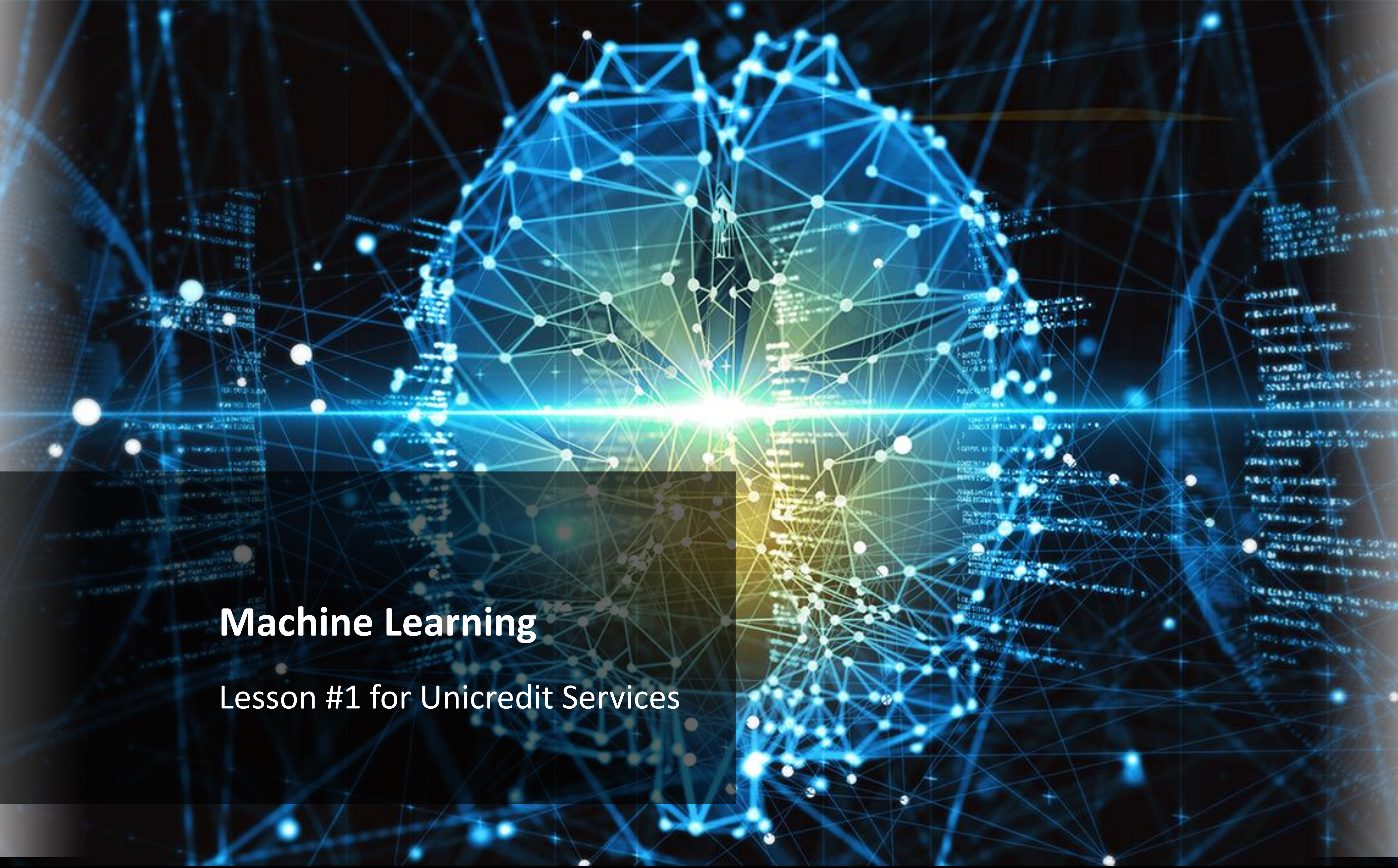


Machine Learning

Lesson #1 for Unicredit Services

iason ESSENTIAL SERVICES FOR FINANCIAL INSTITUTIONS

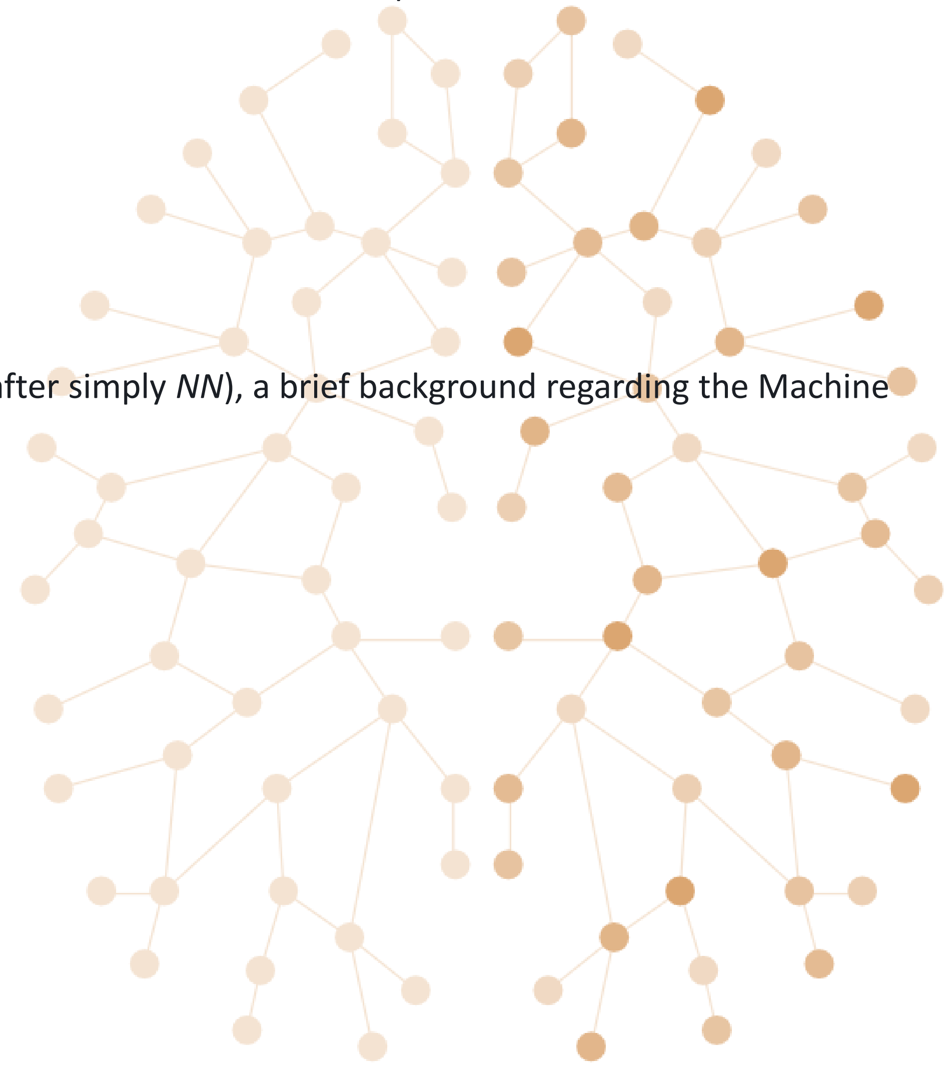
Executive summary

Overview

In this first lesson, we are going to learn the basics of a ***Feed Forward Neural Network***. In particular, at the end of the lesson, one should understand (among others) concepts like:

