Data driven approaches

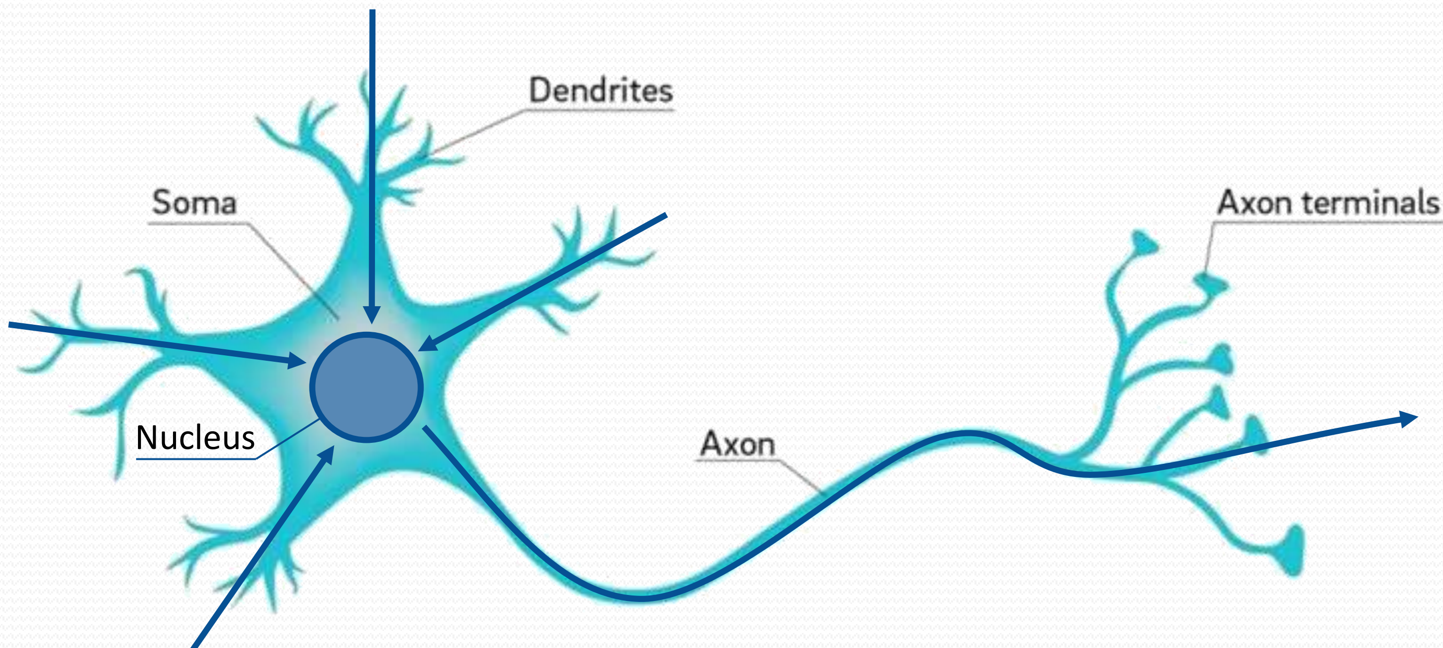
How to process data and extract Knowledge from it ?

Example on Artificial Neural Network – ANN – Definition, structure and tools

Neuron: From biological to artificial

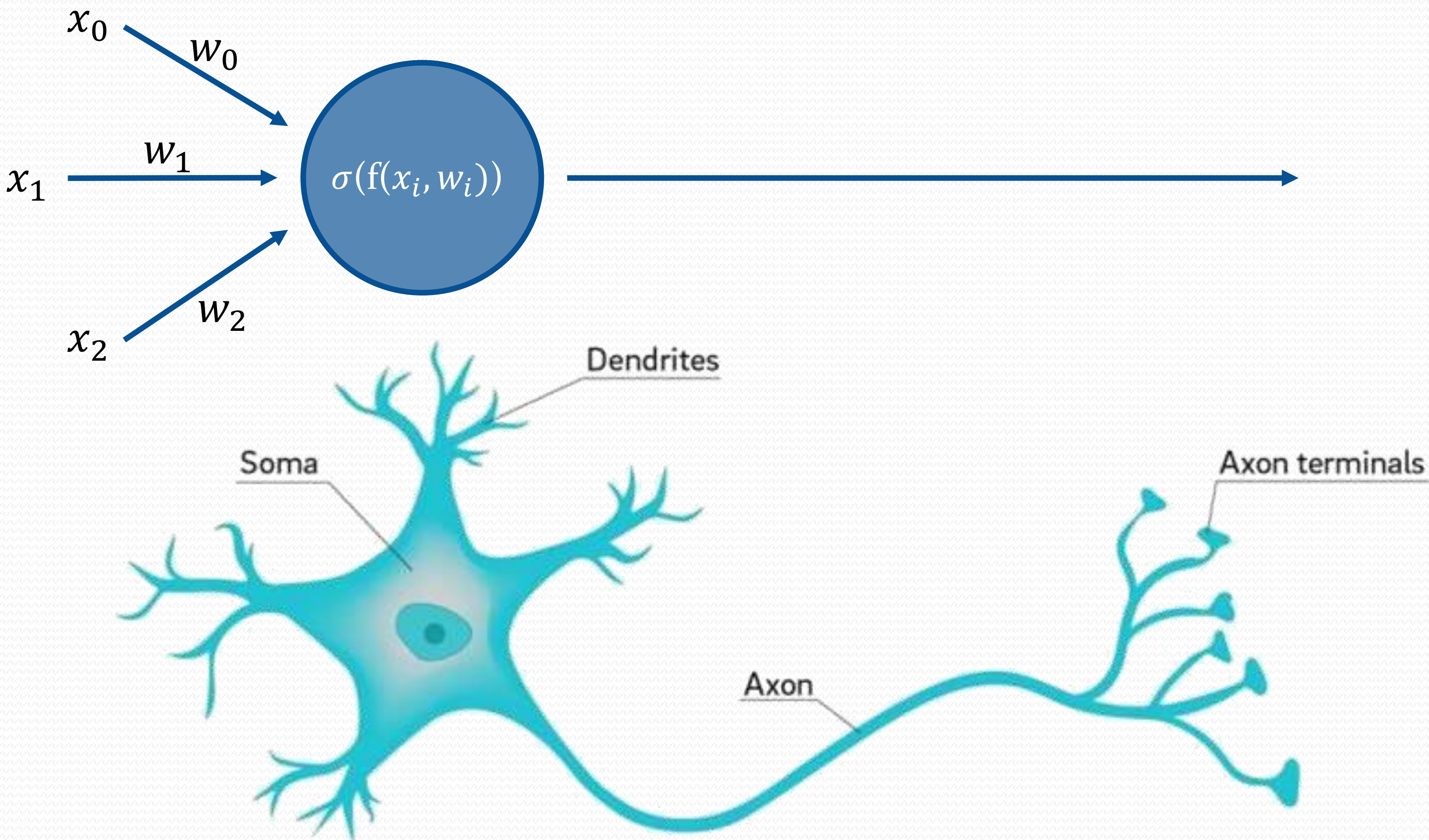
We have about 100 billions neurons. Each is connected to about 100000 others. How does it work ?

