

# **A short walk through machine learning in nuclear physics**

**Isaac Vidaña, INFN Catania**



**Selected Topics in Nuclear & Atomic  
Physics 2022  
September 25<sup>th</sup>-October 1<sup>st</sup> 2022  
Fiera di Primiero**

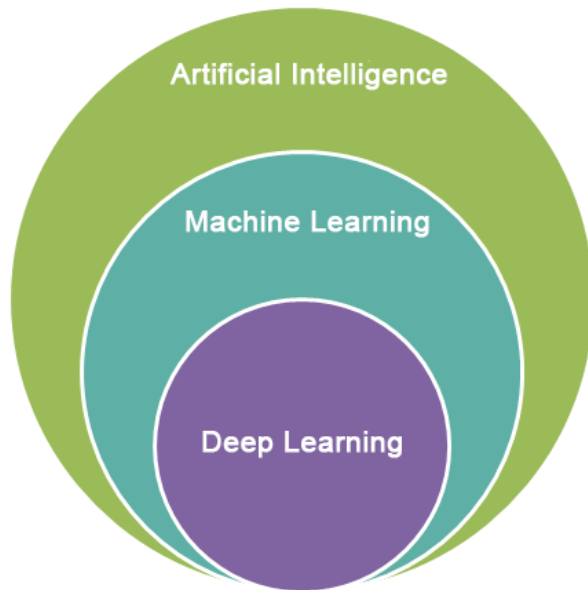


This short lecture is just a **brush-stroke** on the the **basic concepts & ideas** behind **Machine Learning** and its **applications to Nuclear Physics**. To review in a complete and detailed way this topic in the time of two hours of this lecture is basically an impossible task and, therefore, all of you interested is referred to the several excellent books and many reviews that comprehensively cover all different aspects of this fascinating field such as, *e.g.*



1. C. M. Bishop, *Pattern Recognition and Machine Learning*, (Springer 2006).
2. T. Hastie, R. Tibshirami, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (Springer Verlag, Berlin), (2009).
3. K. P. Murphy, *Machine Learning: A Probabilistic Perspective* (The MIT Press, Cambridge, Massachusetts), (2012).
4. Y. LeCun, Y. Bengio, and G. Hinton, *Nature* 521 (2015) 436.
5. I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning (Adaptive Computation and Machine Learning series)*, The MIT Press, Cambridge, Massachusetts, ISBN 9780262035613 (2016).

# Artificial Intelligence, Machine Learning & Deep Learning



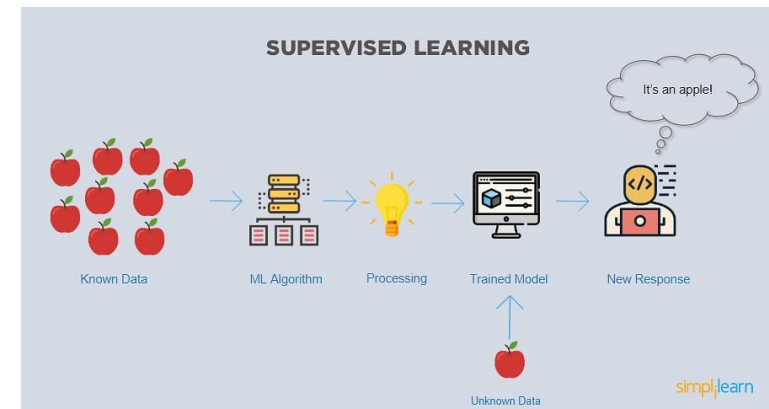
- **Artificial Intelligence** (AI) has become one of the most **exciting & dynamic** areas of research impacting many domains of **science & technology**. The main challenge of AI is:

*Develop algorithms that can sense, reason, act & adapt to solve different type of problems*

- **Machine Learning** is a branch of Artificial Intelligence whose scope is to devise algorithms able to **recognize patterns in previously unseen data without any explicit instructions by an external party**. Different types of ML include
  - **Supervised Learning:** the training dataset contains both the inputs & the desired outputs
  - **Unsupervised Learning:** the training dataset contains only the inputs & finds structures in the data
  - **Reinforcement Learning:** the algorithm learns on its own through a balance between exploration (of the unknown) and exploitation (of the current knowledge)
- **Deep learning** is a subset of machine learning in which multilayer neural networks adapt & learn from vast amounts of data

# Supervised Learning

In supervised learning, **known input-output (feature-label) relations** are given to the machine learning algorithm to **trained** it and **infer a mapping therefrom**. Once the model is **trained based on the known data**, one can use unknown data into the model to get predictions

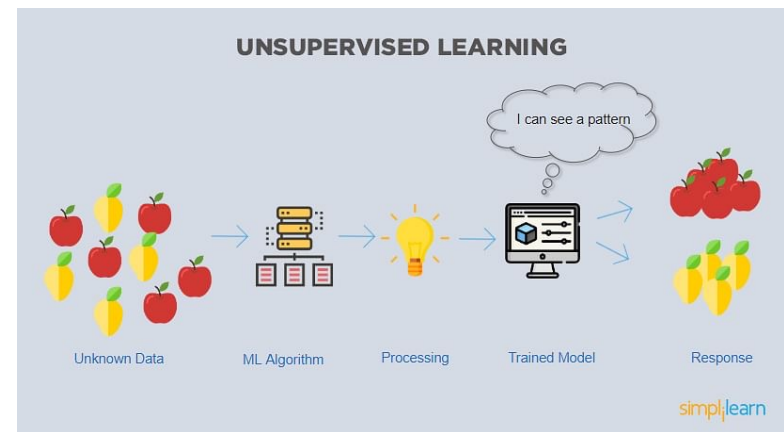


Two common tasks of supervised learning are:

1. **Classification:** used when the output variable is discrete. Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes
2. **Regression:** used when the output variable is continuous. The goal is to find a function that maps input data into continuous output values

# Unsupervised Learning

In unsupervised learning, the output of the input training data is **unknown**. The input data is fed to the Machine Learning algorithm and is used to train the model which then is employed to **search for patterns in the data**



Two common tasks of unsupervised learning are:

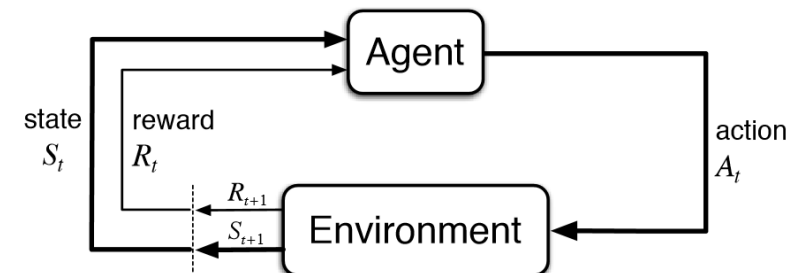
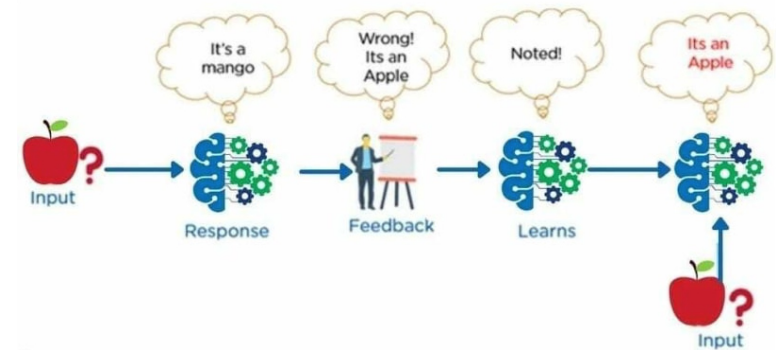
1. **Clustering** : Data are divided into groups with certain common traits, without knowing the different groups beforehand
2. **Generation**: Building a model to generate data that are similar to a training dataset in both examples and distributions of examples

# Reinforcement Learning

**Reinforcement learning** is the closest to what we might associate with the expression ‘learning’. Given a **framework of rules and goals**, an **agent** (algorithm) **learns in an interactive environment** by **trial and error** using **feedback from its own actions and experiences** and it gets **rewarded** or **punished** depending on which strategy it uses. Each **reward** reinforces the current strategy, while **punishment** leads to an adaptation of its policy

There are **five key elements of reinforcement learning** models:

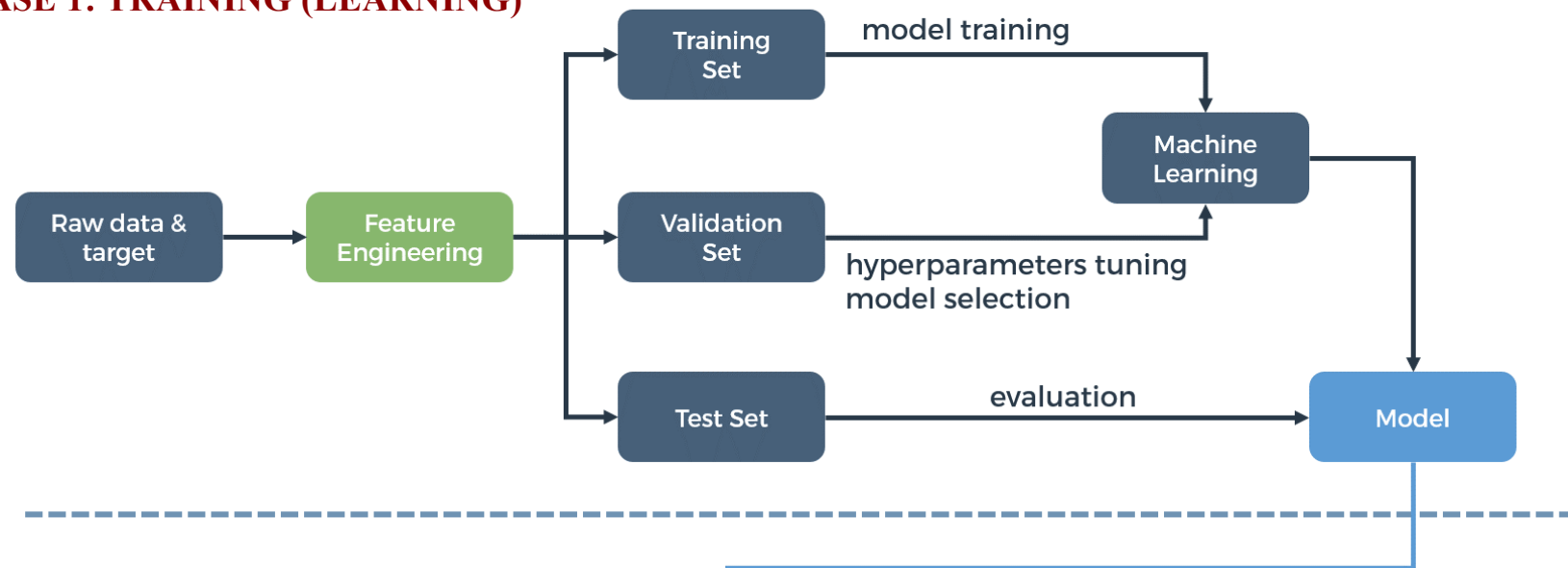
- **Agent:** The algorithm in the model that performs the requested task
- **Environments:** The “world” in which the agent carries out its actions. It uses current states and actions of the agent as input, rewards and next states of the agents as output
- **States:** It refers to the situation of the agent in an environment. There are current and future/next states
- **Actions:** The movement chosen and performed by the agent to gain rewards
- **Rewards:** Reward means desired behaviours which are expected from the agent. Rewards also refer to the feedback for the agent’s actions in a given state