- **Neurons** and **Layers**
- **Forward pass**
- **Loss** and **activation** functions
- Learning through **backpropagation**
- **Under-** vs **over-fitting**

Before diving directly into the theory of Neural Networks (hereafter simply *NN*), a brief background regarding the Machine Learning space will follow



Agenda

Machine Learning, a brief intro

What Machine Learning is?

Many definitions have been provided for the concept of Machine Learning, here just a few…

Arthur Samuel (1959)

*"**Machine Learning** is the field of study that gives the computer the ability to learn without being explicitly programmed."*

Tom Mitchell (1998)

*"A **computer program** is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience."*

Andriy Burkov (2019)

*"**Machine Learning** is a subfield of computer science that is concerned with building algorithms which, to be useful, rely on a collection of examples of some phenomenon. These examples can come from nature, be handcrafted by humans or generated by another algorithm."*

Machine Learning, a brief intro

How does Machine Learning classify?

Quite often, Machine Learning is used interchangeably with AI itself, but this is a misuse of the term

Artificial Intelligence (*AI*)

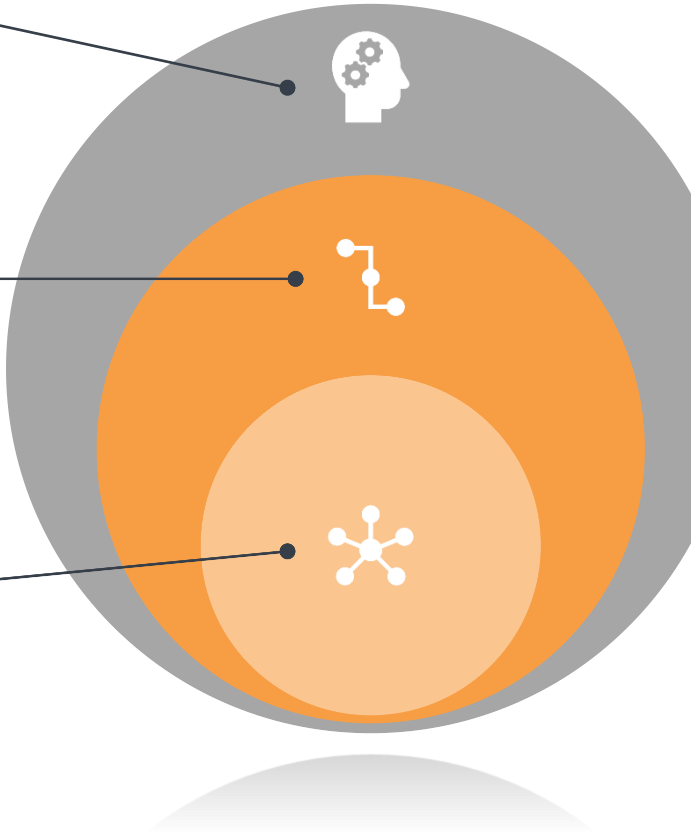
Any technique which enables computers to mimic human behaviour

Machine Learning (*ML*)

Subset of AI techniques which use statistical methods to enable machines to improve with experience

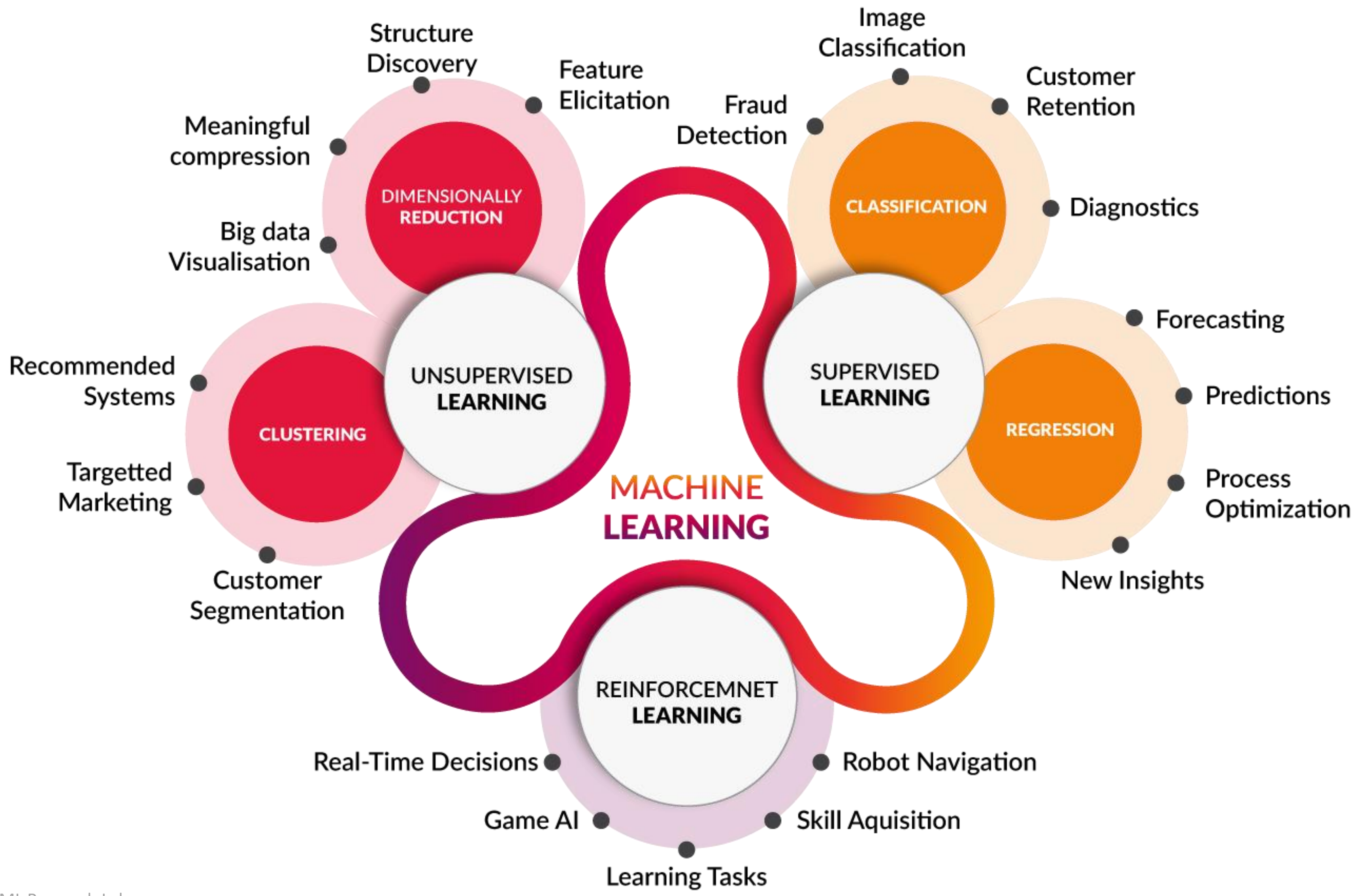
Deep Learning (*DL*)