- From other neurons, through synapses, electrochemical signals enter into the neuron
- The amount of power of these signals can reach a threshold, which activates the neuron
- Generating a new signal to its neighbours



Neuron: From biological to artificial

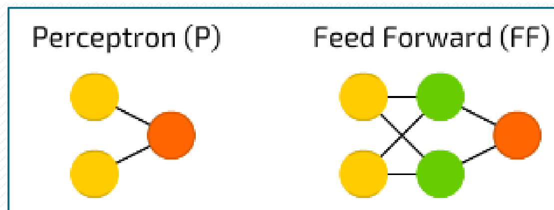
The artificial neurone has the same structure and behaviour:



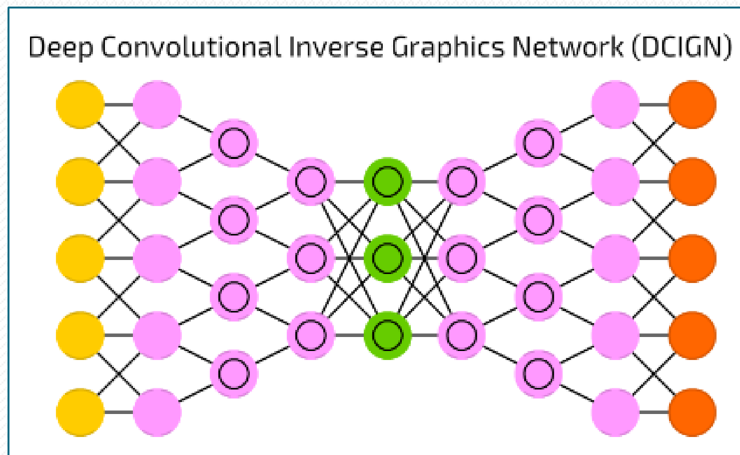
Zoo of neural networks

A lot of different structure of Neural Networks are available in the literature:

- From the simple and/or classical ones:



- To more complex ones:



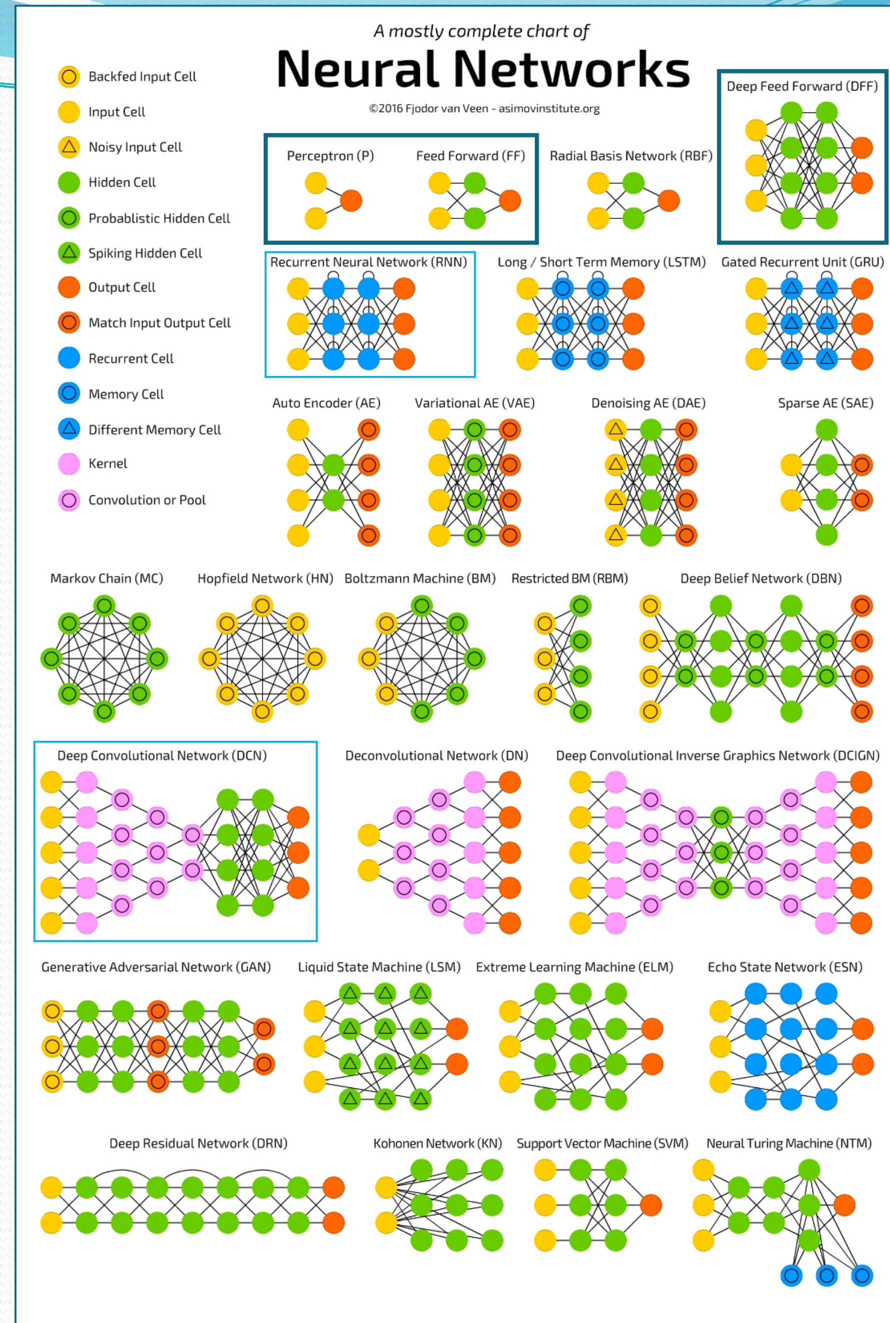
Zoo of neural networks

This leads to an increasing numbers of approaches, type of neurons, type of connections…

Several are dedicated to a particular goal, for instance:

- Recurrent Neural Network (RNN) are often used to manage sequence prediction problems (text, language…)
- Convolutional Neural Network (CNN) are mainly used to process images

This course only focused on the classical and simple one : **Multi Layer Perceptron (MLP)**