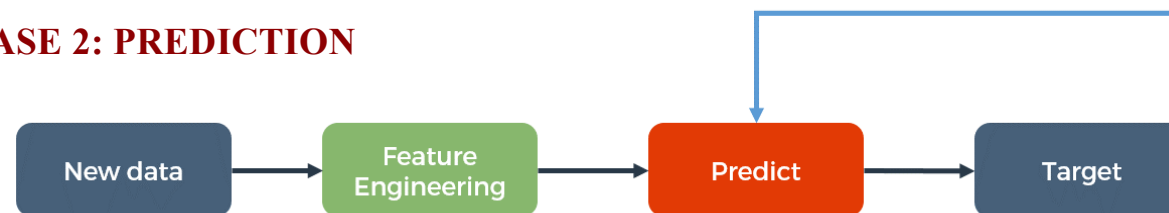
# Machine Learning Process: General Scheme

The task of **making a machine to learn** is made of **2 phases**

## PHASE 1: TRAINING (LEARNING)



## PHASE 2: PREDICTION





# Machine Learning Steps

**7 major steps (6 in training phase & 1 in the prediction one )** can be distinguished in the Machine Learning process

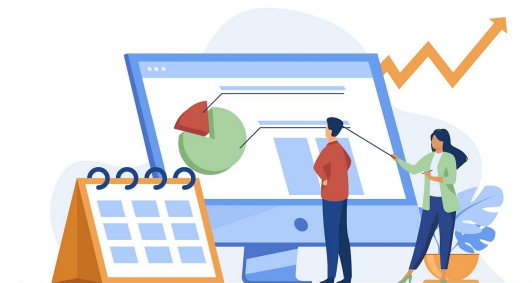
## 1. Collecting Data

Machines **learn from the data** given to them. Therefore, it is of the utmost importance to **collect reliable data** so that a machine learning model can find the **correct patterns**. The quality of the data feeded to the machine will determine how accurate the model will be. If data is incorrect or outdated, the machine will have wrong outcomes or predictions which are not relevant



## 2. Preparing the Data (Feature Engineering)

- **Clean the data** to remove unwanted data, missing values, rows, and columns, duplicate values, data type conversion, etc
- **Visualize the data** to understand how it is structured and understand the relationship between various variables and classes present
- **Split the cleaned data** into two sets - a training set and a testing set. The **training set** is the **set a model learns from**. A **testing set** is used to check the accuracy of a model after training

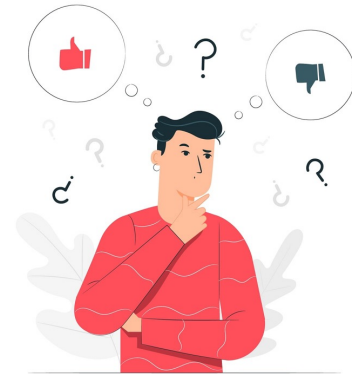




# Machine Learning Steps

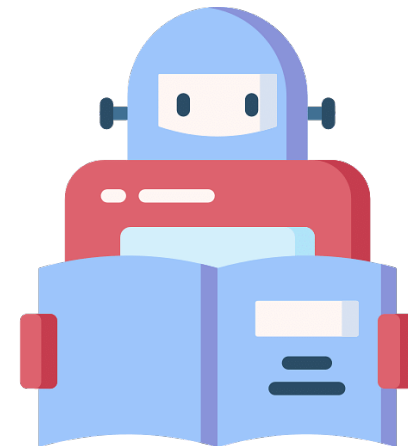
## 3. Choosing a Model

A machine learning model determines the output one gets after running a **machine learning algorithm** on the collected data. It is important to **choose a model which is relevant to the task at hand**, e.g., one has to consider if the model is suited for numerical or categorical data and choose accordingly



## 4. Training the Model

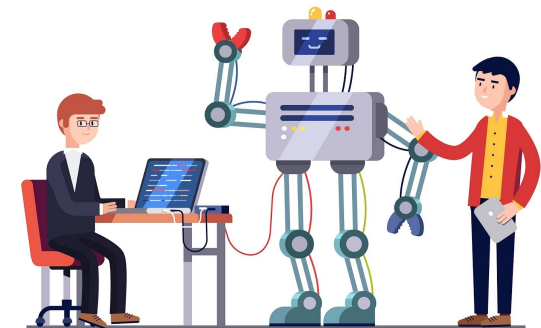
Training is the **most important step** in machine learning. In training, one **passes the prepared data to the machine learning model to find patterns and make predictions**. It results in the model learning from the data so that it can accomplish the task set. Over time, with training, the model gets better at predicting



# Machine Learning Steps

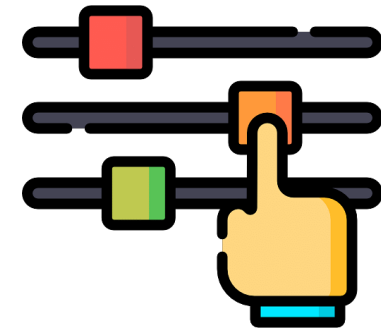
## 5. Evaluating the Model

After training a model, one has to **check to see how it is performing**. This is done by **testing the performance of the model on previously unseen data**. The unseen data used is the testing set. If testing was done on the same data which is used for training, one will not get an accurate measure, as the model is already used to the data, and finds the same patterns in it, as it previously did



## 6. Hyperparameter Tuning

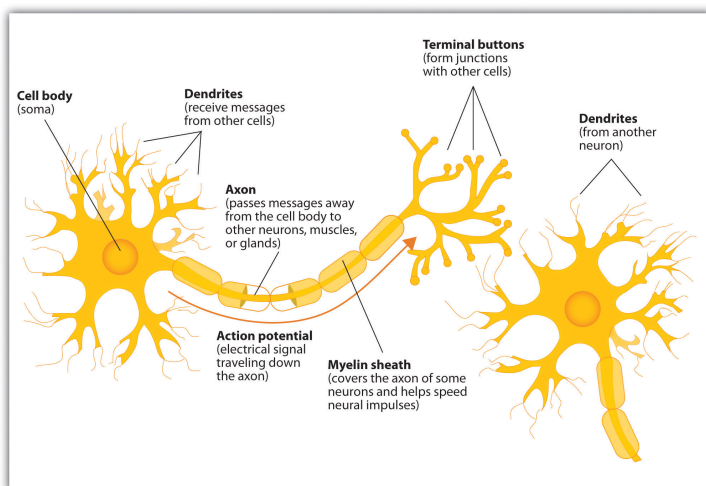
Once one has created and evaluated a model, one has to **see if its accuracy can be improved in any way**. This is done by **tuning the hyperparameters present in your model**. Hyperparameters are the variables in the model (*e.g.*, number of hidden layers & neurons in an artificial neural network). At a particular value of a hyperparameter, the accuracy will be the maximum. Hyperparameter tuning refers to finding these values



## 7. Making Predictions

In the end, one can **use the model on unseen data** to make predictions accurately

## Artificial Neurons – a brief glimpse into the early history of ML



Biological Neurons are **interconnected nerve cells** in the brain that are involved in processing and transmitting chemical and electrical signals. Their main parts are:

- **Dendrites** are branches that receive information from other neurons
- **Cell nucleus** or **Soma** processes the information received from dendrites
- **Axon** is a “cable” that is used by neurons to send information
- **Synapse** is the connection between an axon and other neuron dendrites.

- Trying to understand how the brain works, Warren McCulloch and Walter Pitts introduced the first artificial neuron, the so-called **McCulloch-Pitts (MCP) neuron** in 1943. They thought of it as a **simple logic gate with binary outputs**: multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon
- A few years later, in 1957, Frank Rosenblatt published the first concept of the **perceptron learning rule** based on the MCP neuron model. Rosenblatt **proposed an algorithm** that would **automatically learn** the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not

## Formal Definition of an Artificial Neuron

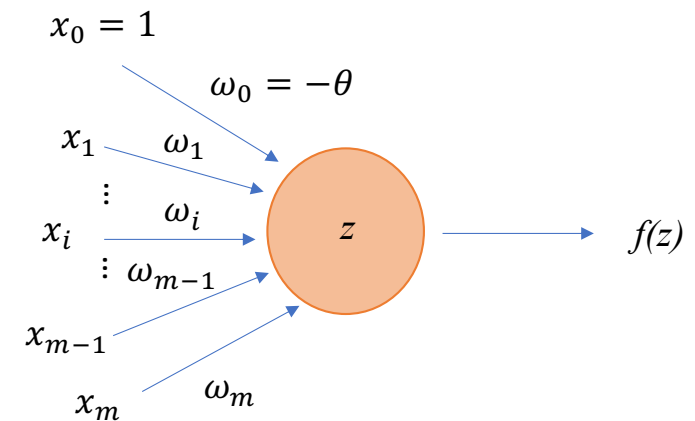
- More formally, we can put the idea behind **artificial neurons** into the context of a **binary classification task** where we refer to our two classes as **1** (positive class) and **-1** (negative class) for simplicity
- We can then define an **activation function**  $f(z)$  that takes a linear combination  $z = \sum_{i=1}^m \omega_i x_i$  (named **net input**) of certain **input values**  $\vec{x} = (x_1, x_2, \dots, x_m)$  where the coefficients  $\omega_i$  are the so-called **weights** and it gives **1** or **-1** if the net input of a particular input value is larger or smaller than a certain defined **threshold  $\theta$**

$$f(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

- For simplicity, we can bring the threshold  $\theta$  to the left side of the equation and define a weight-zero as  $\omega_0 = -\theta$  and  $x_0 = 1$  so that we can write

$$z = \sum_{i=0}^m \omega_i x_i \quad \text{and} \quad f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