Subset of ML algorithms which is based on multi-layer neural networks to allow the machine learn



Machine Learning, a brief intro

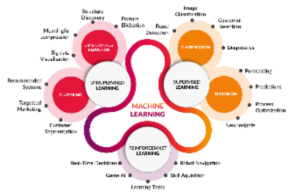
Types of Machine Learning algorithms



SOURCE:
Medium, ML Research Lab

Machine Learning, a brief intro

Supervised Learning



Supervised Learning (SL) is the typology of ML where the algorithm learns from a set of *labelled* examples, $\{(x_i, y_i)\}_{i=1}^{N}$, where under this notation:

- The x_i are the so-called **features**, usually a ***M***-dimensional array – i.e. $x_i = (x_1^{(i)}, \dots, x_M^{(i)})$
- The y_i are the so-called **labels**, which can be either a single-value variable (as for example for some regression problems) or an array itself (as for classification problems)

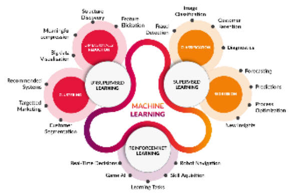
The **goal** of a SL algorithm is to create a model of the studied phenomenon directly from the data, learning from the **target examples** without being explicitly programmed or instructed to **perform that given task**.

Typical problems that can be solved with SL algorithms are:

- **Regression**, where starting from a labelled feature vector the algorithm has to predict a target value (usually real-valued) for an unseen example; i.e.:
    - *Future stock price*
    - *The air pressure hours ahead*
    - …
- **Classification**, where starting from a labelled feature vector the algorithm has to predict a specific class for an unseen example; i.e.:
    - *Face recognition*
    - *Credit rating*
    - …

Machine Learning, a brief intro

Unsupervised Learning



Unsupervised Learning (UL) is the typology of ML where the algorithm learns from a set of *unlabelled* examples, $\{x_i\}_{i=1}^{N}$. With this notation, as for the SL case:

- The x_i are the so-called features, usually a ***M***-dimensional array – i.e. $x_i = (x_1^{(i)}, \dots, x_M^{(i)})$

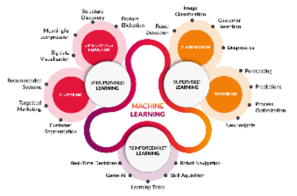
The goal of a UL algorithm is to create a model of the studied phenomenon, again with no pre-defined instructions and directly from the data, that captures the **underlying structure of the samples provided**.

Typical problems that can be solved with UL algorithms are:

- Clustering, where the goal of the algorithm is to separate the provided data into sets/ groups of features which shares common structures/ properties; i.e.:
  - *Customers segmentation*
  - *Word embeddings*
  - …
- Dimensionality reduction, where the algorithm maps a higher dimensional space into a smaller one, based on the underlying (non-linear) relations among the given data; i.e.:
  - *Automatic feature engineering*
  - *Data compression*
  - …

Machine Learning, a brief intro

Reinforcement Learning



Reinforcement Learning (RL), maybe the area of major developments in these years, is a pretty different kind of algorithm. In a RL setting, the machine actually “lives” in an environment where it learns the underlying rules (i.e. *polices*) with a **penalty vs reward** strategy.
The algorithm learns the rules of the world it lives in by iteratively collecting feedbacks from the surrounding environment, persevering along with the actions which allow the adopted strategy to gain the most reward as possible

The goal of a UL algorithm is to create a model of the studied phenomenon, again with no pre-defined instructions and directly from the data, that **maximizes some utility function**.

Typical applications of RL algorithms are:

- **AI gaming**, where a famous example is the DeepMind’s **AlphaGo**
- **Robot Navigation**, where a machine learns how to move in the surrounding space
- **Portfolio Optimization**, where the goal is to maximize the P&L
- …

Agenda

The basics of Neural Networks

A brief overview (1/3)

The concept of **Neural Network** (hereafter simply NN), is not really quite new… Actually the first idea of an **artificial neuron** has been proposed in **1943** by the work of W.S. McCulloch and Walter Pitts *"A logical calculus of the ideas immanent in nervous activity*".

After that work, other two papers published in **1958** played a big role in paving the future of ML:

1. "*The computer and the brain*", by J. Von Neumann
2. "*Psychological review*", by F. Rosenblatt who introduced the **first neural network architecture**, the **Perceptron**

That have been said, one can define a NN as follows:

A **Neural Network** is a computing system inspired by, but not identical to, biological neural networks that constitute animals brain.

In this regards, such networks are a collection of units or nodes called **neurons**, which loosely model their alter-ego in the biological brain. Each neuron is linked to the others by means of connections (just like the biological synapses) which allow to transmit signals.

Why then if these algorithms have been proposed in the research community this long ago, have they been massively exploited only in the very recent years?

The answer is quite obvious: in order to properly work, NN **need great computing power** and **data to be ingested**.