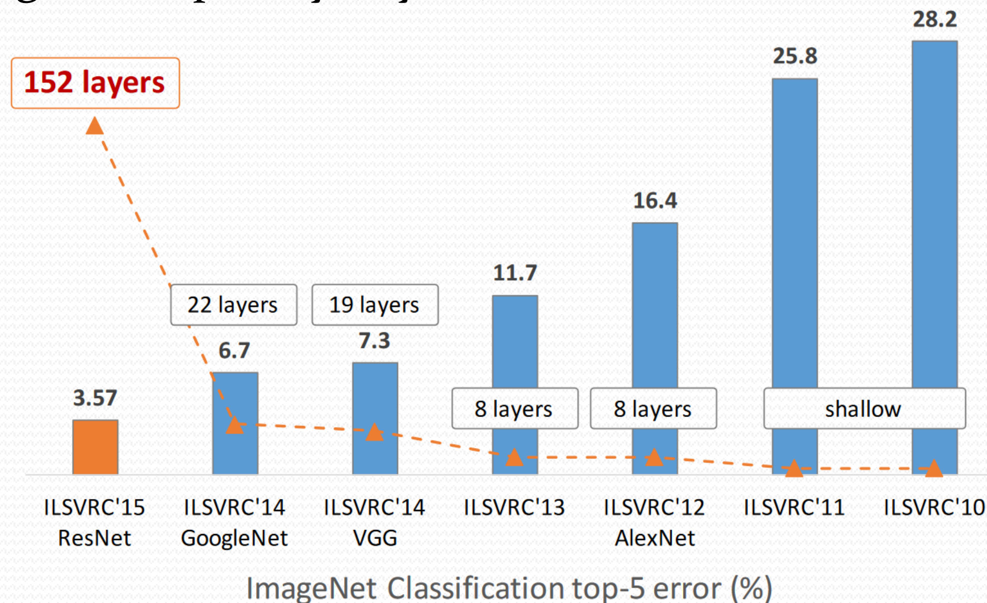


The Asimov Institute – The Neural Network Zoo

What's new in ANN since 1940?

Two main improvements ease the performances of ANN:

- The computer performances (GPU increasing performances) enable the increasing of complexity (layers) of artificial neural networks



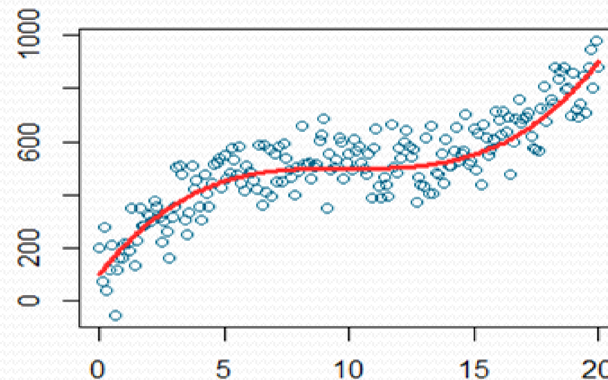
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

- The amount of data available is strongly increasing (more captors, Internet, IoT, connected devises…) => more cases to learn by the ANN
- New algorithms and approaches

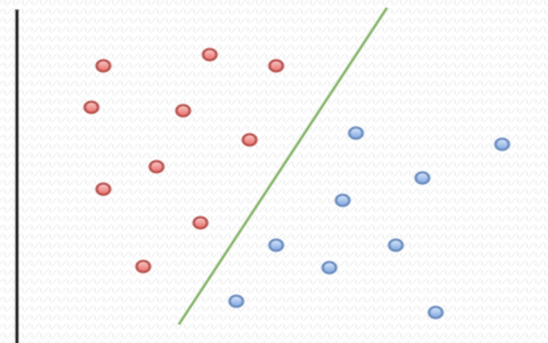
What to do with an MLP ?

MLP are commonly selected to solve classical data problems:

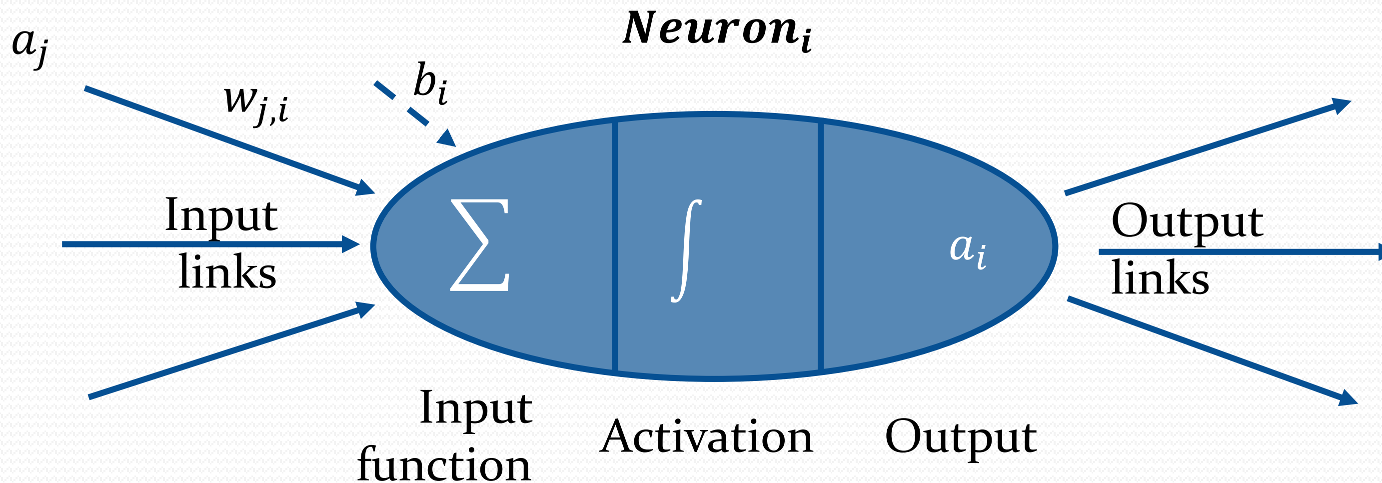
- **Regression** problems when the goal is to find an approximation of a function (not known or too costly to evaluate) represented by a dataset



- **Classification** problems when the goal is to identify to which group a new observation belongs to (2 classes or more…). The separator can be linear or not (=> Effect on the complexity of the MLP)



Modelling of an artificial neuron



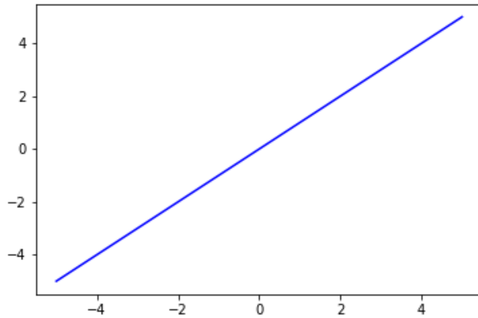
$$a_i = g(b_i + \sum a_k . w_{k,i})$$

Where:

a_j	activation value of neuron j
$w_{j,i}$	weight on the link from neuron j to neuron i
z_i	weighted sum of inputs to neuron i + the bias
g_i	activation function of neuron i
b_i	bias of neuron i

Activation functions

Several activation functions are available regarding what is the purpose of the ANN and the type of data handled.