- In the Machine Learning literature the negative threshold or weight  $\omega_0 = -\theta$  is usually called the **bias unit** and it is represented with the letter **b**



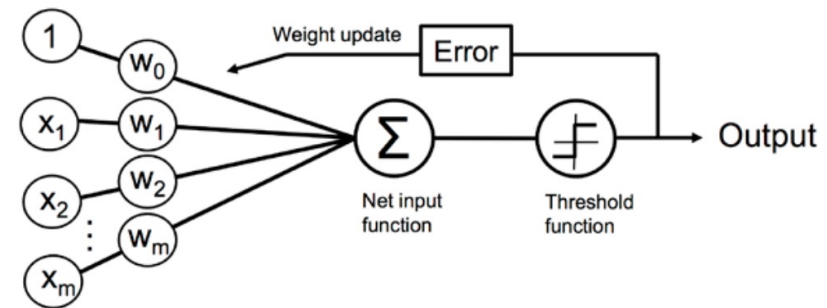
Biological Neuron	Artificial Neuron
Cell nucleus (Soma)	Node
Dendrites	Input
Synapse	Weights or interconnections
Axon	Output

# The Perceptron Learning Rule

The whole idea behind the *MCP neuron* and *Rosenblatt's thresholded perceptron model* is to use a reductionist approach to mimic how a single neuron in the brain works: *it either fires or it doesn't*. Thus, Rosenblatt's initial perceptron rule is fairly simple, and the *perceptron algorithm* can be summarized by the following steps and illustrated by the diagram:

1. Initialize the weights to 0 or small random numbers
2. For each training example  $\vec{x}$ 
  - a. Compute the output value  $\hat{y}$
  - b. Update the weights according to the *perception learning rule*

$$\omega_j \rightarrow \omega_j + \eta(y - \hat{y})x_j$$



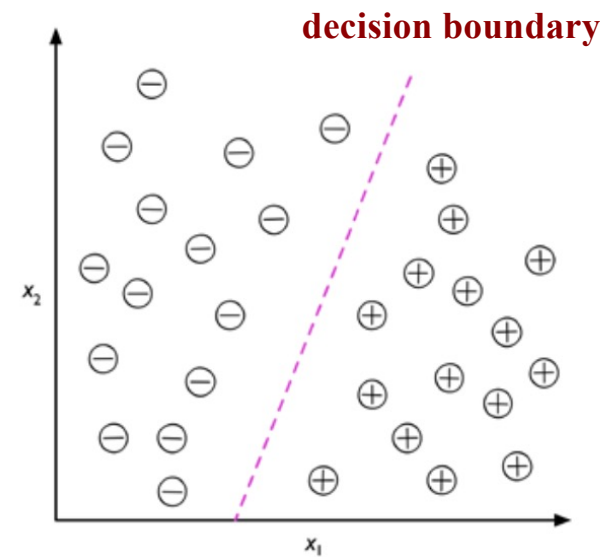
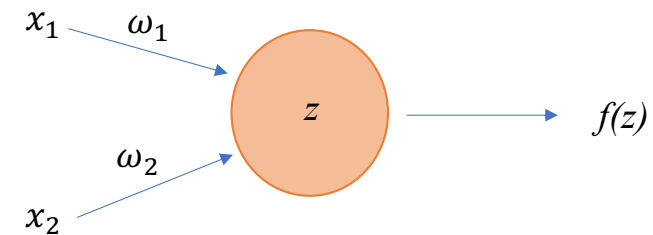
Here  $\eta$  is the *learning rate* (a constant between 0 and 1),  $y$  is the *true class label* and  $\hat{y}$  the *predicted class label*

## Example of a Binary Classification Task

- Consider 30 training examples: 15 of which are labeled as **negative class (-)** and the other 15 as **positive class (+)**
- The dataset is **two-dimensional**, which means that each training example has two values associated to it:  $x_1$  and  $x_2$
- Applying the Perceptron Learning Rule is possible to find the **decision boundary**

$$\omega_1 x_1 + \omega_2 x_2 + b = 0$$

that can **separate those two classes** and will allow to **classify new data** into each of those two categories given its values  $x_1$  and  $x_2$



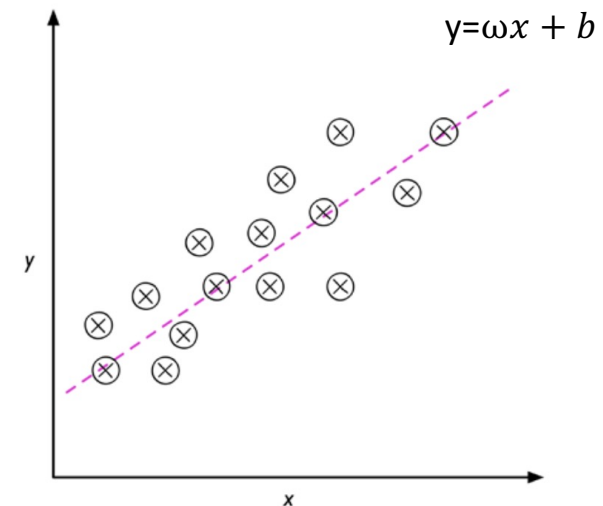
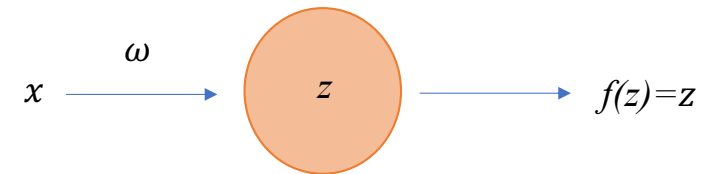
## Preceptron & Linear Regression

- The perceptron can be used to **predict the outcome of (linear) continuous functions** if the **identity function**  $f(z) = z$  is used as **activation function** instead of the unit step function
- Given a set of **features** variables  $x$  and their corresponding **target** values  $y$  that are linearly related

$x_1$	$y_1$
$x_2$	$y_2$
...	...
$x_m$	$y_m$

By applying the Perceptron Learning Rule is possible to find the **straight line** that minimizes the distance – most commonly the average square distance – between the data points and the fitted line and make predictions for new data

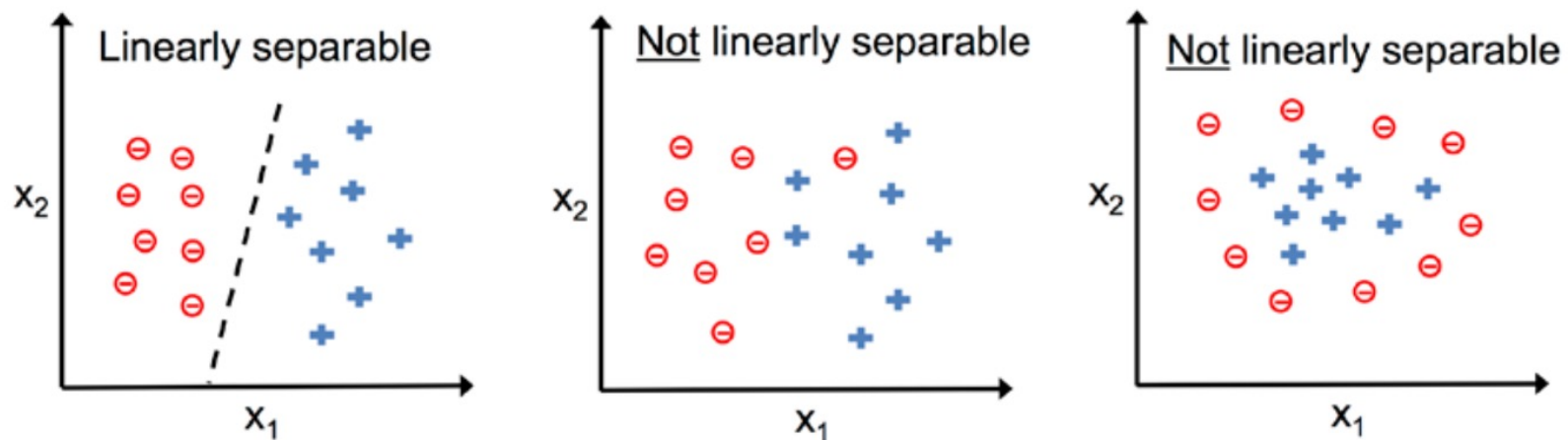
$$y = \omega x + b$$





## Convergence of the Perceptron Algorithm

It is important to note that the **convergence** of the perceptron is **only guaranteed if the two classes are linearly separable** and the **learning rate is sufficiently small**. If the two classes cannot be separated by a linear decision boundary the perceptron **would never stop updating the weights**. In this case, we can set a maximum number of passes over the training dataset (**epochs or iterations**) and/or a threshold for the number of tolerated misclassifications



The solution, as we will see later, will imply the introduction of **non-linear activation functions**

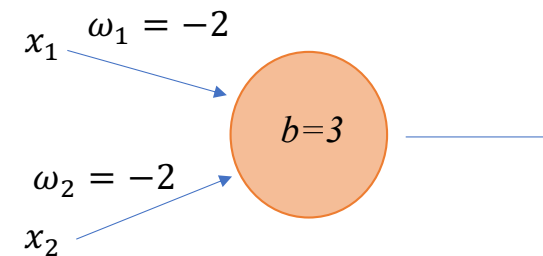
## Perceptron & Logical Functions

An interesting thing about **perceptrons** is that they can be **used to compute** the **elementary logical functions**

- Suppose we have a perceptron with two inputs  $x_1$  and  $x_2$ , each with weight  $-2$ , which can take values 0 and 1, and that the overall bias is 3

The perceptron produces the following results:

$x_1$	$x_2$	Perceptron's result
0	0	1
0	1	1
1	0	1
1	1	0

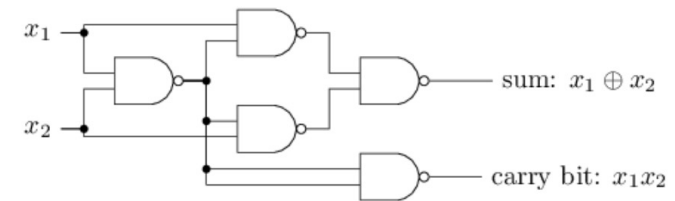


But that is nothing else than the **truth table** of the logical **NAND** gate. Therefore, this perceptron implements a **NAND** gate

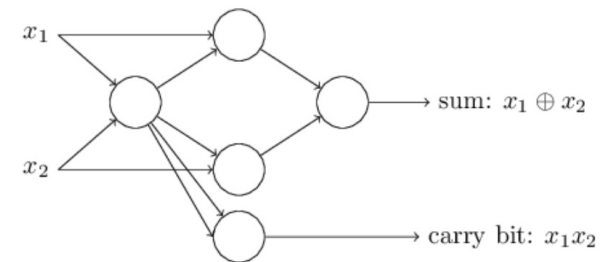
## Perceptron & Logical Functions

Now since the **NAND** gate is **universal for computation**, *i.e.*, we can build any computation up out of NAND gates, we can use **networks of perceptrons** to compute **any logical function** at all.

