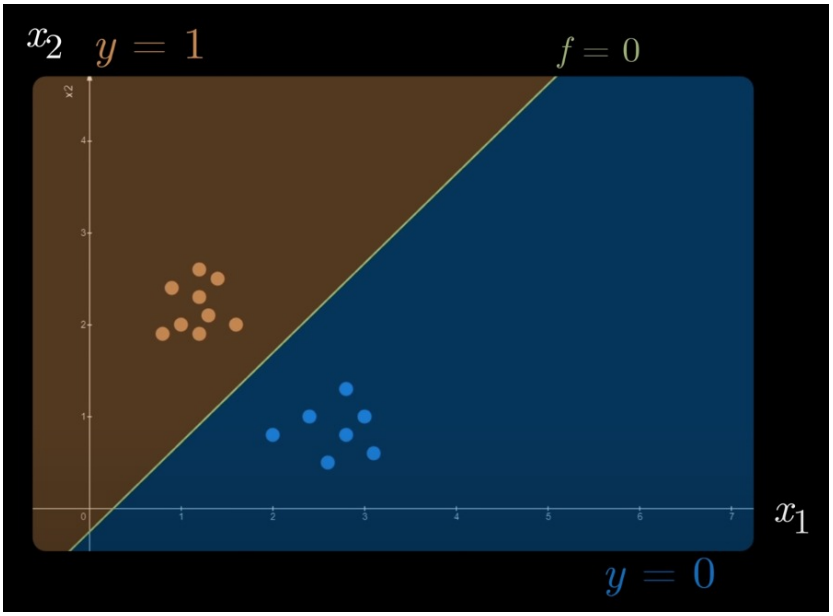
3 Multilayer Perceptron

In the 80s, **Geoffrey Hinton** developed the **multi-layer perceptron**, the first true **artificial neural networks**.
the **Perceptron** is actually a **linear model**.
Indeed, if we plot the graph of its **aggregation function** $f(x1, x2) = w1.x1 +$

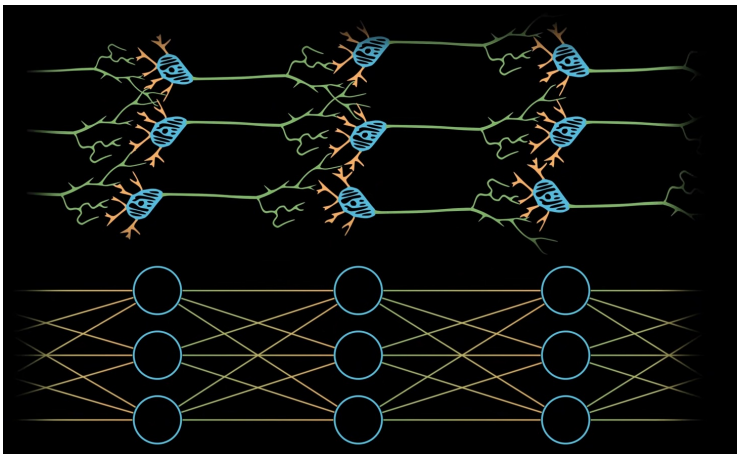
$w2.x2$ we then obtain a line, whose inclination depends on the **parameters** w and whose position can be modified with a small additional parameter called the *bias*.