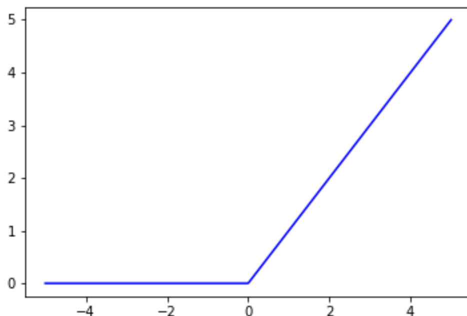


$$\boldsymbol{u} = \boldsymbol{x}$$

Linear function

Dedicated to regression

No transformation



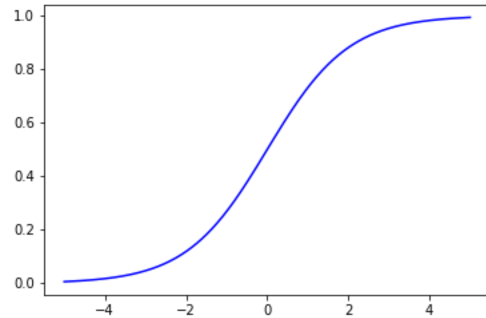
$$\boldsymbol{u} = \mathbf{max}(\mathbf{0}; \boldsymbol{x})$$

ReLU (Rectified Linear Units)

Filter negative values

Use in Deep Learning (easy to calculate (evaluation and derivation)

Activation functions

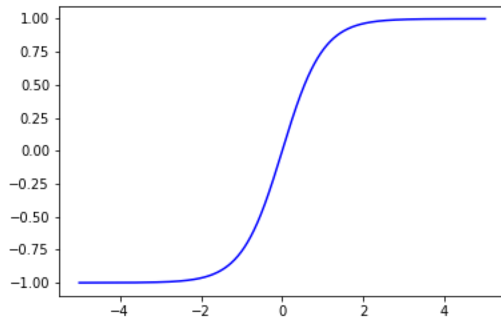


$$u = \frac{1}{1 + e^{-x}}$$

Sigmoïd

Move the activation in [0, 1] range

Use for Classification



$$u = \frac{e^{2x} - 1}{e^{2x} + 1}$$

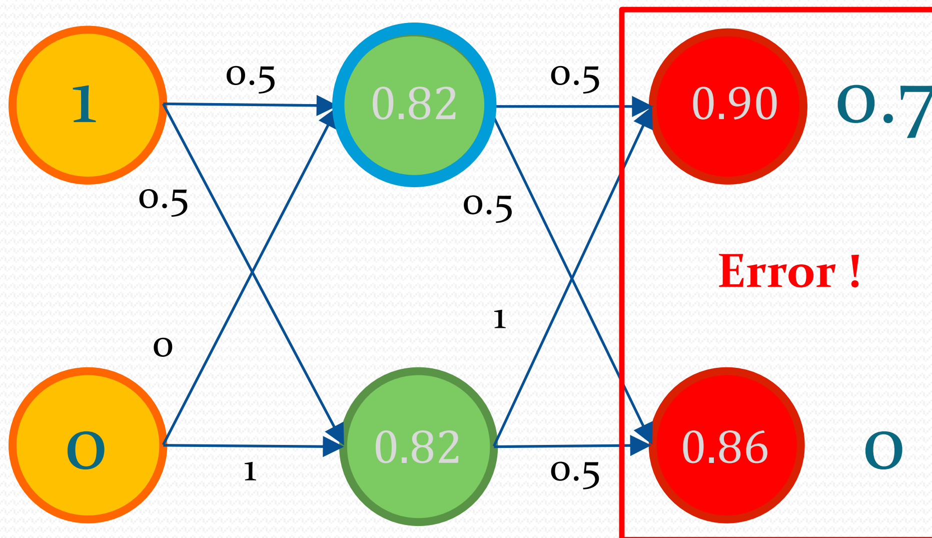
Hyperbolic Tangent

Move the activation in [-1, 1] range

Use for Classification

Forward Pass

The process followed by an ANN to evaluate its outputs regarding a case is called the **forward pass**. Consider this simple network (sigmoid and bias = 1). From the inputs whose outputs are known, for each neuron :



- Calculate the weighted sum of their inputs
- Add the bias
- Apply the activation function and generate the output value
- Complete the layer and go to next one

$$\frac{1}{1 + e^{-(1 * 0.5 + 0 * 0) + 1}} = \mathbf{0.82}$$

Loss function - Definition

The loss function ($\mathbf{L}(\boldsymbol{\theta}) = \boldsymbol{f}(\widehat{\boldsymbol{Y}_i}, \boldsymbol{Y}_i)$) models the error due to the configuration $\theta = (W, b)$ of the ANN (weights, biases).

The **goal is** then **to** find the best configuration **minimizing the loss function**: $\theta_{best} = Argmin_{\theta}\ (f(\widehat{Y_{learning}}(\theta), Y_{learning}))$

Examples of loss functions:

- Mean Square Method (MSM): $MSM = \frac{1}{n}.\sum_{i,j}(\widehat{y_{ij}(\theta)} - y_{ij})^2$
- Mean Absolute Error (MAE): $MAE = \frac{1}{n}.\sum_{i,j}\left|\widehat{y_{ij}(\theta)} - y_{ij}\right|$
- Cross Entropy Loss (for binary classifications):

$$CEL = \frac{1}{n}.\sum_{i,j} -y_{ij}\log\left(\widehat{y_{ij}(\theta)}\right) + \left(1 - y_{ij}\right)\log(1 - \widehat{y_{ij}(\theta)})$$

Loss function - Optimization of the ANN

To find the best configuration of the network, the goal is to find the minimum of the loss function in the domain space of all parameters ($\theta = (W, b)$).

Since no mathematical expression are available, numerical approaches are mandatory.