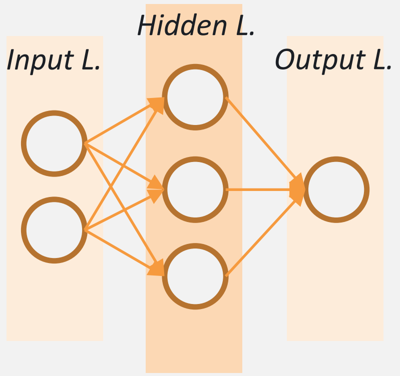
The basics of Neural Networks

A brief overview (2/3)

As anticipated in the previous definition, each NN architecture (i.e. *feed-forward, recurrent, …*) shares the same main components, namely:

1. The **Neuron**, the basic unit of a neural network that is in charge of taking **very simple and specialized decisions**
2. The **Layer**, a **collection** of such **neurons**

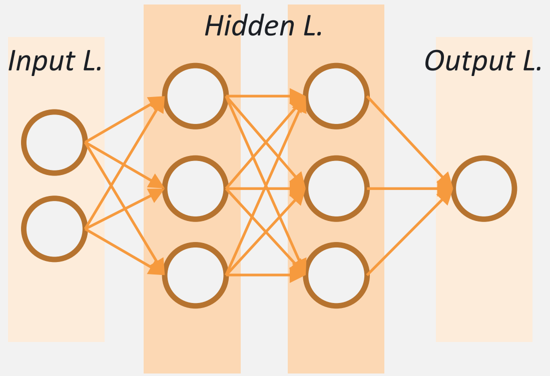
Before diving into the details of these components, it is worth defining what a **shallow** and a **deep NN** are.



***Shallow** NN*

Neural network which usually has **only 3 layers**, namely:

1. The **Input** layer, which collects all the **inputs**
2. The **Hidden** layer, which performs the **internal transformations**
3. The **Output** layer, which performs the **prediction(s)**



***Deep** NN*

Neural network where **more hidden layers** are stacked together between the input and the output layers

The basics of Neural Networks

A brief overview (3/3)

While **shallow neural networks** are able to **tackle complex problems**, **deep learning networks** are usually **more accurate**, being able to **capture more subtle structures** underlying the given data.

As a rule of thumb, increasing the number of hidden layers does **not always** provide better performances, since doing so:

- means **increasing** the **effort** required for **train** the algorithm and **interpret its results**
- can lead to **overfitting**

Adding on what has been explained so far, it is important to provide an important fact that helps furthermore appreciate why NN are having such a big impact in the automation nowadays

Universal Approximation Therorem

A feed-forward network with a single layer containing a finite number of neurons can approximate continuous functions on compact subsets on $\mathbb{R}^n$, under mild assumptions on the activation function

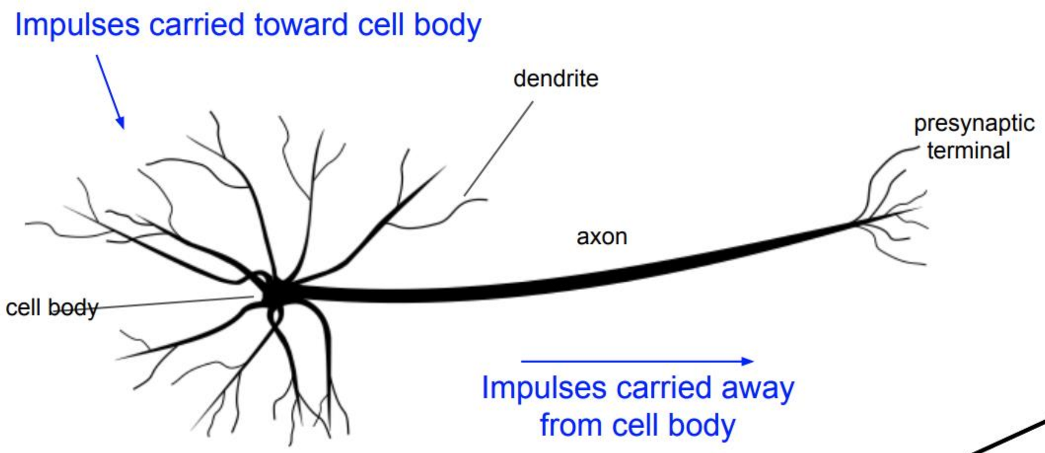
In general, **neural networks can emulate almost any function**, and answer to a huge variety of questions given enough training data and computing power

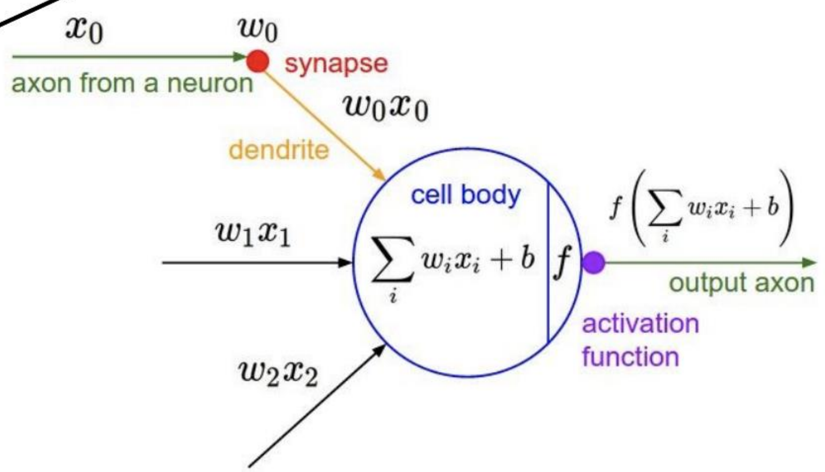
The basics of Neural Networks

Neuron – *The main idea*

Building on the biological analogy early provided introducing the neural networks, the **neuron** is the basic computing unit of a NN where each:

- receives input signals from a sets of connections, as it happens with the **dendrites** in the brain
- and produces an output singal, as it happens along the **axon** in the brain.


Impulses carried toward cell body
dendrite
presynaptic terminal
axon
cell body
Impulses carried away from cell body



The basics of Neural Networks

Neuron – *The formal definition*

In order to explain how a neuron works, let's first define some notation. In particular, we will denote:

- $x = (x_1, \dots, x_N) \in \mathbb{R}^N$ as the **input signal** feeding our neuron
- $W = (w_1, \dots, w_N) \in \mathbb{R}^N$ as the **weights** connecting each *i*-th input signal component to the neuron (just like the dendrites do in our brain)
- $\Phi(\cdot)$ as the **activation** function, which controls how much the neuron will react (i.e. fire) in response to the given input signal x

As explained, the input signal received by the neuron is processed and passed by. But how this "processing" step is performed? Basically the computation is split in **2 phases**:

1. A **linear transformation** that mixes all the info carried in the input signal
2. A (usually) **non-linear transformation** that "decides" whether the input signal is relevant for the neuron (*i.e. how much the neuron reacts in response to the input signal*) or not

Mathematically, we can summarize these steps with the computation of $\Phi(Wx^T)$, where:

1. $Wx^T = z$ is the **linear pass**
2. $\Phi(z)$ is the non linear transformation – the so-called **activation**

Together, this two operations make the so-called **forward pass**

The basics of Neural Networks

Neuron – *Examples*

The first example of neuron is the beforementioned **Perceptron**

For such a neuron, the **activation is a binary response**, either 0 or 1, with respect to a predefined threshold:

$$\Phi(z) = \begin{cases} 0, & if\ z \leq threshold \\ 1, & if\ z > threshold \end{cases}$$

In practice (and not only for the *Perceptron*), the threshold is embedded in the linear transformation (actually an *affine transformation*) and its value is learned by the algorithm itself, along with the weights W.