$$f(x1, x2) = w1.x1 + w2.x2 + b$$

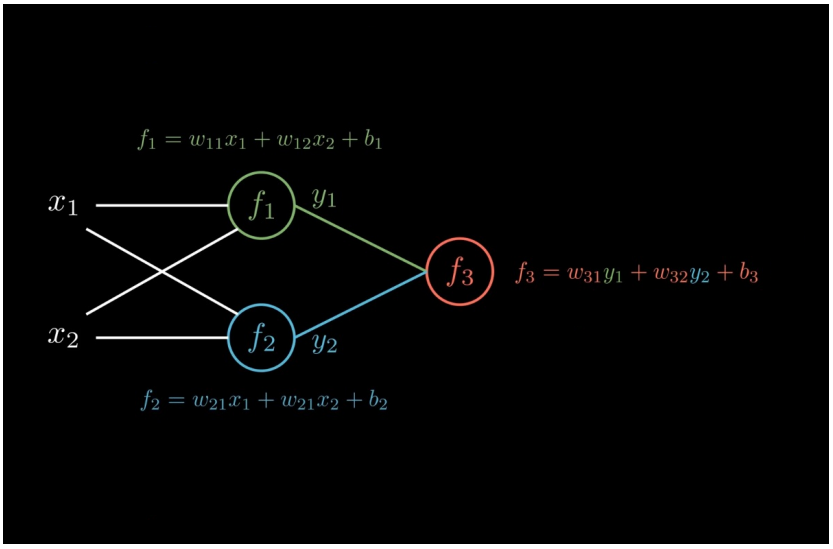
With this line you can do great things, like separating 2 classes of points, since thanks to our **activation function** everything above this line will give an **output** $y = 1$ and everything below will give $y = 0$.



The only problem is that a lot of the phenomena in our universe are not **linear phenomena**.
And in that case, the perceptron alone is not very useful.
But remember the idea of **McCulloch** and **Pitts**: by connecting several **neurons** together, it is possible to solve more complex problems than with one.

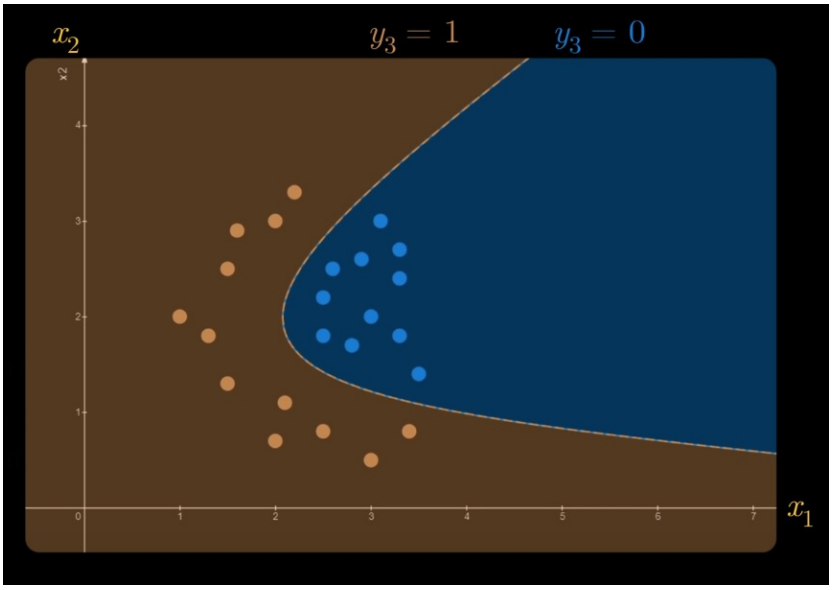


So let's see what happens if we connect for example **3 Perceptrons** together.
The first 2 receive each the **inputs** $x1$ and $x2$. They do their little calculation, according to their **parameters**, and return an **output** y which they send in turn to the **third Perceptron**, which will also make its own calculations to produce a **final output**.