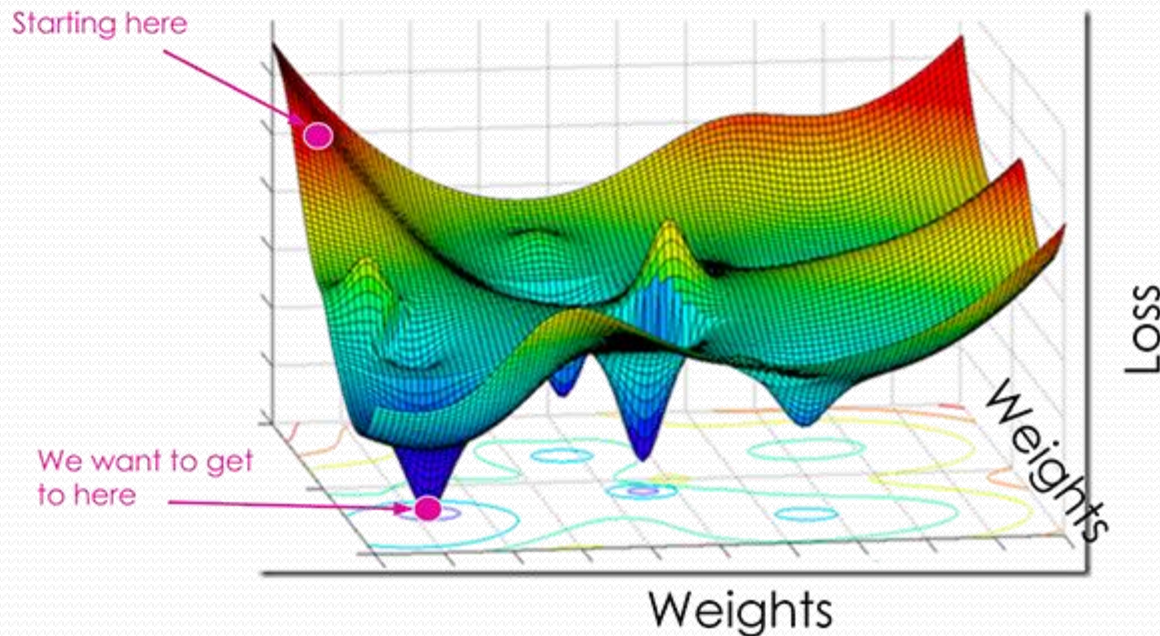


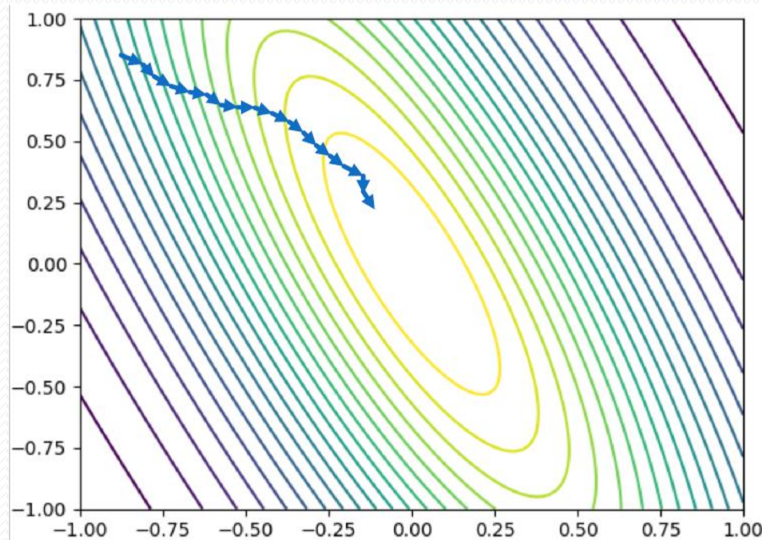
Figure from https://www.pyimagesearch.com

Learning - From prediction errors

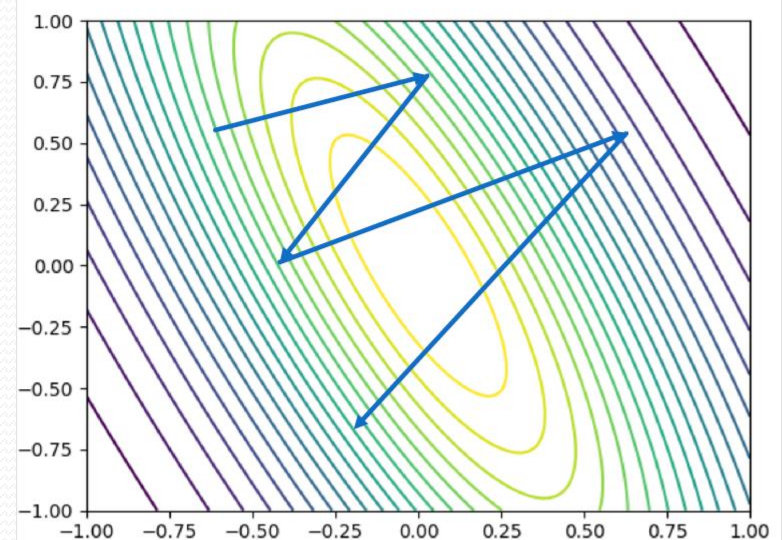
The goal of the learning phase is to minimize the loss function (due to all weights and biases of the network) by using a gradient approach:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\lambda}.\boldsymbol{\Delta L}(\boldsymbol{\theta})$$

Where λ is the learning rate. Take care how this parameter is quantified!



λ too low : long time to converge and be trapped in local optimum

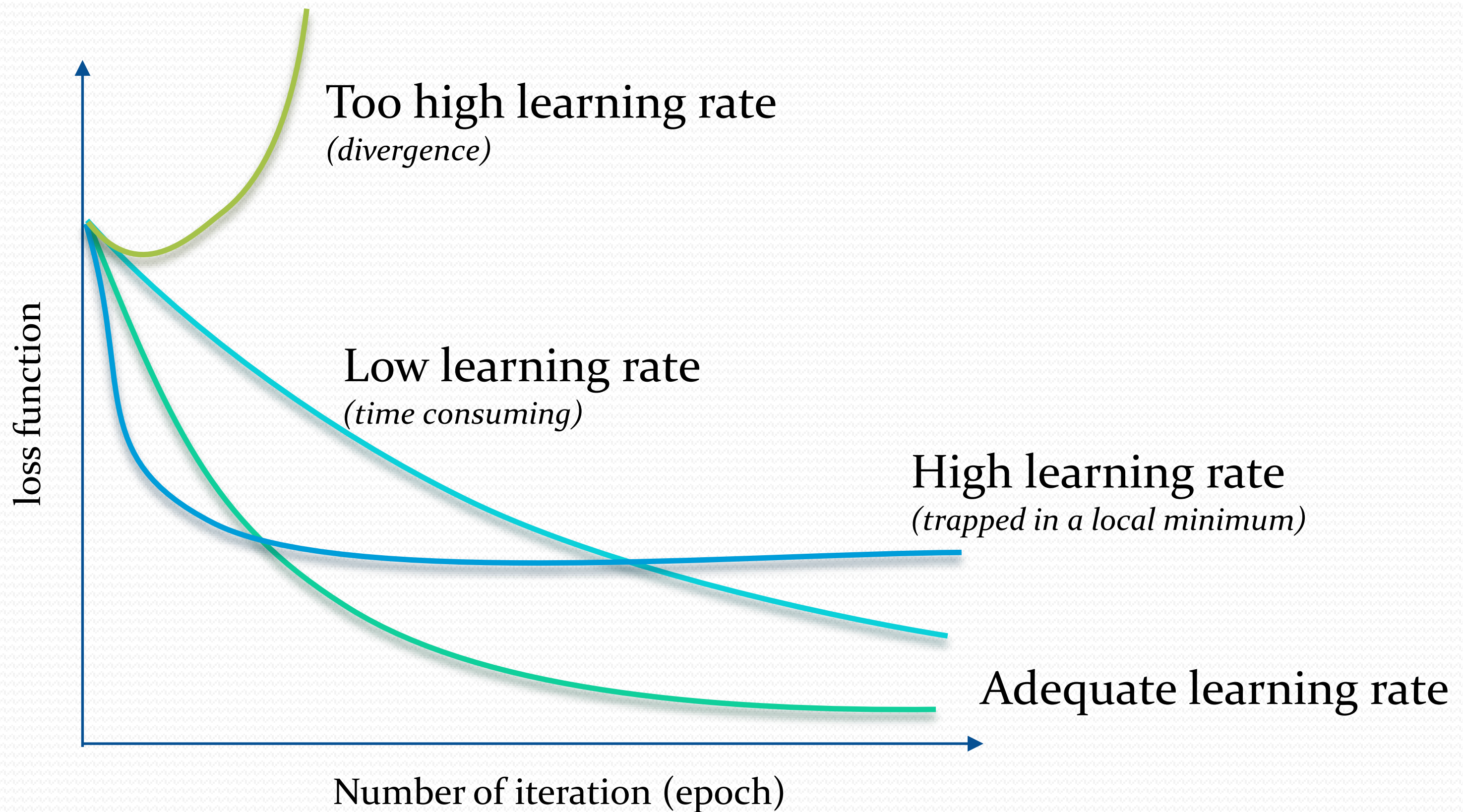


λ too big : chaotic exploration with big risk of divergence!