Given this, the equations presented so far translate into:

$$\Phi(z) = \Phi(Wx^T + b) = \begin{cases} 0, & if\ Wx^T + b \leq 0 \\ 1, & if\ Wx^T + b > 0 \end{cases}$$

where the threshold b is called **bias**.

One can think of this **bias** as a **measure of how easy is for the Percepetron to get activated**, hence to output a value equal to 1. For a Perceptron with a really big bias, it is extremely easy for it to fire (i.e. to output 1); viceversa, if the bias is very negative then the Perceptron gets switched off, being difficult for it to output 1

The basics of Neural Networks

Layer – *The main idea*

A **layer** is really nothing else than a **collection of neurons**.

If we were to use only a single neuron, we would indeed:

- Be able only to produce single-valued feedbacks; mathematically, we would create a mapping function from a *n*-dimensional space to a single real value: $\Phi(\cdot)\colon\ \mathbb{R}^n \longrightarrow \mathbb{R}$
- Not be able to capture different interactions among the input data; the basic idea is that each neuron captures a different aspect of the features

Given this, we can define a layer as a list of neurons where the **forward pass is defined as a set of forward pass operations**, one for each neuron in the layer

To help understand it better, let's formalize the concept; assume that:

- L is the number of neurons in the layer
- $x \in \mathbb{R}^N$ is, as before, the input signal
- $W = (\mathrm{W}_1, \dots, \mathrm{W}_\mathrm{L})^\mathrm{T}$ is the weight matrix, where $W_l = (w_{l,1}, \dots, w_{l,N})$ for every layer $l = 1, \dots, L$
- $\Phi(\cdot)$ is the activation function, but in this setting it operates **neuron-wise**

With these info, we define the **forward pass** as the sequence of the following operations:

1. $Wx^T + b = (W_1x^T + b, \dots, W_Lx^T + b) = z \in \mathbb{R}^L$
2. $\Phi(z) = \big(\Phi(z_1), \dots, \Phi(z_L)\big) = a \in \mathbb{R}^L$

The input signal hence follows a non-linear transformation from $\mathbb{R}^N$ to $\mathbb{R}^L$

Agenda

Under the hood

Overview

This section is aimed at covering the **main concepts and mechanism of the learning of a neural network**.

In particular, we are going to focus on the learning process in a *Supervised Learning framework*, along with some best practices to be kept in mind for the training phase.

To do so, we are going to explore the following concepts:

1. **Training, Validation** and **Test sets**
2. **Backpropagation algorithm**
3. **Activation functions**
4. **Loss functions**
5. **Under- vs Over-fitting**

Under the hood

Introduction to some technical concepts

In order to enter into the details of the training and evaluation of a ML model, it is important to introduce some other basic concepts …

Hyperparameters

All those **parameters** that the **model owner has to choose independently**, that hence **cannot* be learnt** by the algorithm itself; examples are: number of layers, learning rate, momentum, …

(Actually in this regards there's a lot of research, often referred to as Auto Machine Learning)*

Training Set

The **portion of the dataset used for the training phase** – for example, in a SL setting it is composed by tuples of features and labels

Test Set

The **portion of the dataset used for the testing phase**, i.e. when the prediction of the algorithm is evaluated against ***unseen examples***

Validation Set

In many applications it is worth saving an extra **portion of the dataset in order to fine-tune the hyperparameters**, before testing the model against the unseen data (i.e. the test set)

Batch

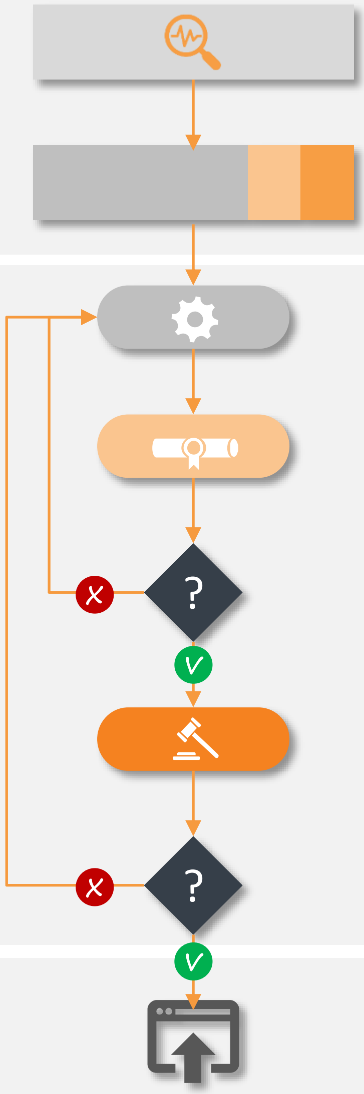
Subset of a dataset used to perform an **evaluation step**

Cost or Loss function

Metric against which the prediction of the algorithm is **evaluated**

Under the hood

General process – *From the data to the deploy*



Data investigation and preparation

Retrieve, **analyze** and **clean** the dataset

Split into Training, Validation and Test **sets**

Model assessment

Train the algorithm on the training set

Validate the algorithm on the validation set, fine-tuning the hyperparamenters

Is the model **validated?**

Test the algorithm on the test set

Does the model **perform sufficiently good** on the test set?

Deploy

Deploy the model in production

Under the hood

How a NN actually learns? – *The case of a SL algorithm* *(1/2)*

A **NN learns thanks to an optimization process which looks at the gradient w.r.t. each weights** (and biases) of the chosen cost function, **updating these parameters accordingly** (*the greater the gradient, the greater the update*)

Building on what has just been depicted, this is schematically how a NN learns in a SL setting :

1. The **input is propagated forward** (i.e. forward pass) along the network, layer by layer

2. At the **final layer**, the output layer, the NN **makes a prediction**

3. Such **prediction** is **evaluated** against the target value(s) – i.e. the label(s) - given a distance metric (i.e. **cost or loss function**) which evaluates how good the prediction is by providing a score: the **error value** – the bigger the error, the worst the NN bet

4. Starting from that error, the training **algorithm evaluates the contribution of each weight and bias** of the network, starting from the last layer up to the input's

5. This **contribution is evaluated in terms of gradient**, which is **propagated backward** (thanks to the chain rule), again from the output layer up to the input's

6. The **parameters** of the NN are then **updated** on the basis of their respective gradient value: the bigger the gradient, the bigger the update

Focus on next slides

Under the hood

How a NN actually learns? – *The case of a SL algorithm* *(2/2)*

In more details, the learning problem can be formulated in terms of **minimization of a predefined loss function $\boldsymbol{C(y)}$**.