For example, we can use **NAND** gates to build a circuit which adds two bits,  $x_1$  and  $x_2$ . This requires computing the bitwise sum,  $x_1 \oplus x_2$ , as well as a carry bit which is set to 1 when both  $x_1$  and  $x_2$  are 1, *i.e.*, the carry bit is just the bitwise product  $x_1 x_2$ :



To get an equivalent network of perceptrons we replace all the **NAND** gates by **perceptrons with two inputs**, each with weight  $-2$ , and an overall bias of 3.



# General Architecture of an Artificial Neural Network

The architecture of an artificial neural network (ANN) consists of an **input layer**, one or more **hidden layers**, and an **output layer** of several **interconnected artificial neurons**

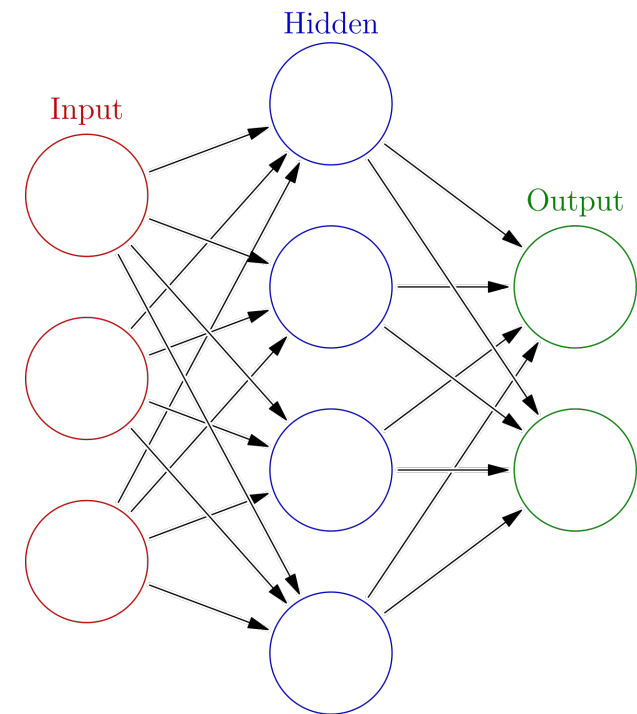
- **Input layer:** Made up of those neurons that introduce input patterns into the network. No processing occurs in these neurons
- **Hidden layers:** Formed by those neurons whose inputs come from previous layers and whose outputs pass to neurons of later layers
- **Output layer:** Made up by the neurons whose output values correspond to the outputs of the entire network

## Terminology:

- $\omega_{jk}^l$ : **weight** between the neuron  $k$ -th in the layer  $(l-1)$ -th and the neuron  $j$ -th in the layer  $l$ -th
- $b_j^l$ : **bias** of neuron  $j$ -th in the layer  $l$ -th
- $a_j^l = f(z_j^l)$ : **activation** (output) of neuron  $j$ -th in the layer  $l$ -th (note that  $a_j^l = \hat{y}_i$ )
- $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$ : **weighted input to the activation function** of neuron  $j$ -th in the layer  $l$ -th

Total number of fitting parameters:

$$n_p = \sum_{k=1}^{L-1} (N_k + 1) N_{k+1}$$



# A mostly complete chart of neural networks

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

Perceptron (P)



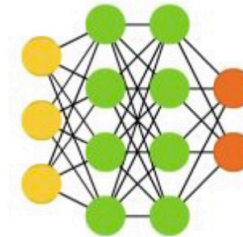
Feed Forward (FF)



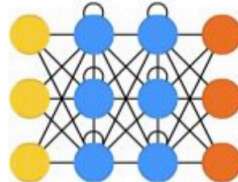
Radial Basis Network (RBF)



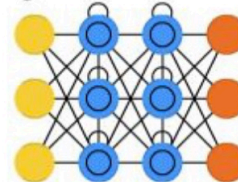
Deep Feed Forward (DFF)



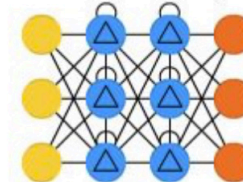
Recurrent Neural Network (RNN)



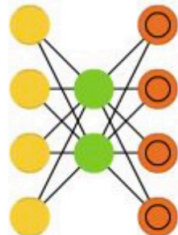
Long / Short Term Memory (LSTM)



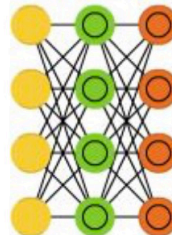
Gated Recurrent Unit (GRU)



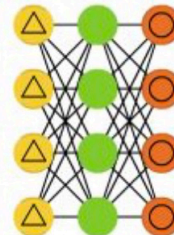
Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)

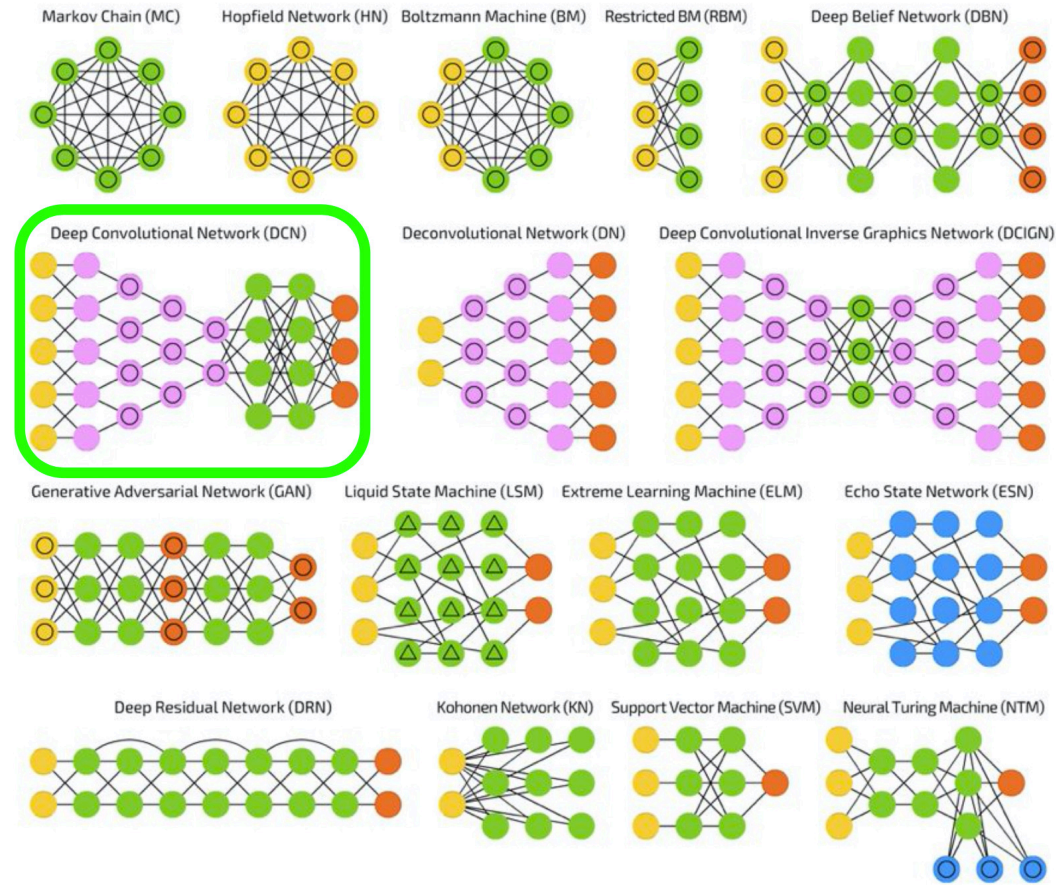


Sparse AE (SAE)



## A mostly complete chart of neural networks

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool





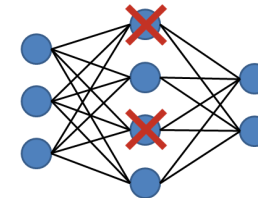
# Neural Network Hyperparameters

**Hyperparameters** are the **variables which determine the network structure** (e.g., number of hidden layers and neurons, type of regularization technique, initializatial values of weights & biases, type of activation function ... ) and the **variables which determine how the network is trained** (e.g., learning rate, number of epochs (iterations), bach size, ...). They are **set before the training** of the network

- **Hyperparameters related to the network structure**

- **Number of hidden layers & neurons :** Many hidden layers and neurons layer can increase accuracy. Smaller number of hidden layers and neurons may cause **underfitting**

- **Dropout:** is a regularization technique to avoid **overfitting** (seen later). It consist on dropping randomly neurons from the neural network during training in each iteration. The **number of dropped neurons** is another hyperparameter



- **Initial values of weights & biases:** different weight initialization schemes can be used to start the training

- **Type of activation function:** different types of activation function (seen later) can be used to introduce non-linearities

- **Hyperparameters related to the training of the network**

- **Learning rate:** defines how quickly a network updates its parameters

- **Number of epochs or iterations:** is the number of times the whole training data is shown to the network while training

- **Batch size:** is the number of samples given to the network after which parameter update happens