It is possible to define dynamic λ, decreasing during the convergence. To avoid getting stuck by a local minimum, a momentum can be defined

Loss function evolutions

Plotting the evolution of the loss during the training is a good way to adjust its main parameter : the learning rate λ.



Gradient Descent - Strategies to calculate it

Three main strategies are used to calculate the loss gradient $\boldsymbol{\Delta L(\theta)}$ and so to train the system:

- **Batch Gradient Descent**: the gradient is calculate on the whole set of database. This approach is, of course very long, but it increases the convergence
- **Stochastic Gradient Descent**: the gradient is calculate on each input of the database. This increases the speed of the (weights, biases) updates but risks to face instabilities
- **Mini-Batch Gradient Descent**: batches are generated randomly from the database to mix the two previous strategies: all the data of a batch are used to evaluate the gradient which is used to update the ANN's parameters.

Back propagation - Last Layer

For the kth weight in the last layer, with the ith case, the gradient is defined:

$$\frac{\delta L_i}{\delta w_k^{[n]}} = \frac{\delta L_i}{\delta \widehat{y_{ik}}} . \frac{\delta \widehat{y_{ik}}}{\delta z_k^{[n]}} . \frac{\delta z_k^{[n]}}{\delta w_k^{[n]}} = \frac{\delta L_i}{\delta \widehat{y_{ik}}} . g'\left(a_k^{[n]}\right) . a_k^{[n-1]}$$

$$\frac{\delta L_i}{\delta b_k^{[n]}} = \frac{\delta L_i}{\delta \widehat{y_{ik}}} . g'\left(a_k^{[n]}\right)$$

Effect of the considered weight on the input

Effect of the input of the neuron on its output

Effect of the considered neuron on the global loss

Example with the MSM + Sigmoid:

$$\frac{\delta L_i}{\delta w_k^{[n]}} = (\widehat{y_{ik}} - y_{ik}).(\widehat{y_{ik}}.(1 - \widehat{y_{ik}})) . a_k^{[n-1]}$$

Then the new weight is calculated:

$$w_k^{[n]} = w_k^{[n]} - \boldsymbol{\lambda}. (\widehat{y_{ik}} - y_{ik}).(\widehat{y_{ik}}.(1 - \widehat{y_{ik}})) . a_k^{[n-1]}$$

Back propagation - Hidden Layer(s)

For the hidden layers ($l \neq n$), the weight have more way to impact the gradient. This equation gives the way to modify all weights:

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \lambda .\, e_i^{[l]} .\, a_j^{[l-1]}$$

With $a_j^{[0]}$ is the x_{ij} (the jth element of the ith case)

Where $e_j^{[l]}$ is defined in a recursive way:

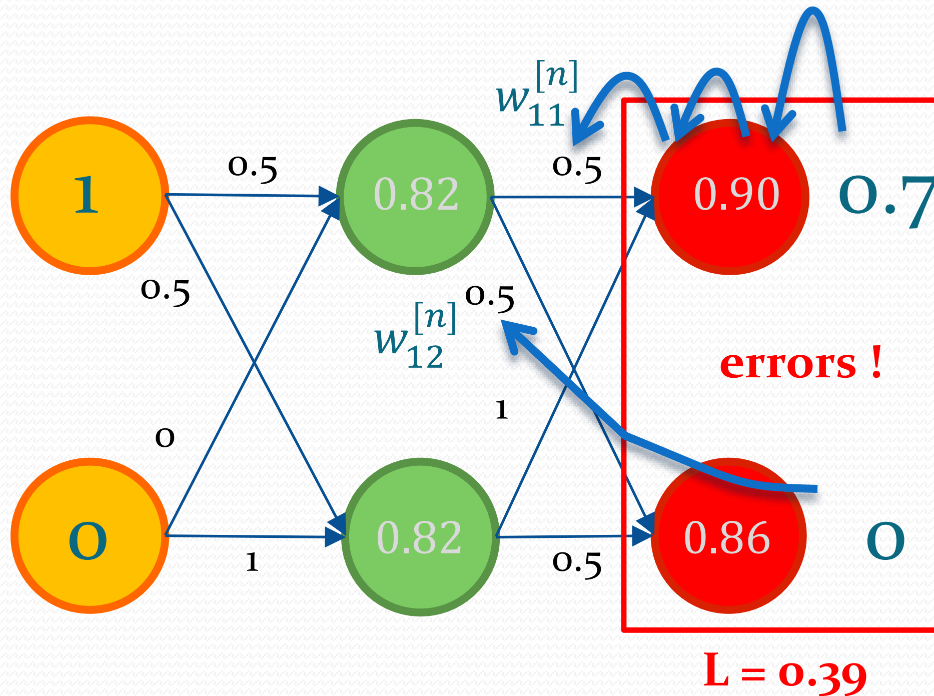
$$e_j^{[l-1]} = g'\left(a_j^{[l-1]}\right) . \sum_k w_{kj}^{[l]} .\, e_k^{[l]}$$

$$e_j^{[n]} = (\widehat{y_{ij}} - y_{ij}) .\, g'(a_j^{[n]})$$

Back propagation - Example

Considering the total error $\boldsymbol{L}$ is calculated with: $L = \frac{1}{2} . \sum_{i,j} (\widehat{y_{ij}(\theta)} - y_{ij})^2$

Learning rate λ = 0.1



New_$w_{11}^{[n]}$= $w_{11}^{[n]}$- $\boldsymbol{\lambda}$. $(\widehat{y_{ik}} - y_{ik}).(\widehat{y_{ik}}.(1 - \widehat{y_{ik}}))\,.\,a_k^{[n-1]}$

New_$w_{11}^{[n]}$= 0.5-0.1* (0.9−0.7) *(0.9*(1 - 0.9)) * 0.82

New_$w_{11}^{[n]}$= **0.499**

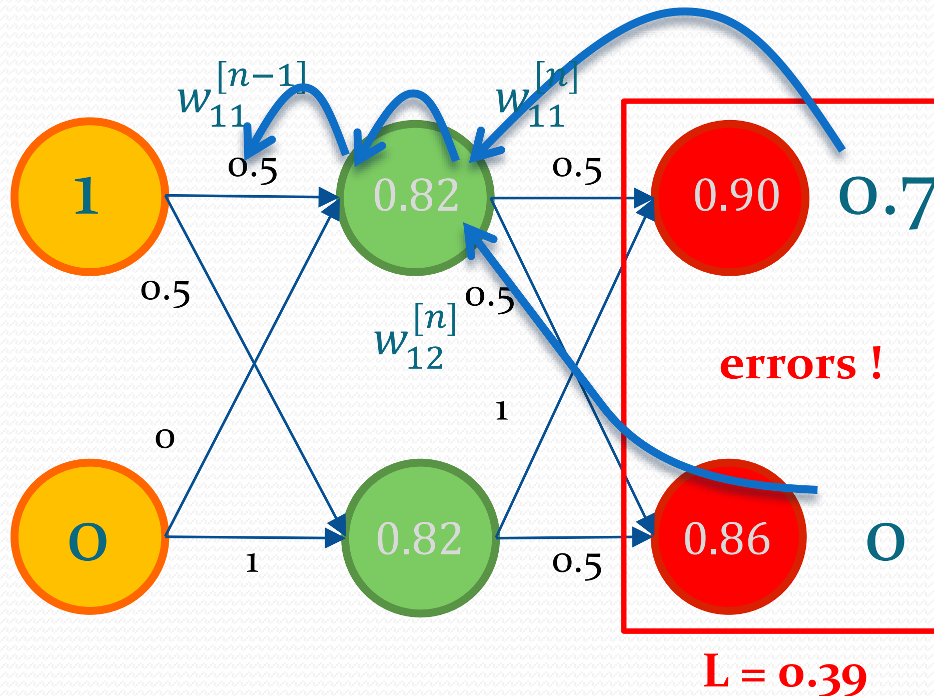
New_$w_{12}^{[n]}$= 0.5-0.1* (0.86 − 0) *(0.86*(1 - 0.86)) * 0.82