Given this, the **whole learning phase is built around the computation of the gradient** of $C(y)$, with respect to the NN learnable parameters – i.e. weights and biases.

To keep it simple, assume that the NN has only 2 layers:

- A *n*-dimensional input layer, $x \in \mathbb{R}^N$
- A *1*-dimensionale output layer, $\Phi(Wx^T + b) = \hat{y} \in \mathbb{R}$

Assume also that, for each input x, y is the correct label

Then in order for the algorithm to start learning the correct model settings to perform the desired prediction, the following steps are performed:

1. **Initialize randomly** the NN parameters (i.e. weights and biases)
2. **Perform the forward pass**, where $\hat{y}$ is computed
3. **Evaluation of $\boldsymbol{C(\hat{y}; y)}$**, that is a particular “distance” between the NN p output, $\hat{y}$, and the true value to be predicted, y
4. **Evaluation of the contribution** of each single NN parameter **to the cost**; for example, in case of a single weight
$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial \Phi} \cdot \frac{\partial \Phi}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$
5. **Update the NN parameters** accordingly to their gradient; for example, with *Gradient Descent*, $\boldsymbol{w_i = w_i - \lambda \cdot \partial C / \partial w_i}$, where $\boldsymbol{\lambda}$ is the learning rate which controls the magnitude of the updating step
6. **Start from #1**, until a specified condition (*i.e. loss small enough*), or stopping criterion (*i.e. reached maximum number of iterations, …*), is satisfied

Under the hood

Backpropagation – *A brief overview*

In the previous example the NN had only 2 layers, but what if it is deeper? We would need an algorithm capable to propagate the gradient through the entire network.

With that in mind, the **role** of such an algorithm will be iteratively ***look for the set of weights and biases which lead to the minimum cost value as possible***

The **backpropagation** algorithm, usually also referred to as backprop, is the **procedure that allows the (feedforward) NN to calculate the gradients**.

(*Besides this, backopropagation is often used loosely to refer to the entire learning algorithm, also including how the gradient is used, such as by stochastic gradient descent*)

Not a new concept …

The backpropagation algorithm was **originally introduced in the 1970s**, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.

Algorithm 1 Backpropagation Algorithm

```
1:  procedure TRAIN
2:      $X \leftarrow$ Training Data Set of size mxn
3:      $y \leftarrow$ Labels for records in X
4:      $w \leftarrow$ The weights for respective layers
5:      $l \leftarrow$ The number of layers in the neural network, 1...L
6:      $D_{ij}^{(l)} \leftarrow$ The error for all l,i,j
7:      $t_{ij}^{(l)} \leftarrow 0$. For all l,i,j
8:      $For \quad i = 1$ to $m$
9:          $a^l \leftarrow feedforward(x^{(i)}, w)$
10:         $d^l \leftarrow a(L) - y(i)$
11:         $t_{ij}^{(l)} \leftarrow t_{ij}^{(l)} + a_j^{(l)} \cdot t_i^{l+1}$
12:     if $j \neq 0$ then
13:         $D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)} + \lambda w_{ij}^{(l)}$
14:     else
15:         $D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)}$
16:         where $\frac{\partial}{\partial w_{ij}^{(l)}} J(w) = D_{ij}^{(l)}$
```

Under the hood

Optimizers – *An intro to the different algorithms* *(1/2)*

Provided the general description of how a NN learns, what remains untouched is **how actually the algorithm behind optimizes its learnable parameters**. We have seen that the gradient evaluation along the whole network is the corner stone of the process, but how weights and biases are updated?

The **optimizer** is the procedure which updates the weights and biases to minimize the given loss function. In this regards, loss function acts as guides to the terrain telling optimizer if it is moving in the right direction to reach the bottom of the valley, the global minimum.

	Description	Mathematically ...
Gradient Descent	*Basic algorithm responsible for the NN optimization. It simply moves the model parameters in the opposite direction of the gradient, proportionally of a fixed parameter, the learning rate* Different king of GD algos exist: • **Batch GD** or **Vanilla GD**, which evaluates the gradient ΔC as average of the gradient of all the samples in the dataset • **Stochastic GD (SGD)**, which evaluates the gradient ΔC as average of the gradient of a batch of data, sampled randomly from the entire dataset	$\vartheta_{t+1} = \vartheta_t - \lambda \Delta C(\vartheta; x, y)$
Momentum	*Algorithm built on the analogy of a ball rolling downhill. As the ball will gain momentum as it rolls down the hill, in updating the weights momentum takes the gradient of the current step as well as the gradient of the previous time steps. This helps us move faster towards convergence*	$v_t = \gamma v_{t-1} + \eta \Delta C(\vartheta; x, y)$ $\vartheta_{t+1} = \vartheta_t - v_t$

Under the hood

Optimizers – *An intro to the different algorithms* *(2/2)*

	Description	Mathematically …
Nesterov Accelerated Gradient (*NAG*)	*Nesterov acceleration optimization is like a ball rolling down the hill but who knows exactly when to slow down before the gradient of the hill increases again*	$\vartheta_{t+1} = \vartheta_t - v_t$ $v_t = \gamma v_{t-1} + \eta \Delta C(\vartheta - \gamma v_{t-1})$
Adaptive Gradient Algorithm (*Adagrad*)	*Adagrad is an adaptive learning rate method where larger updates are for made for infrequent parameters and smaller updates for frequent parameters*	$\vartheta_{t+1} = \vartheta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \cdot g_t$ Where G_t is the sum of the squares of the past gradients w.r.t. all parameters ϑ
Adaptive Moment Estimation (*Adam*)	*Adam maybe is the **go-to solution** since, on average, it performs the best. It do so exploiting both a momentum term and an adaptive learning rate*	$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t$ $v_{t+1} = \beta_2 m_t + (1 - \beta_2) g_t^2$ $\vartheta_{t+1} = \vartheta_t - \frac{\eta \widehat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$ Where $\widehat{m}_t$ and $\hat{v}_t$ are the bias corrected estimates of the fiorst and second moment respectively

Under the hood

Optimizers – *Comparison*

	Pros	Cons
Gradient Descent	• Simple technique, easy to understand and implement	• Converges slower than newer algorithms • Has more problems with being stuck in a local minimum than newer approaches
Momentum	• Faster convergence than GD	• Risks to overshoot the minum, if the momentum gets too big
NAG	• Thanks to the «lookahaed» term, it can predict the best direction to move towards, before actually computing the step • Works slighly better than momentum	• It is more complex to implement and slower to compute
Adagrad	• Eliminates the need of manually tuning the learning rate	• It suffers the «diminishing learning rate» problem: since the gradient term at the denominator increasease at each iteration, the learning rate converges at 0 preventing the NN to learn further
Adam	• Eliminates the need of manually tuning the learning rate • Reduces the radically diminishing learning rates of Adagrad	• More complex to implement

iason

Under the hood

Activation functions – *An overview* *(1/2)*

An activation function is a **transformation of a neuron output** that **determines** the **feedback** of an output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated (“fired”) or not, based on whether each neuron’s input is relevant for the model’s prediction.