## Activation Functions

- As mentioned before one of the **limitations of the Perceptron Algorithm** is that it **only converges for linear** (classification or regression) **problems**. The same is true for any **ANN made of perceptrons** since, it will be simply **a linear combination of perceptrons**
- In order to pick up the **non-linearities of the input data** that enable ANN to capture **complex non-linear relationships in the dataset** and make new predictions, ANN should employ **non-linear activation functions**
- There **exist several possible choices** for the activation function **depending on the particular problem** one is trying to solve. The choice of the activation function has a large impact on the capability and performance of the neural network, and **different activation functions may be used in different parts of the model**. All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model
- Activation functions are typically **differentiable** (*i.e.*, the first-order derivative can be calculated for a given input value). This is required given that neural networks are typically trained using the **backpropagation of error algorithm** that requires the derivative of prediction error in order to update the weights of the model
- There are many **different types of activation functions** used in neural networks, **although** perhaps only **a small number of functions used in practice for hidden and output layer**

# ReLU Hidden Layer Activation Function

The **Rectified Linear Activation (ReLU)** function is perhaps the most common function used for hidden layers because it is **simply to implement** and is **able to overcome some limitations** of other activations functions. It is simply given by

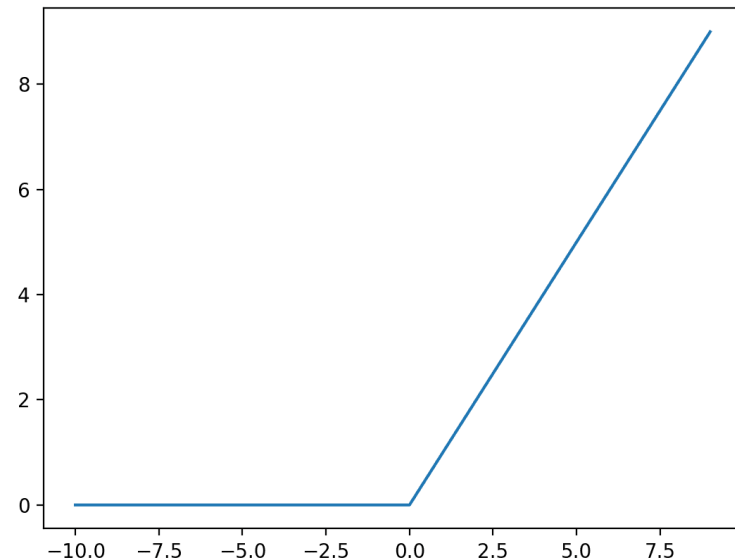
$$f(z) = \max(0, z)$$

## Advantages:

- **Fewer vanishing gradient** problems that slow the learning process
- **Efficient computation**: only comparison
- **Scale-invariant**:  $\max(0, az) = a \max(0, z)$  for  $a \geq 0$

## Potential Problems:

- **Not differentiable at  $z=0$** : the derivative at  $z=0$  can be arbitrarily chosen to be 0 or 1
- **Not zero-centered**
- **Unbounded**
- **Dying ReLU Problem**: neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. In this state, no gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state and **"dies"**



# Sigmoid Hidden Layer Activation Function

The **Sigmoid** activation function **takes any real value as input** and **outputs values in the range 0 to 1**. The larger the input (more positive), the closer the output value will be to 1, whereas the smaller the input (more negative), the closer the output will be to 0. It is given by the expression

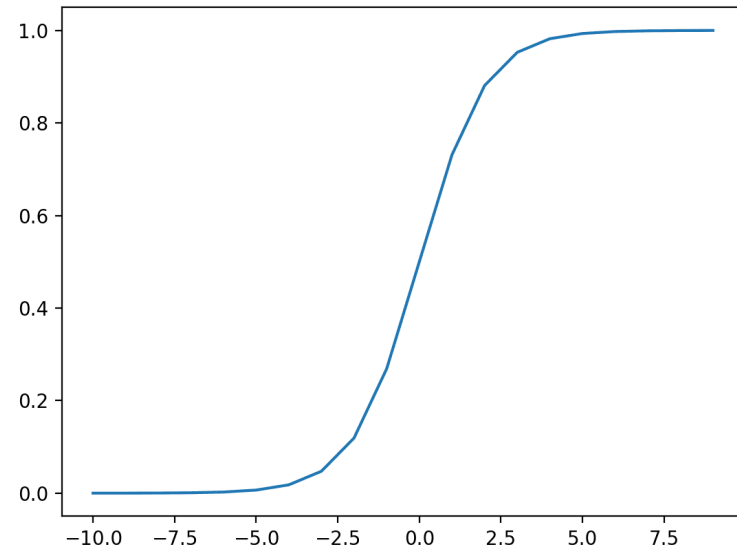
$$f(z) = \frac{1}{1 + e^{-z}}$$

## Advantages:

- **Differentiable at each value  $z$**  :  $f'(z) = f(z)(1 - f(z))$
- **Zero-centered**
- **Bound** in the range  $[0,1]$

## Potential Problems:

- **Vanishing gradient**: for values of  $|z| > 5$  its gradient becomes very small which can lead to a slow learning of the neural network



# Tanh Hidden Layer Activation Function

The **hyperbolic tangent (Tanh)** activation function is very similar to the **sigmoid one**. It also **takes any real value as input** and **outputs values in the range 0 to 1**. It is given by

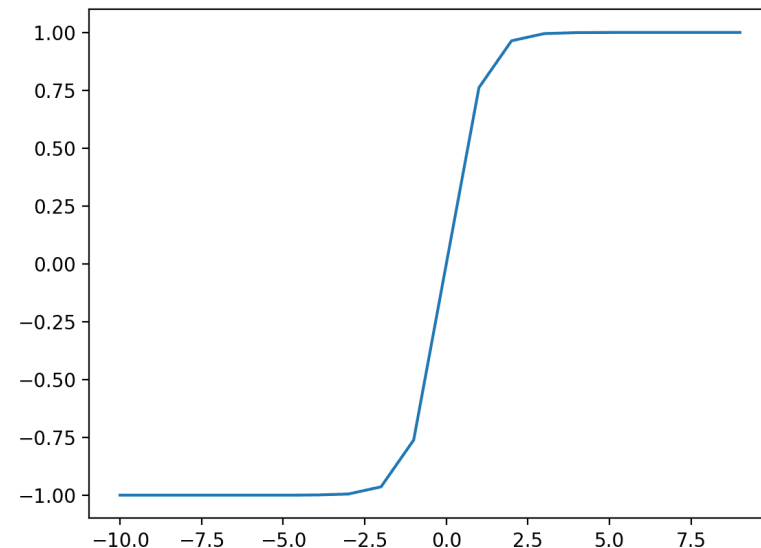
$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

## Advantages:

- **Differentiable at each value  $z$  :**  $f'(z) = 1 - f(z)^2$
- **Zero-centered**
- **Bound** in the range  $[0,1]$

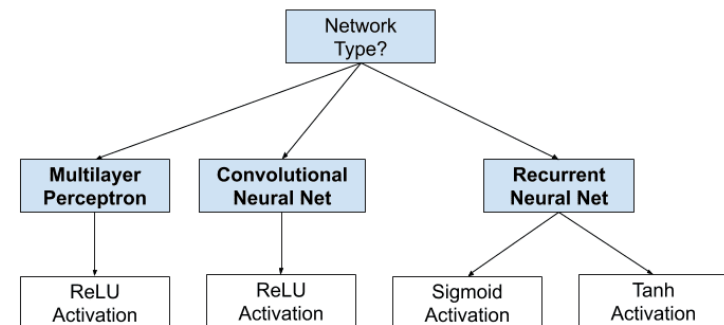
## Potential Problems:

- **Vanishing gradient:** for values of  $|z| > 5$  its gradient becomes very small which can lead to a slow learning of the neural network



## How to Choose a Hidden Layer Activation Functions

- As mentioned before a neural network **will almost always have the same activation function** in all hidden layers. And is typically **chosen based on the type of neural network architecture**
- Traditionally, the **sigmoid activation function** was the default activation function in the 1990s. Perhaps through the mid to late 1990s to 2010s, the **Tanh function** was the default activation function for hidden layers
- Modern neural network models with common architectures, such as **Multilayer Perceptron (MLP)** and **Convolutional Neural Networks (CNN)** make use of the **ReLU activation function**, or extensions of it not mentioned here. **Recurrent networks** still commonly use **Tanh** or **sigmoid activation functions**, or even both.



# Output Layer Activation Functions

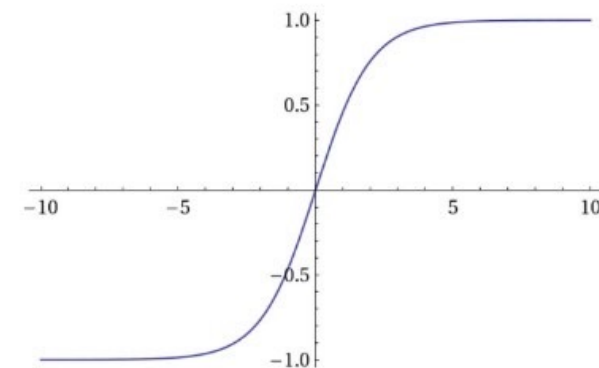
The **output layer** is the layer in a neural network model that directly **outputs a prediction**. Typical output layer activation functions include:

- **Linear or identity activation function:**  $f(z)=z$  (already mentioned before)
- **Sigmoid activation function**
- **Softmax activation function**

**Softmax activation function:**

Is a kind of **generalization of the sigmoid activation function** given by

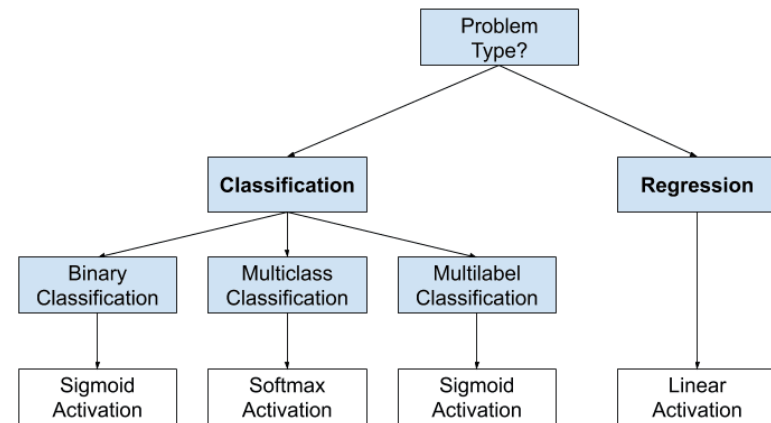
$$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$



# How to Choose a Output Layer Activation Functions

You must **choose the activation function** for your **output layer** based on the **type of prediction problem** that you are solving:

- If your problem is a **regression problem**, you should use a **linear activation function**
- If your problem is a **classification problem**, then there are **three main types of classification problems** and **each may use a different activation function**
  - ✓ If there are two mutually exclusive classes (**binary classification**), then your output layer will have one node and a **sigmoid activation function** should be used
  - ✓ If there are more than two mutually exclusive classes (**multiclass classification**), then your output layer will have one node per class and a **softmax activation** should be used
  - ✓ If there are two or more mutually inclusive classes (**multilabel classification**), then your output layer will have one node for each class and a **sigmoid activation function** is used





# The Learning Process of an ANN

The **learning (or training) process of an ANN** involves the **minimization of a cost** (also called **loss** or **error**) **function** (which compares the desired output (target) and the predicted one by the ANN) in order to **obtain the optimal set of fitting parameters** (**weights** and **biases**) of the network. The minimization is usually done by using **algorithms** such as the so-called **gradient descent**

## Choice of a Cost Function

In general, the **choice of the cost function depends on the type of problem** one is solving with a neural network. In supervised learning, there are **two main types** of cost functions:

- **Regression Cost Functions** — used when solving a regression problem. Two examples of them are the *Mean Squared Error*, the *Mean Absolute Error*
- **Classification Cost Functions** — used when solving a classification problem. Among these type we can distinguish the *Binary Cross-Entropy* and the *Categorical Cross-Entropy*

## Cost Functions

### ■ Regression Cost Functions

#### ✓ Mean Squared Error (MSE)

$$C_{MSE} = \frac{1}{N} \sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2$$

#### ✓ Mean Absolute Error (MAE)

$$C_{MAE} = \frac{1}{N} \sum_{i=1}^N |y^{(i)} - \hat{y}^{(i)}|$$

- One of the most popular cost functions
  - Being difference between the predicted ( $\hat{y}^{(i)}$ ) and the target ( $y^{(i)}$ ) values squared, it does not matter whether the predicted values are above or below the target ones
  - Is a **convex function** with a clearly defined global minimum. This allows a **more easily use** of the **gradient decent optimization algorithm**
  - However, it is **very sensitive to the outliers**; if a predicted value is significantly greater than or less than its target value, this will significantly increase the cost
- 
- Used as an **alternative** of the MSE for instance when the training data has a large number of outliers to mitigate the increase of the cost due to these values
  - However, as the average distance approaches 0, **gradient descent optimization will not work**, as the function's derivative at 0 is undefined

## Cost Functions

### ■ Classification Cost Functions

#### ✓ Binary Cross-Entropy

$$C_{BCE} = -\frac{1}{N} \sum_{i=1}^N (y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i)))$$

#### ✓ Categorical Cross-Entropy

$$C_{CCE} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p(y_{ij}))$$

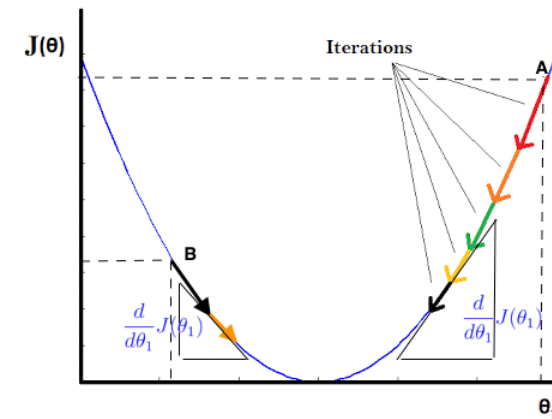
- Is the cost function used in binary classification models
- Classification neural networks work by outputting a vector of probabilities (the probability that the given input fits into each of the pre-set categories) then selecting the category with the highest probability as the final output
- In binary classification, there are only two possible actual values of y: 0 or 1. Thus, to accurately determine the cost between the actual and predicted values, it needs to compare the actual value (0 or 1) with the probability that the input aligns with that category ( $p(i)$  = probability that the category is 1;  $1 - p(i)$  = probability that the category is 0)
- It is the generalization of the Binary Cross-Entropy to the case of a multi-class classification problem

## Batch Gradient Descent

**Batch Gradient Descent** or simply **Gradient Descent** is an **iterative optimization algorithm** for finding a **local minimum** of a **differentiable function**

Idea: Take repeated steps in the opposite direction of the gradient since the **gradient of a multi-variable function  $F(\vec{x})$  defines the direction of its maximum increase**. One starts with a guess  $\vec{x}_0$  and considers the sequence  $\vec{x}_1, \vec{x}_2, \vec{x}_3, \dots$  according to

$$\vec{x}_{n+1} = \vec{x}_n - \eta \vec{\nabla} F(\vec{x}_n), \text{ with } \eta > 0$$



With this idea in mind the **weights  $\omega_{jk}^l$  & biases  $b_j^l$**  of the network are updated at each iteration according to:

$$\omega_{jk}^l \rightarrow \omega_{jk}^l - \eta \frac{\partial C}{\partial \omega_{jk}^l}, \quad b_j^l \rightarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l}$$

where  $\eta$  is the so-called **learning rate**, one of the **hyperparameters** of the newtwork, and it scales the magnitude of the weighs and biases updates

# Batch Gradient Descent Algorithm

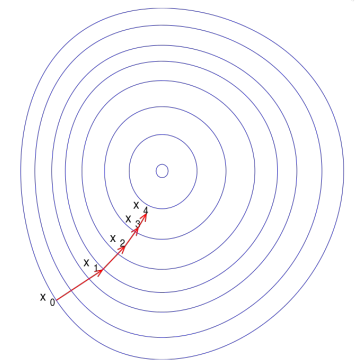
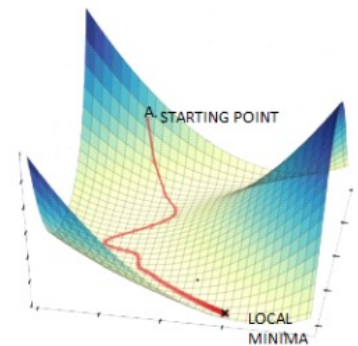
The Batch Gradient Descent Algorithm is quite simple. For **each epoch** (or **iteration**) of the training do the following steps:

1. Feed the network with the **entire training input dataset**  $\vec{x}$

2. Calculate the cost function and update the weights & biases

$$\omega_{jk}^l \rightarrow \omega_{jk}^l - \eta \frac{\partial C}{\partial \omega_{jk}^l}, \quad b_j^l \rightarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l}$$

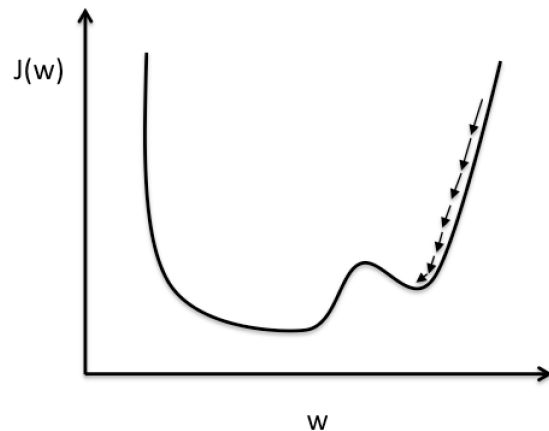
3. Repeat steps 1 – 2 until the convergence the cost function is substantially reduced



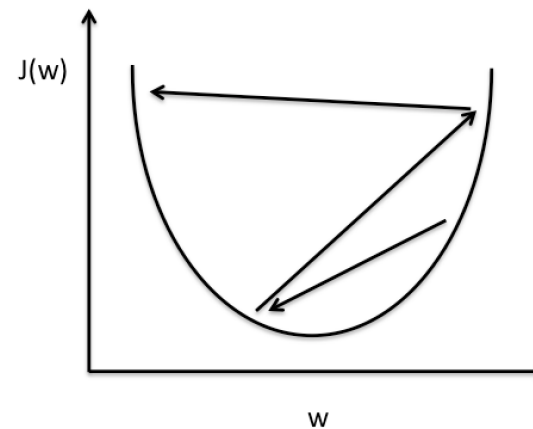
## A comment in the learning rate $\eta$

A proper value of  $\eta$  plays a crucial role in gradient descent

- Choose  $\eta$  **too small** and the algorithm **will converge very slowly** or **get stuck in the local minima**
- Choose  $\eta$  **too big** and the algorithm **will never converge** either because it **will oscillate between around the minima** or it **will diverge by overshooting the range**

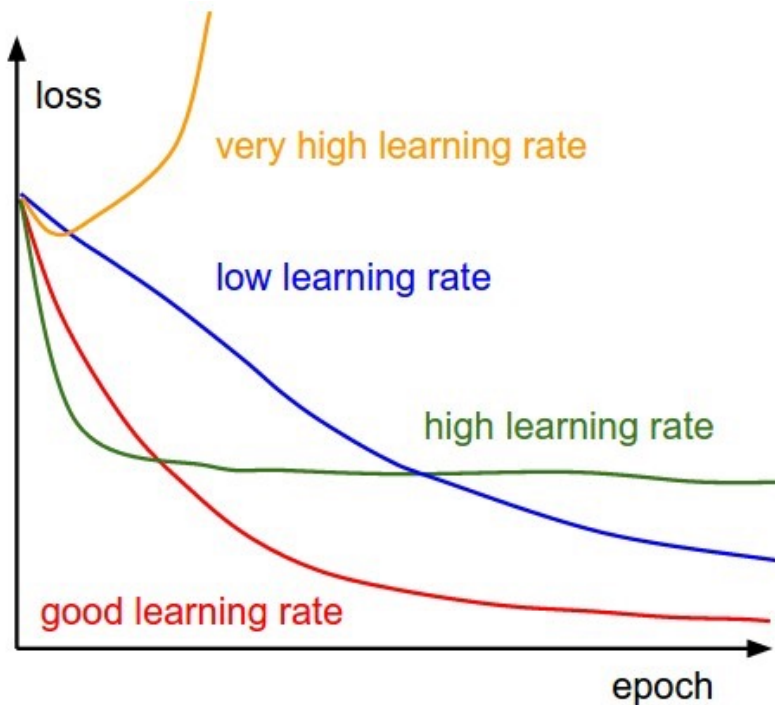


Small learning rate: Many iterations until convergence and trapping in local minima.



Large learning rate: Overshooting.