New_$w_{12}^{[n]}$= **0.491**

Back propagation - Example

Considering the total error $\boldsymbol{L}$ is calculated with: $L = \frac{1}{2}.\sum_{i,j}(\widehat{y_{ij}(\theta)} - y_{ij})^2$

Learning rate λ = 0.1



New_$w_{11}^{[n-1]}$= $w_{11}^{[n-1]}$- λ. $e_0^{[1]}$. $a_0^{[0]}$

$e_0^{[1]} = g'\left(a_0^{[l-1]}\right).\sum_k w_{kj}^{[l]}.\, e_k^{[l]}$

$e_0^{[1]} = (0.82 * (1 - 0.82)).\sum_k w_{kj}^{[l]}.\, e_k^{[l]}$

Or $e_k^{[l]}$ were already calculated on the previous step:

$e_0^{[l]}$= (0.9-0.7)*0.9*(1-0.9) = 0.018

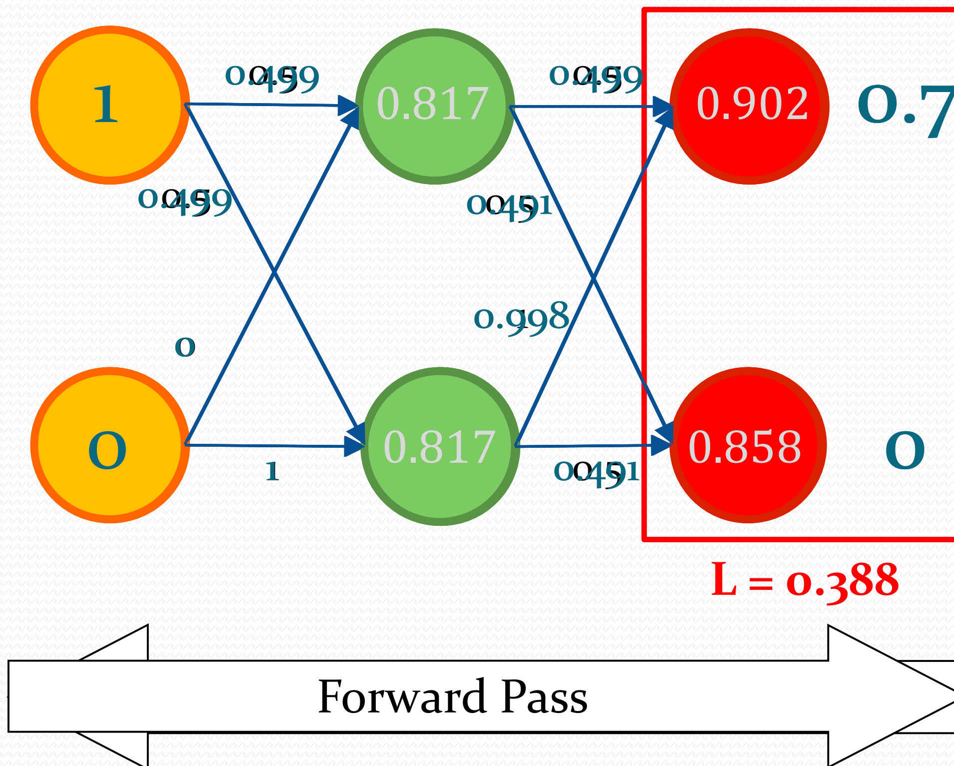
$e_1^{[l]}$= (0.86-0)*0.86*(1-0.86)= 0.103

$e_0^{[1]}$= 0.148.(0.018*0.5+0.103*0.5) = 0.009

New_$w_{11}^{[n-1]}$= 0.5- 0.1* 0.009 * 1 = **0.499**

Back propagation - Example

After applying the backpropagation, we can see the effect of these new weights on the loss function. The same work must be done on the biases.



When the feed forward was applied on the network…

Evaluate the new weights and biases through back propagation

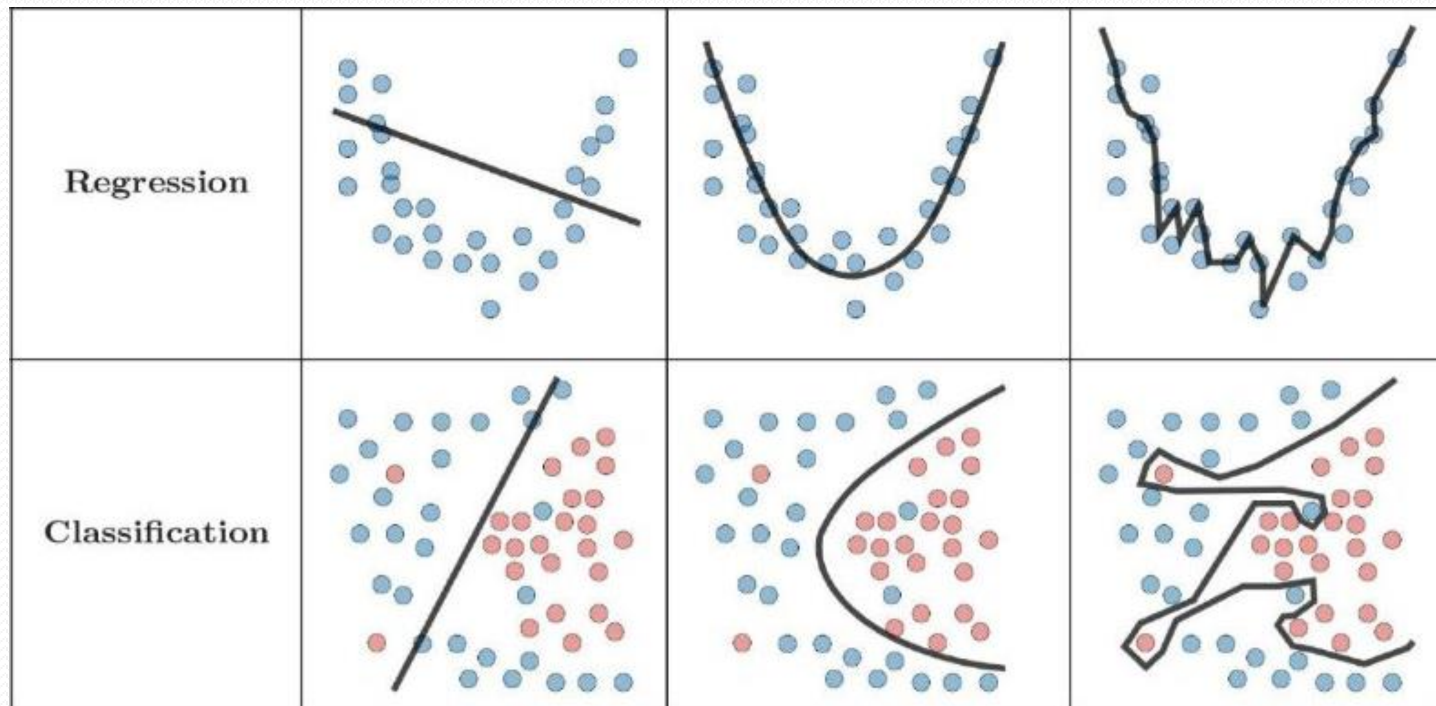
Redo the feed forward on this new configuration and assess the new loss function