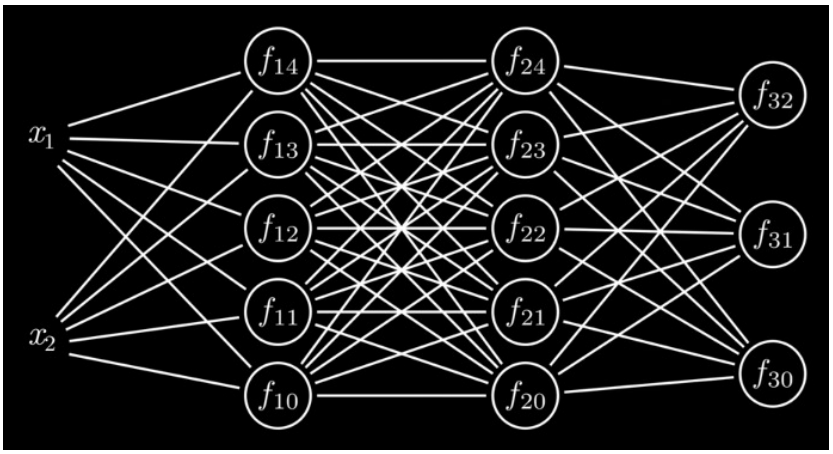


If we plot the **graphical representation** of **the final output** as a func-

tion of the **inputs** $x1$, $x2$, we obtain this time a **non linear model** which is much more interesting.



With this example, you have your first **artificial neural network**.
3 neurons, divided in 2 layers (an input layer and an output layer) this is what we call a **Multilayer Perceptron**. And you can add as many layers and neurons as you want.



4 Training Phase

4.1 Feed Forward

The **Feed Forward** determine how the **input data** travels through the network to produce an **output**. For this, we pass the input values to the first layer, compute its output, then feed this output into the next layer, and so on, until we reach the final layer of the **model**. At each step, the information is transformed by the layer's **parameters** and **activation functions**, progressively building the network's **prediction**. With this process, we obtain the model's **output** for a given **input**, which can then be compared to the expected response (y_{true} value) to evaluate the **performance** of the network.

For each $c = 1, 2, \ldots, C$:
Linear combination:

$$Z^{[c]} = W^{[c]}.A^{[c-1]} + b^{[c]}$$

Activation function:

$$a^{[c]} = \begin{cases} softmax(Z^{[c]}) & \text{if } c = C \text{ (output layer)} \\ \sigma(Z^{[c]}) = \frac{1}{1+e^{-Z^{[c]}}} & \text{if } c < C \text{ (hidden layers)} \end{cases} \tag{1}$$

where:

- $A^{[0]} = X$ (input data)
- $W^{[c]}$: Weight matrix of layer c
- $b^{[c]}$: Bias vector of layer c
- $Z^{[c]}$: Pre-activation value of layer c
- $A^{[c]}$: Activation output of layer c

Softmax function:

$$softmax(z_j) = \frac{e^{z_j - max(z)}}{\sum_{k=1}^{K} e^{z_k - max(z)}}$$

where:

- z_j: Pre-activation value for class j in the output layer.
- K: Total number of classes.
- $max(z)$: Maximum value among all z_k (used for numerical stability).
- e: Base of the natural logarithm.

4.2 Activation Function: Sigmoid

The **Sigmoid** function is one of the most widely used **activation functions** in neural networks, especially in the early stages of Deep Learning. Its role is to introduce **non-linearity** into the model, allowing the network to learn more complex relationships than a simple linear function.
The Sigmoid function takes any real-valued input and compresses it into a range between 0 and 1, which makes it particularly suitable for representing probabilities. However, it also has drawbacks, such as the **vanishing gradient problem**, since very large or very small input values produce gradients close to zero, slowing down learning in deep networks.

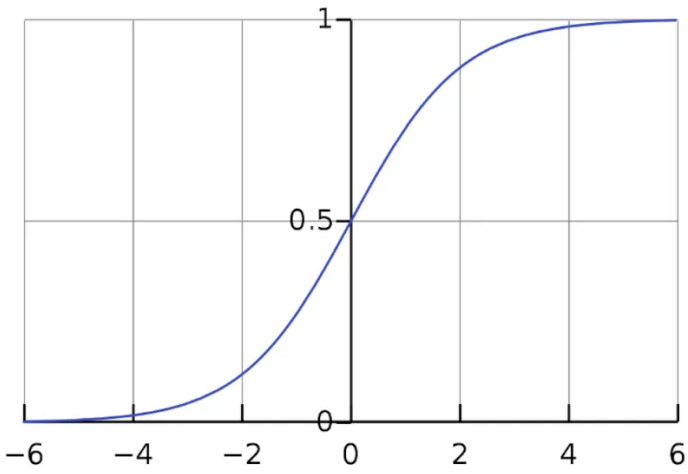
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The derivative of the Sigmoid, used in **Back Propagation**, is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

where:

- z: Input value of the neuron before activation (pre-activation).
- $\sigma(z)$: Output of the Sigmoid function, bounded between 0 and 1.
- $\sigma'(z)$: Derivative of the Sigmoid, used to compute gradients during Back Propagation.



4.3 Back Propagation

The **Back Propagation** is here to determine how the **output** of the network varies according to the **parameters** present in each layer of the **model**. For this, we compute a gradient chain, indicating how the output varies according to the last layer, then how the last layer varies according to the second last, then how the second last varies according to the second last, etc, until we get to the very first layer of our network.
With this information, these **gradients**, we can then update the **parameters** of each layer, so that they **minimize** the **error** between the **model output** and the expected response (y_{true} value).

Step 1 — Output layer ($c = C$):

$$dZ^{[C]} = A^{[C]} - Y$$

Step 2 — Gradients for layer c:

$$dW^{[c]} = \frac{1}{m}\, dZ^{[c]}\, (A^{[c-1]})^T$$

$$db^{[c]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[c](i)}$$

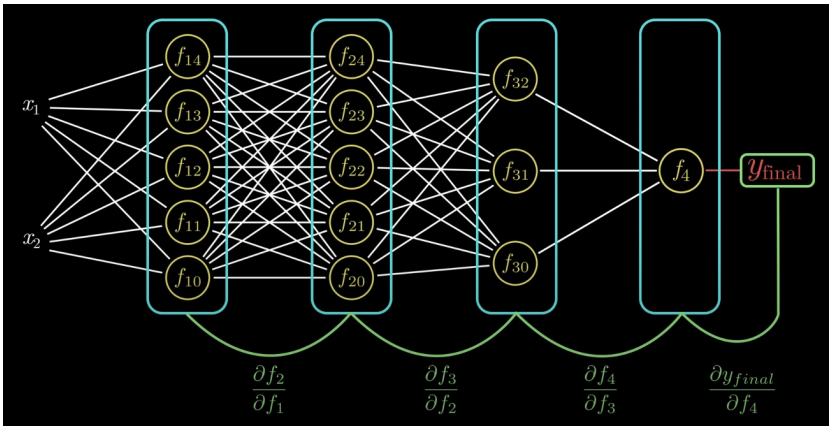
Step 3 — Backpropagation to the previous layer ($c > 1$):

$$dZ^{[c-1]} = (W^{[c]})^T \, dZ^{[c]} \odot \sigma'(Z^{[c-1]})$$

$$\sigma'(Z^{[c-1]}) = A^{[c-1]} \cdot (1 - A^{[c-1]})$$

where:

- m: Number of training examples.
- Y: One-hot encoded true labels.
- $A^{[c]}$: Activation of layer c.
- $Z^{[c]}$: Pre-activation value, $Z^{[c]} = W^{[c]} A^{[c-1]} + b^{[c]}$.
- $W^{[c]}, b^{[c]}$: Weights and biases of layer c.
- $dW^{[c]}, db^{[c]}$: Gradients of the cost with respect to weights and biases.
- $\odot$: Element-wise (Hadamard) product.