# Effect of the learning rate $\eta$ in the convergence of the Gradient Descent Algorithm



This figure tries to summarize the effect of  $\eta$  on the **convergence** of the gradient descent algorithm

- The **yellow** curve shows the **divergence** of the algorithm when the learning rate is really high wherein the learning steps overshoot.
- The **green** curve shows the case where learning rate is not as large as the previous case but is high enough that the steps keep **oscillating** at a point which is not the minima.
- The **red** curve would be the **optimum curve** for the cost drop as it drops steeply initially and then saturates very close to the optimum value.
- The **blue** curve is the least value of  $\eta$  and **converges very slowly** as the steps taken by the algorithm during update steps are very small.



# Stochastic Gradient Descent

- In general, when computing the cost function we look at the loss associated with each training example and then sum these values together for an overall cost of the **entire dataset**. This is the most basic form of gradient descent, also known as **batch gradient descent** since we compute the cost in **one large batch computation**
- However, for big datasets **this can take a long time to compute**. Moreover, it raises the question: **do we really need to see all of the data before making improvements to our parameter values ?**
- A way to deal with big datasets is **to split the training data** into **mini batches** which can be processed individually. When each mini batch is processed, we look at the cost function relative to the data from the current mini batch and update our parameters accordingly. Then we continue to do this iterating over all of the mini batches until we have seen the whole dataset. One full cycle through the dataset is referred to as an **epoch or iteration**. This is the idea behind of the so-called **stochastic gradient descent**

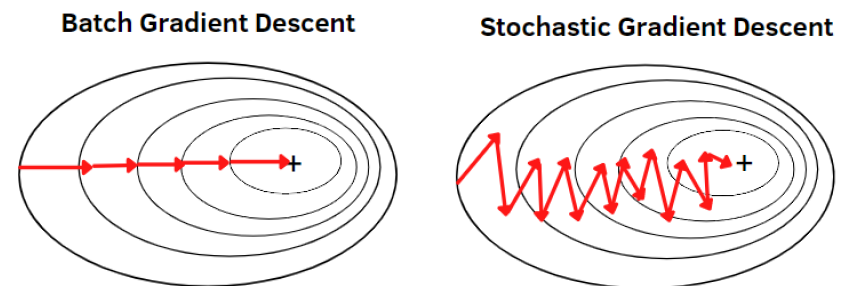
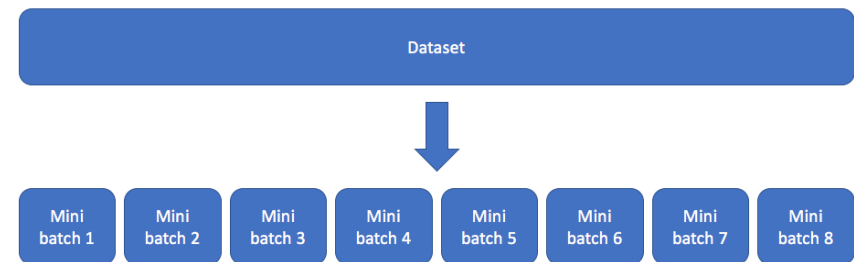
# Stochastic Gradient Descent

The idea is to **split the training dataset** into small subsets of **training inputs**  $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m$  each one known as **mini-batch** and **for a given epoch or iteration** do the following steps:

1. Pick randomly a mini-batch of size  $m$ :  $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m$
2. Feed it to the network
3. Calculate the average gradient of the cost function of the mini-batch and update the weights & biases

$$\omega_{jk}^l \rightarrow \omega_{jk}^l - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{\vec{x}_i}}{\partial \omega_{jk}^l}, \quad b_j^l \rightarrow b_j^l - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{\vec{x}_i}}{\partial b_j^l}$$

4. Repeat steps 1 – 3 until all the training inputs are exhausted which is said to complete an epoch (iteration) of training. At that point we start with a new training epoch



## Advanced Optimization Techniques

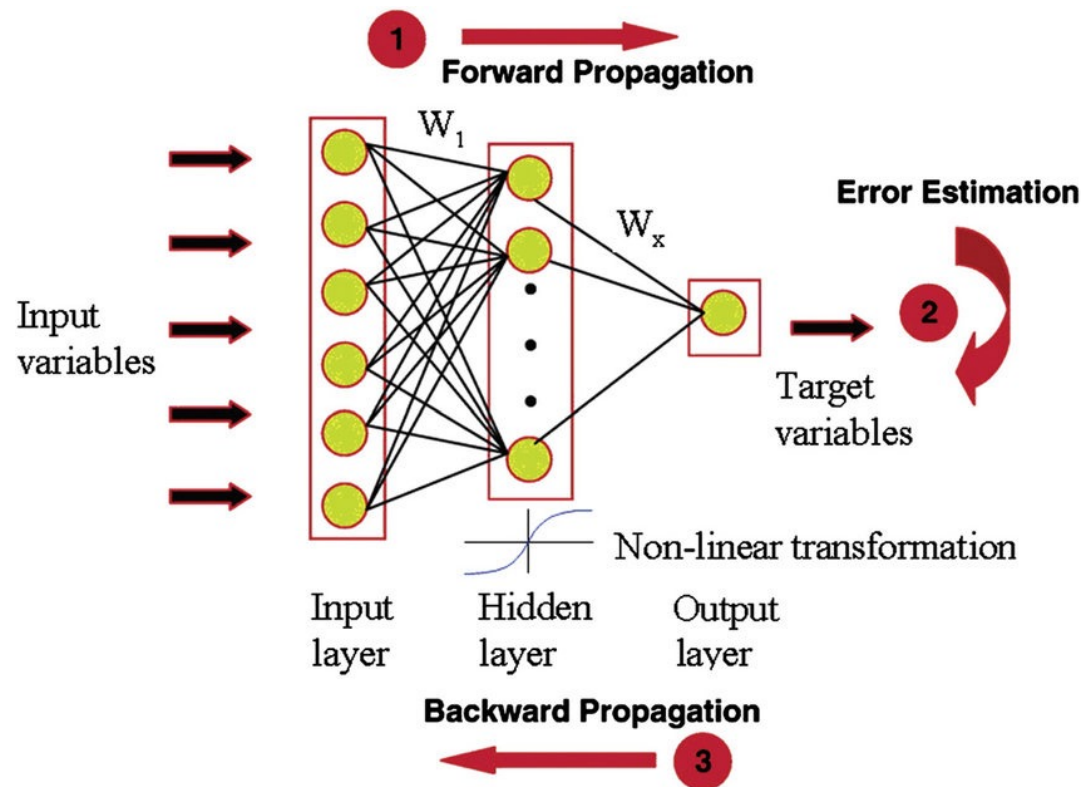
Advanced optimization techniques developed to provide a **faster learning** than the Gradient Descent Algorithm include the following ones

- Momentum Average Gradient
- Nesterov Accelerated Gradient (NAG)
- Adagrad
- Adadelata
- RMSProp
- Adaptive Moment Estimation (Adam)
- Adamax
- Adaptive Gradient Descent
- Momentum Gradient Decents
- Nadam
- AMSGrad

## The Backpropagation Algorithm

- Backpropagation is a method used to **calculate efficiently the gradient** of the **cost function** and **adjust the connection weights & the biases** to reduce the error during the learning process
- Backpropagation calculates **gradient** of the **cost function** with respect to each weight & bias by applying the **chain rule**, computing the gradient one layer at a time, **iterating** backwards from the last layer to the first one
- Weights & biases are updated by means of methods such as e.g. **gradient descent, stochastic gradient descent** or other methods

# The Backpropagation Algorithm: General Scheme

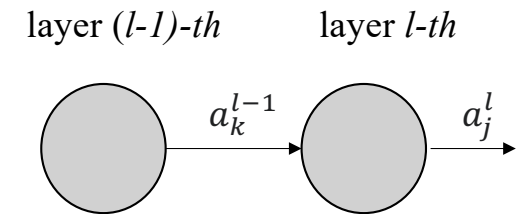


1. In a feed-forward ANN information **propagates sequentially** from through all the layers from the input to the output ones
2. The error (also known as **cost or loss function**) is evaluated
3. The error is **propagated backwards** to determine the new values of the fitting parameters at each layer & neuron
4. Steps 1 to 3 are repeated iteratively until a small error is reached

## The Backpropagation Algorithm: Main Ingredients

Before presenting the backpropagation algorithm let us first recall the notation and present the main ingredients:

- $\omega_{jk}^l$ : **weight** between the neuron  $k$ -th in the layer  $(l-1)$ -th and the neuron  $j$ -th in the layer  $l$ -th
- $b_j^l$ : **bias** of neuron  $j$ -th in the layer  $l$ -th
- $a_j^l = f(z_j^l)$ : **activation** (output) of neuron  $j$ -th in the layer  $l$ -th (note that  $a_j^l = \hat{y}_j$ )
- $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$ : **weighted input to the activation function** of neuron  $j$ -th in the layer  $l$ -th



A **little change  $\Delta z_j^l$**  in the weighted input of neuron  $j$ -th in the layer  $l$ -th **will propagate** through later layers in the network, finally causing the **overall cost to change** by an amount  $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ , where  $\frac{\partial C}{\partial z_j^l}$  can be interpreted as a measurement the of the **error of neuron  $j$ -th in the layer  $l$ -th**

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} f'(z_j^l) \quad (\text{BP1})$$

## The Backpropagation Algorithm: Main Ingredients

Since the weighted inputs in the layer  $(l+1)$ -th depends on the weighted inputs of the previous layer  $l$ -th, we can write

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Now, from  $z_k^{l+1} = \sum_j \omega_{kj}^{l+1} a_j^l + b_k^{l+1}$  we have

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \omega_{kj}^{l+1} f'(z_j^l) \longrightarrow \boxed{\delta_j^l = \sum_k \delta_k^{l+1} \omega_{kj}^{l+1} f'(z_j^l)} \quad (\text{BP2})$$

And therefore, the **gradient of the cost function** is simply given by

$$\boxed{\frac{\partial C}{\partial \omega_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial \omega_{jk}^l} = \delta_j^l a_k^{l-1}} \quad (\text{BP3})$$

$$\boxed{\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l} \quad (\text{BP4})$$

## The Backpropagation Algorithm: Summary

The backpropagation equations (BP1)-(BP4) provide us with a **fast way of computing the gradient of the cost function and adjusting the weights & biases**. Let's explicitly write it in the form of an algorithm