If we think of the **Perceptron** case seen earlier, our activation function was a **step function** which outputs either 0 or 1 given a certain threshold, the bias. This allows the algorithm do discriminate between ”no” and “yes” in a very simple case (*i.e. logical operations can be mimicked with Perceptrons*)

Is this the best we can do? Of course no… Let’s try to give an idea with a simple example.

Suppose we have some **images** of **humans** and others **not** containing images of humans.

Now while the computer processes these images, ***we would like our neurons to adjust its weights and bias so that we have fewer and fewer images wrongly recognized***. This requires that a small change in weights (and/or bias) causes only a small change in outputs.

Unfortunately, a Perceptron-like neural network does not show this little-by-little behavior. A Perceptron is either 0 or 1 and that is a big jump and it will not help it to learn.

We need something different, smoother. We need a function that progressively changes from 0 to 1 with no discontinuity.

Under the hood

Activation functions – *An overview* *(2/2)*

Why are activation functions that important and useful? Well:

1. Activation functions work as a **gate** for each neuron, **deciding whether and how much the neurons should respond** to an external signal (i.e. their input)

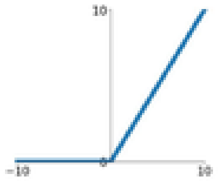
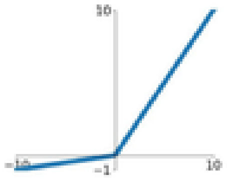
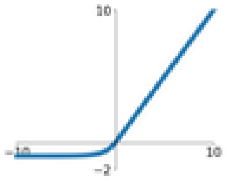
2. Since their non-linearity, is due to the activation functions that a neural network can perform non-linear operations and hence **understand non-linear relations among the data provided**; without activation functions, a NN is really just a simple linear regression …

3. Activation functions can **help to normalize the output** of each neuron to a range, for example between 1 and 0 or between -1 and 1

*Since their role, **choosing the right activation function is important** in order to reach the desired goal.*

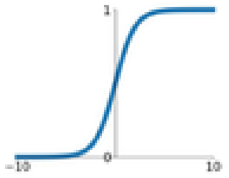
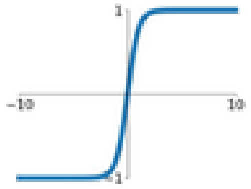
Under the hood

Activation functions – *The most used functions* *(1/2)*

Activations		Pros	Cons
ReLU $\max(0, x)$		• It **avoids** and rectifies **vanishing gradient** problem. • **Not** computationally **expensive** • **Converge fast**	• Could result in **Dead Neurons**: if it gets negative, it hardly recovers from 0 • The range of ReLu is [0, inf). This means it **can blow up** the activation
Leaky ReLU $\max(0.1x, x)$		• Inherits the **pros of ReLU** • **Tries to fix the "dying ReLU"** problem, outputting negative values	• The range of ReLu is [0, inf). This means it **can blow up** the activation
ELU $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$		• Inherits the **pros of ReLU** • **Tries to fix the "dying ReLU"** problem	• The range of ReLu is [0, inf). This means it **can blow up** the activation • Converges to a fix value in the **negative area**, hence the **gradient converges to 0**

Under the hood

Activation functions – *The most used functions* *(2/2)*

Activations		Pros	Cons
Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$		• **Analog**, but continuous version of the **step function** • Has a **smooth gradient** • Has a **limited output**, i.e. range (0,1)	• **Computationally expensive** • Suffers of **vanishing gradient**
tanh $\tanh(x)$		• Has a **smooth gradient** • The **gradient is steeper** than Sigmoid's	• Suffers from **vanishing gradient**
Softmax $\sigma(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$		• **Ideal** for a **multiclass classification** problem we are actually trying to attain the probabilities to define the class of each input.	• Suffers from **vanishing gradient**

Under the hood

Activation functions – *Best practices*

Generally speaking, there are some best practices worth to be kept in mind:

1. As a rule of thumb, you can **begin with using ReLU** function and then move over to other activation functions in case ReLU doesn't provide with optimum results

2. Always keep in mind that **ReLU** function should **only** be used in the **hidden layers**

3. If ReLU is giving problems (i.e. **dead neurons**), try **other versions of ReLU** as Leaky ReLU

4. The **sigmoid** and **hyperbolic tangent** activation functions have to be used with carefully in networks with many layers due to the **vanishing gradient problem**.

5. **Sigmoid** functions and their combinations (i.e. softmax) generally work better in the case of **classifiers**

6. **Normalize the data** in order to achieve higher validation accuracy, and standardize if you need the results faster

Under the hood

Cost functions – *An overview*

A **cost or loss function** is a **measure of "how good"** a neural network did with respect to its given training sample and the expected output. A cost function is a single value, not a vector, because it rates how good the neural network did as a whole.

To keep the discussion simple, yet providing a good understanding, following some commonly used cost functions

Cost Function	Mathematically ...	Problem to be solved	Output	Final activation
Mean Squared Error (MSE)	$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$	Regression	Numerical value	Linear
Mean Absolute Error (MAE)	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i - \hat{y}_i\rvert$	Regression	Numerical value	Linear
Huber	$\begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ \lvert y-\hat{y}\rvert \le \delta \\ \delta\lvert y-\hat{y}\rvert - \frac{1}{2}\delta^2 & otherwise \end{cases}$	Regression	Numerical value	Linear
Binary Cross Entropy	$-(y \cdot \log(\hat{y}) + (1-y)\cdot\log(1-\hat{y}))$	Binary classification	Binary outcome	Sigmoid
Cross Entropy	$-\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$	Multiclass classification	Single label, multiclass	Softmax

iason

Under the hood

Bias-Variance tradeoff

The **Bias** is a measure of how far off the model estimated values are from the true values. Given this, it refers to the error deriving from **erroneous assumptions in the learning algorithm**