4.4 Loss Function

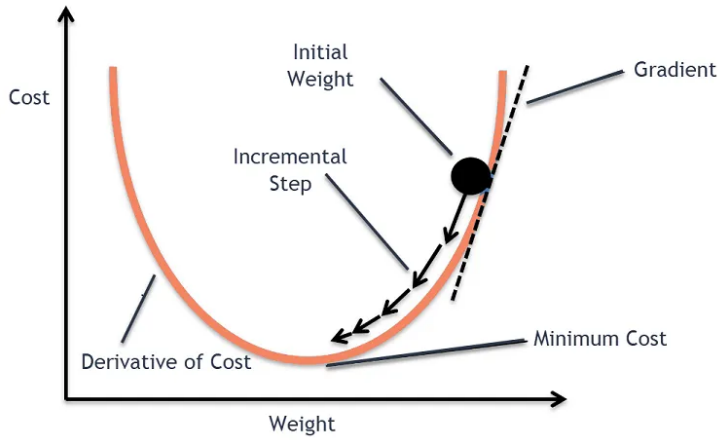
The **Categorical Cross Entropy** is used as the **loss function** to measure how far the predicted probability distribution of the network is from the true

distribution of the target classes. For this, we compare the **probability** assigned to the correct class with the **actual expected** value (y_{true}), applying a **logarithmic penalty**: the smaller the predicted probability for the correct class, the higher the penalty. This **loss** is computed for each sample and then averaged over the batch, giving a single value that reflects the **model's overall performance**. With this value, we can guide the optimization process to adjust the network's parameters and improve its classification **accuracy**.

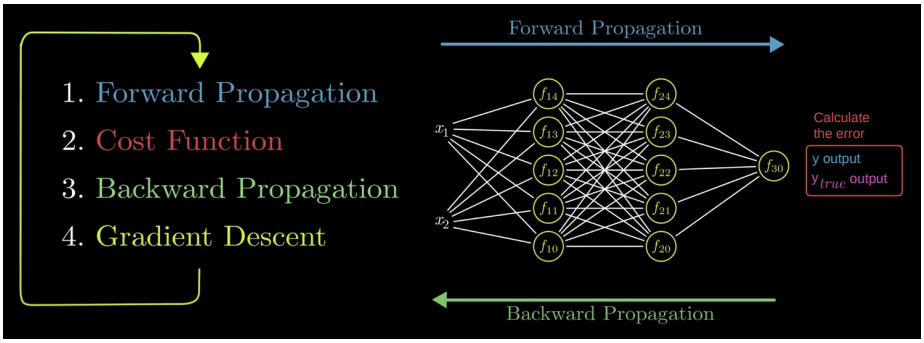
$$CCE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} log(p_{i,c})$$

4.5 Gradient Descent

The **Gradient Descent** is used as the **optimization algorithm** to **update** the **parameters** of the network in order to **minimize** the **loss function**. For this, we compute the **gradients** of the loss with respect to each **parameter** (from the **Back Propagation** process) and adjust the parameters in the opposite direction of the **gradient**. The size of this adjustment is controlled by the **learning rate**: a small rate leads to slower but more stable convergence, while a large rate can speed up learning but risks overshooting the minimum. This process is repeated **iteratively** over the dataset, progressively moving the parameters toward values that reduce the error and improve the network's performance.



To develop and train **artificial neural networks**, we repeat the following **four steps** in a loop: The first step is the **Forward Propagation** step: we pass the **data** from the first layer to the last, in order to produce an **output** y. The second step is to calculate the **error** between this **output** and the **reference output** y_{true} that we want to have. For this we use what is called a **cost function**. Then, the third step is the **Back Propagation**, we measure how this **cost function** varies with respect to each layer of our **model**, starting from the last one and going up to the very first one. Finally, the fourth and last step is to correct each **parameter** of the **model** with the **gradient descent algorithm**, before looping back to the first step, **Forward Propagation**, to start a new training cycle.



4.6 Nesterov Momentum

The **Nesterov Accelerated Gradient (NAG)** is an improvement of the **Gradient Descent with Momentum**. Instead of computing the gradient at the current parameters, it looks ahead by applying the momentum term first, then evaluates the gradient at this look-ahead position. This anticipatory update often leads to faster convergence and better stability compared to classical momentum.