1. **Input  $x$ :** set the corresponding activation  $a_j^1 = x_j$  for each neuron  $j$ -th of the input layer
2. **Feedforward:** for each layer  $l = 2, 3, \dots, L$  compute  $z_j^l = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l$  and  $a_j^l = f(z_j^l)$
3. **Output error  $\delta_j^L$ :** compute the error of each neuron of the last layer  $L$ ,  $\delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L) = \frac{\partial C}{\partial \hat{y}_j} f'(z_j^L)$
4. **Backpropagate the error:** for each layer  $l = L - 1, L - 2, \dots, 2$  compute  $\delta_j^l = \sum_k \delta_k^{l+1} \omega_{kj}^{l+1} f'(z_j^l)$
5. **Gradient of the cost function:**  $\frac{\partial C}{\partial \omega_{jk}^l} = \delta_j^l a_k^{l-1}$  ,  $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
6. **Update the weights & biases:**  $\omega_{jk}^l \rightarrow \omega_{jk}^l - \eta \frac{\partial C}{\partial \omega_{jk}^l}$  ,  $b_j^l \rightarrow b_j^l - \frac{\partial C}{\partial b_j^l}$
7. **Repeat steps 2 to 6 till convergence is achieved**



## In which sense is Backpropagation a fast algorithm ?

To answer this question, suppose we want to compute the gradient of the cost function  $C$  by simply using the approximation

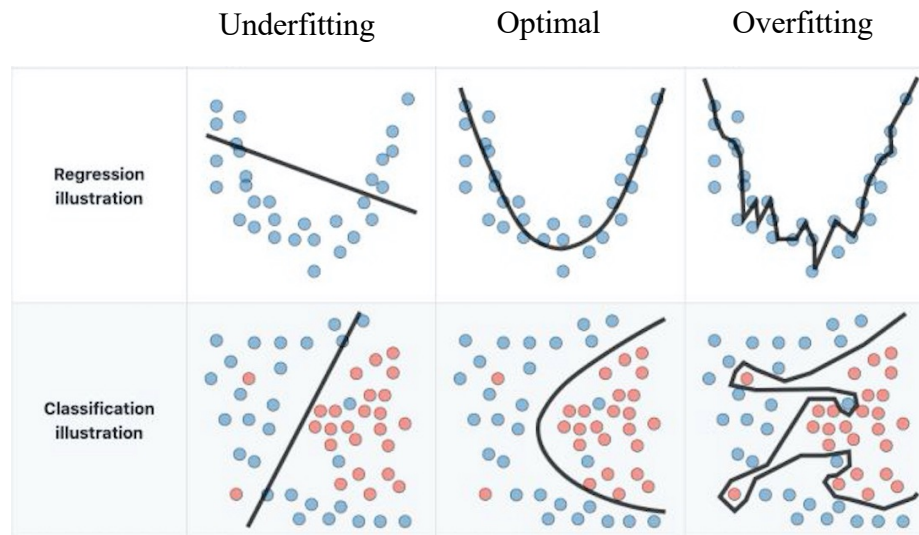
$$\frac{\partial C}{\partial w_{jk}^l} \approx \frac{C(\omega_{jk}^l - \epsilon e_{jk}^l, b_j^l) - C(\omega_{jk}^l, b_j^l)}{\epsilon}, \quad \frac{\partial C}{\partial b_j^l} \approx \frac{C(\omega_{jk}^l, b_j^l - \epsilon e_j^l) - C(\omega_{jk}^l, b_j^l)}{\epsilon}$$

where  $\epsilon > 0$  is a small positive number and  $e_{jk}^l$  ( $e_j^l$ ) is a unit vector in the direction of  $\omega_{jk}^l$  ( $b_j^l$ )

This looks very promising, we only have to compute  $C(\omega_{jk}^l, b_j^l)$ ,  $C(\omega_{jk}^l - \epsilon e_{jk}^l, b_j^l)$  and  $C(\omega_{jk}^l, b_j^l - \epsilon e_j^l)$  for each distinct weight  $\omega_{jk}^l$  and bias  $b_j^l$ . **However**, this is **extremely expensive computationally speaking**, specially for neural networks with a extreme large number (millions) of weights and biases

What is clever about the backpropagation algorithm is that it **enables us to compute simultaneously all the partial derivatives  $\frac{\partial C}{\partial w_{jk}^l}$  and  $\frac{\partial C}{\partial b_j^l}$  using just one forward pass through the network followed by one backward pass through the network**, *i.e.*, the computational cost of the forward and backward passes is the same. The **numerical cost of backpropagation** is **roughly the same as** making **just two forward passes**

# Overfitting of an ANN



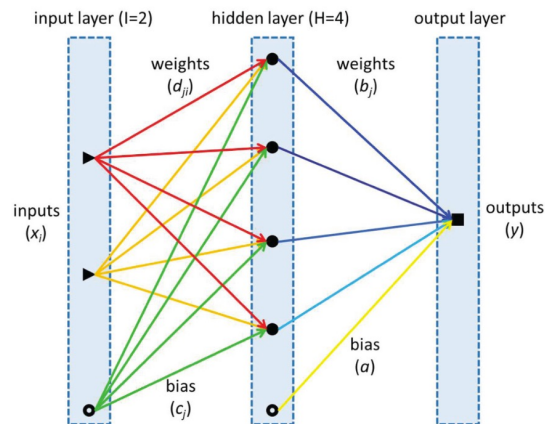
- A major issue in the development of an ANN is **overfitting** (also known as **overtraining**), which basically means that the network, due to its high flexibility to approximate complex non-linear functions, **tries to fit the data entirely and ends up memorizing all the data patterns**.
- Due to **overfitting** the **predictability** of the network on testing data becomes **questionable**
- Strategies to avoid **overfitting** include among others:
  - **early stopping** of the training: stops the training process once the model performance stops improving on the validation dataset
  - **dropout**: reduce overfitting by dropping randomly neurons from the neural network during training in each iteration
- In addition to these which can be used together, overfitting can be reduced by:
  - **enlarging** the input dataset (specially in those case where the input dataset is not large enough)
  - **adding noise** to the input dataset making the network less able to memorize data patterns since they change randomly during the training

# Bayesian Neural Networks (BNN)

- In the Bayesian approach, **the Neural Network parameters  $\vec{\omega}$**  (weights & biases) **are described probabilistically**

Suppose we have a set of data  $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$  where  $x_k$  and  $y_k$  ( $k = 1, 2, \dots, N$ ) are input and output data and  $n$  is the number of data. Then the probability distribution of  $\omega$  after the data  $D$  are taken into account, the **posterior distribution**  $p(\omega|D)$ , is given based on Bayes's theorem

$$p(\omega|D) = \frac{p(D|\omega)p(\omega)}{p(D)} \propto p(D|\omega)p(\omega)$$



- $p(D)$ : normalization constant which ensures the posterior distribution is a valid probability density and integrates to 1
- $p(w)$ : prior distribution based on our background knowledge
- $p(D|\omega)$ : likelihood function usually assumed to be a **Gaussian distribution**  $p(D|\omega) = \exp\left(-\chi^2/2\right)$ , with

$$\chi^2 = \sum_{i=1}^N \left[ \frac{S(\vec{x}; \vec{\omega}) - y_i}{\Delta y_i} \right]^2, \quad S(\vec{x}; \vec{\omega}) = a + \sum_{j=1}^H b_j f\left(c_j + \sum_{i=1}^I d_{ji} x_i\right)$$

With the posterior distribution the BNN prediction can be calculated by

$$\langle S \rangle = \int S(\vec{x}; \vec{\omega}) p(\omega|D) d\omega$$

# Other Machine Learning Techniques

Acronym	Method	Brief Description	Learning Type
AE/VAE	Auto Encoders / Variational Auto Encoders	ANN capable of learning efficient representations of the input data without any supervision	U
ANN	Artificial Neural Networks	Models for learning defined by connected units (or nodes) and hidden layers with well defined inputs and outputs.	S
BED	Bayesian Experimental Design	Bayesian inference for experimental design	S
BM	Boltzmann Machines	Generative ANN that can learn a probability distribution from sets of changing inputs	U
BMA/BMM	Bayesian Model Averaging/ Mixing	Bayesian inference applied to model selection, combined estimation, or performed over a mixture model	S
BO	Bayesian Optimization	Optimization of functions without an <i>a priori</i> knowledge of functional forms.	S and Semi-S
BNN	Bayesian Neural Networks	ANN where the parameters of the network are represented by probabilities learnt by Bayesian inference	S
CNN	Convolutional Neural Networks	ANN where convolution is used to reduce dimensionalities	S
EMB	Ensemble Methods & Boosting	Methods based on collections of decision trees as simple learners	S
GAN	Generative Adversarial Networks	System of two ANN where a generative network generates outputs while a discriminative network evaluates them	U
GP	Gaussian Processes	Collection of random variables which have a joint Gaussian distribution used in Bayesian inference	Semi-S
KNN	<i>k</i> -nearest neighbors	Non-parametric method where inputs consist of the <i>k</i> closest training examples in a dataset	S
KR	Kernel Regression	Extension of linear regression methods to include non-linear function kernels	S
LR	Logistic Regression	Convex optimization method based on maximum likelihood estimate for classification problems	S
LSTM	Long short-term memory	RNN capable of learning long-term dependencies	S
PCA	Principal Component Analysis	Dimensionality reduction technique based on retaining the largest eigenvalues of the covariance matrix	U
REG	Linear Regression	Linear algebra methods used for modeling continuous functions in terms of their explanatory variables	S
RL	Reinforcement Learning	Learning achieved by trial-and-error of desired and undesired events	Neither S nor U
RNN	Recurrent Neural Networks	ANN where connections between nodes allow for temporal dynamic behavior	S
SVM	Support Vector Machines	Convex optimization techniques with efficient ways to distinguish features in datasets	S

# Implementation of Machine Learning: TensorFlow & Pytorch



- **TensorFlow** is a *free and open-source software library* for machine learning and artificial intelligence
- Developed by the **Google Brain** team for internal Google use in research and production, the initial version was released under the in 2015 Google released the updated version of TensorFlow, named TensorFlow 2.0, in September 2019
- It can be used in a wide variety of programming languages, most notably **Python**, as well as **Javascript**, **C++**, and **Java**
- It can be used in **Google Colab** (<https://colab.research.google.com>)



- **PyTorch** is an *open source machine learning framework* based on the used for applications such as *computer vision* and *natural language processing*
- Primarily developed by **Meta AI (Facebook)**, initialliy released in September 2016
- Although the **Python** interface is more polished and the primary focus of development, PyTorch also has a **C++** interface

- ✧ You for your time & attention
- ✧ The organizers for their invitation