The **Variance** refers instead to the change in parameter estimates across different data sets. It refers to the error deriving from the **sensitivity to small fluctuations in the training data**

Provided the definition of these two important concepts, it is worth now exploring their relation …

Bias-Variance tradeoff

Building on what has been depicted, the so-called *bias-variance tradeoff* refers to the problem that arise in the duality of Fitting vs Generalizing:

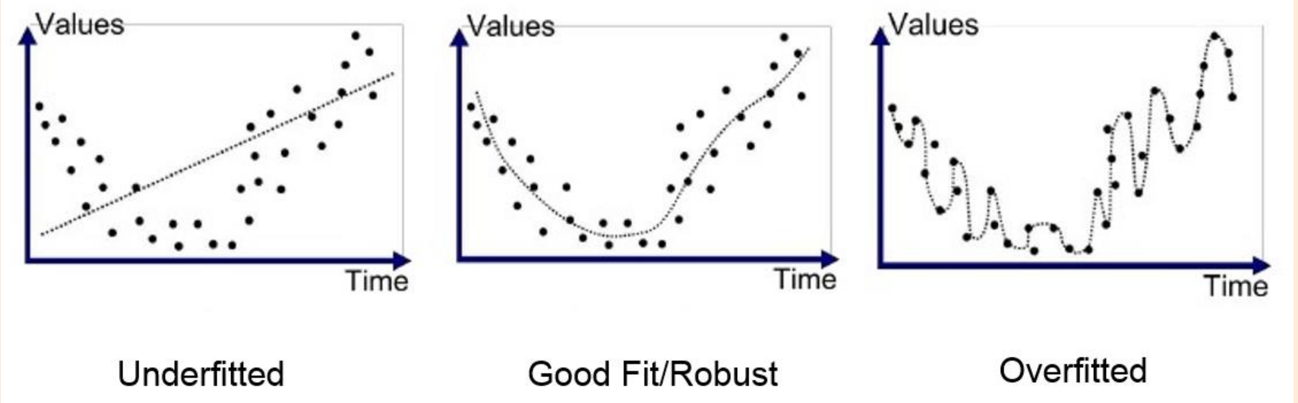
1. We want a model that **correctly fits** the given (training) data
2. But that, at the same time, **generalizes well** on unseen data.

Unfortunately, it is **not possible to solve the two problems at best simultaneously**: the model owner should decide a **tradeoff between accuracy during testing and final accuracy**, on production data

Under the hood

Under- vs Over-fitting – *Definition*

The concepts of Bias and Variance translates in practice in other two, maybe more familiar in general: Under- and over-fitting



Underfitting

Overfitting

A model is set to **underfit** when it does not capture the relations among the data, resulting in a **poor prediction**.

Reasons of this mainly derive **from assumptions too simplistic** about the true underlying model.

A model is set to **overfit** when it does captures also the randomness in the data, resulting in the **inability to generalize**.

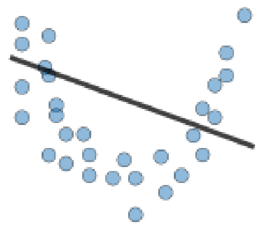
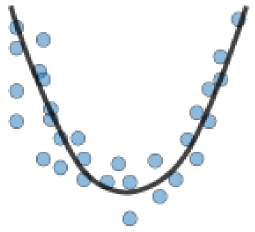
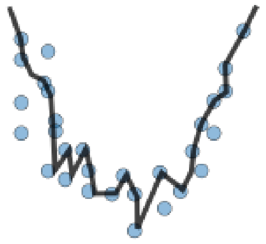
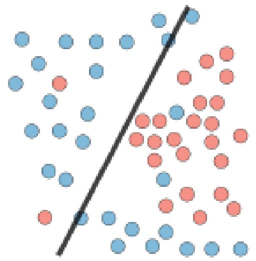
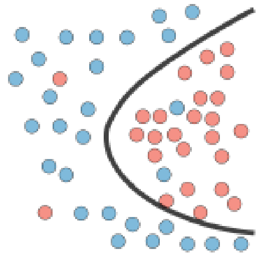
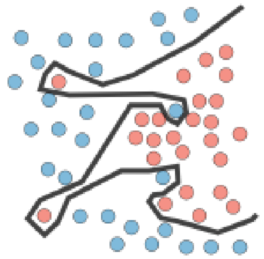
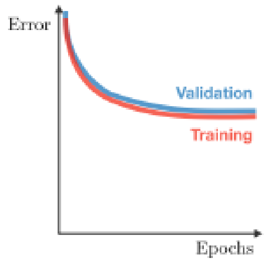
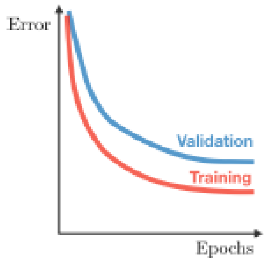
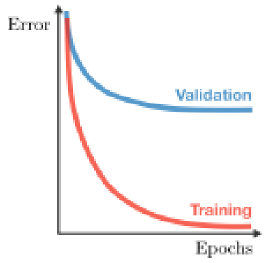
Reasons of this mainly derive from adopting a **predictor too complex** for the datataset used.

⬆ Bias ⬇ Variance

⬇ Bias ⬆ Variance

Under the hood

Under- vs Over-fitting – *Examples*

	Underfitting	Just right	Overfitting
Symptoms	- High training error - Training error close to test error - High bias	- Training error slightly lower than test error	- Low training error - Training error much lower than test error - High variance
Regression			
Classification			
Deep learning			
Remedies	- Complexify model - Add more features - Train longer		- Regularize - Get more data

SOURCE:
CS 229 – Machine Learning

Under the hood

Under- vs Over-fitting – *Best practices to prevent them*

Technique	How it works	Helps preventing
Early Stopping	*Stop training when some specific and predifined condition is no longer satisfied during the training phase* An example is stopping the training when the validation loss stops improving significantly (i.e. Less than a chosen threshold) for a ceratain number of epochs	Overfitting
Regularization	*Introduce a penalty in the loss function, in order to force the algorithm to prefer some model configuration (usually more simple)* Commong examples are Lasso (L1) and Ridge (L2) regularizations	Overfitting
Dropout	*Technique meant at preventing overfitting the training data by dropping out units in a neural network. In practice, neurons are either dropped with probability p or kept with probability 1 – p* This is similar to prune nodes in a tree or a forest	Overfitting
Increase complexity	*Add more feature or layers* Some features, for example, can be engineered in order to introduce more information in the model	Underfitting

Contacts

Antonio Menegon
Manager and FinTech Stream Leader

Mobile: +39 366 9534672
E-mail: antonio.menegon@iasonltd.com

www.iasonltd.com