Step 1 — Look-ahead:

$$\tilde{\theta}_t = \theta_{t-1} - \mu\, v_{t-1}$$

Step 2 — Gradient at look-ahead:

$$g_t = \nabla_\theta \mathcal{L}(\tilde{\theta}_t)$$

Step 3 — Velocity and parameter update:

$$v_t = \mu\, v_{t-1} + \eta\, g_t$$

$$\theta_t = \theta_{t-1} - v_t$$

where:

- θ: Model parameters (weights $W^{[c]}$, biases $b^{[c]}$).
- v: Velocity term (same shape as θ).
- μ: Momentum coefficient, typically 0.9.
- η: Learning rate.
- $\tilde{\theta}_t$: Look-ahead parameters after applying momentum.
- g_t: Gradient of the loss function at the look-ahead parameters.

4.7 Early Stopping

The **Early Stopping** technique is a form of **regularization** used to prevent **overfitting** during training. The principle is to monitor the **validation loss** at each epoch and stop the training when the model no longer improves, instead of continuing until the maximum number of epochs is reached. This ensures that the model generalizes better and avoids fitting too closely to the training data.

Step 1 — Monitor validation loss:

$$\mathcal{L}_{val}^{(t)} = \text{Loss}(Y_{val}, \hat{Y}_{val}^{(t)})$$

Step 2 — Compare with best loss:

$$\mathcal{L}_{val}^{(t)} > \min_{k<t} \mathcal{L}_{val}^{(k)} \quad \Rightarrow \quad \text{no improvement}$$

Step 3 — Stop if patience exceeded:

if no improvement for p consecutive epochs: stop training

where:

- $\mathcal{L}_{val}^{(t)}$: Validation loss at epoch t.
- $\hat{Y}_{val}^{(t)}$: Model predictions on the validation set at epoch t.
- Y_{val}: True labels of the validation set.
- p: **Patience**, i.e., number of consecutive epochs without improvement tolerated before stopping.
- $\min_{k<t} \mathcal{L}_{val}^{(k)}$: Best validation loss observed so far.

5 Prediction Phase

The **Prediction** step uses the **trained parameters** to compute the network's **output probabilities** and the corresponding **class labels**. We first run a **forward pass** to obtain the final activations (softmax probabilities), then apply a decision rule (argmax) to select the most likely class.

Step 1 — Forward pass to output probabilities:

$$Z^{[C]} = W^{[C]} A^{[C-1]} + b^{[C]}, \qquad P = A^{[C]} = \text{softmax}\Big(Z^{[C]}\Big)$$

Step 2 — Class prediction (argmax):

$$\hat{y}_i = \underset{j \in \{1,\ldots,K\}}{\arg\max}\ P_{j,i}$$

Step 3 — Return format:

$$\hat{Y} = \big[\hat{y}_1,\ \hat{y}_2,\ \ldots,\ \hat{y}_m\big] \in \mathbb{N}^{1\times m}, \qquad P \in \mathbb{R}^{K\times m}$$

where:

- C: Index of the last (output) layer; K: number of classes; m: number of samples in the batch.
- $W^{[C]}, b^{[C]}$: Weights and biases of the output layer.
- $A^{[C-1]}$: Activations of the last hidden layer.
- $Z^{[C]}$: Pre-activation of the output layer.
- $P = A^{[C]}$: Softmax probabilities at the output layer, $P_{j,i} \in [0,1]$ and $\sum_{j=1}^{K} P_{j,i} = 1$.
- $\hat{y}_i$: Predicted class index for sample i (via argmax over classes).
- $\hat{Y}$: Row vector of predicted class indices (shape $1 \times m$).

Notes.

- **predict_proba** returns $P = A^{[C]}$ (the softmax probabilities).
- **predict** returns $\hat{Y}$ computed by the argmax decision rule.