… and continue until reaching the convergence or the stopping criteria…

Issue - Overfitting

Overfitting happens when the loss function is good (very low) on the learning set, but is not able to generalize its predictions to additional or unseen examples.

This problem, in regression purpose, is very close to polynomial approximation issues (selection of the right polynomial degree).

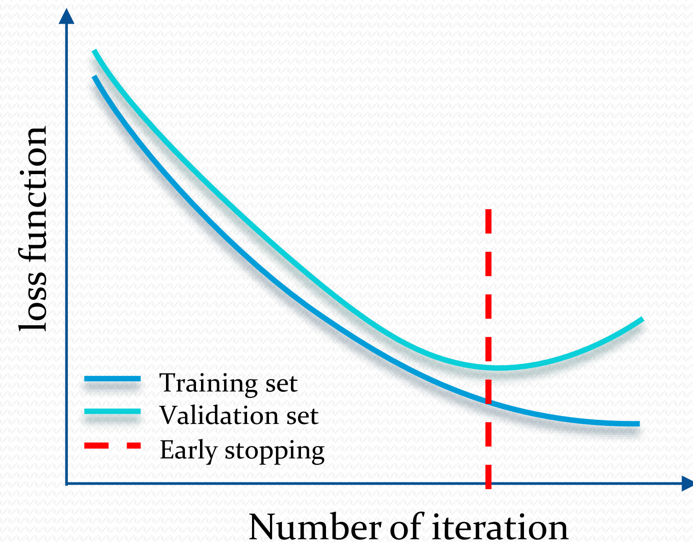


Underfitting **Overfitting**

Overfitting - Solutions

To avoid the overfitting situation, two main solutions are used:

- **Early Stopping:** Among the dataset, some are kept in a validation set that is not used by the ANN for its learning phase. The loss function is calculated on both train and validation sets. If after several iterations the loss function of validation set didn't improve, stop the training.



- **Dropout:** To avoid having neurons specializing on a particular example, the dropout approach randomly switch off some neurons during the training.

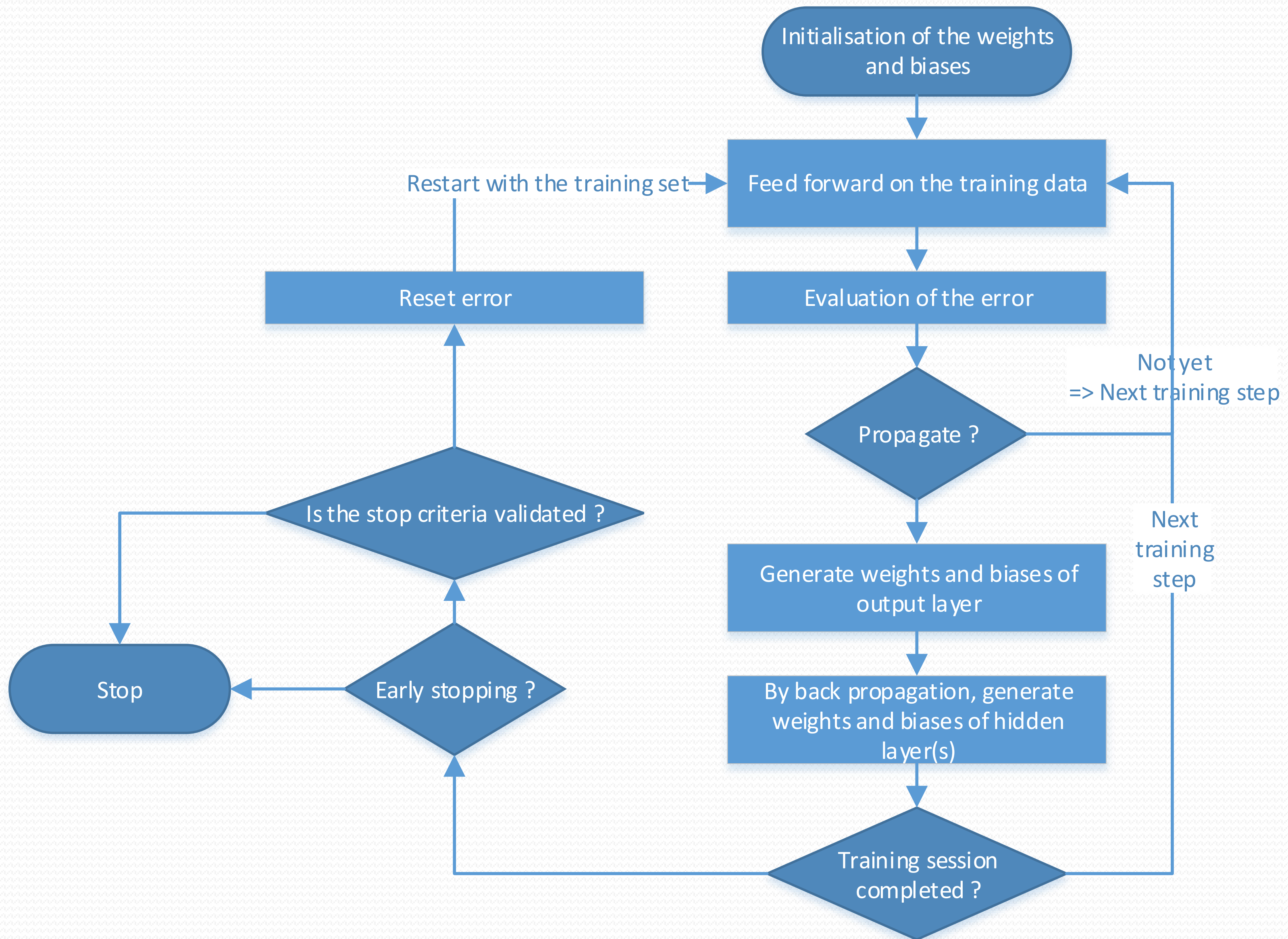
Then, how to split the dataset ?

To train an ANN, to test it and to avoid it from over-training, three pack of points are required:

Where:

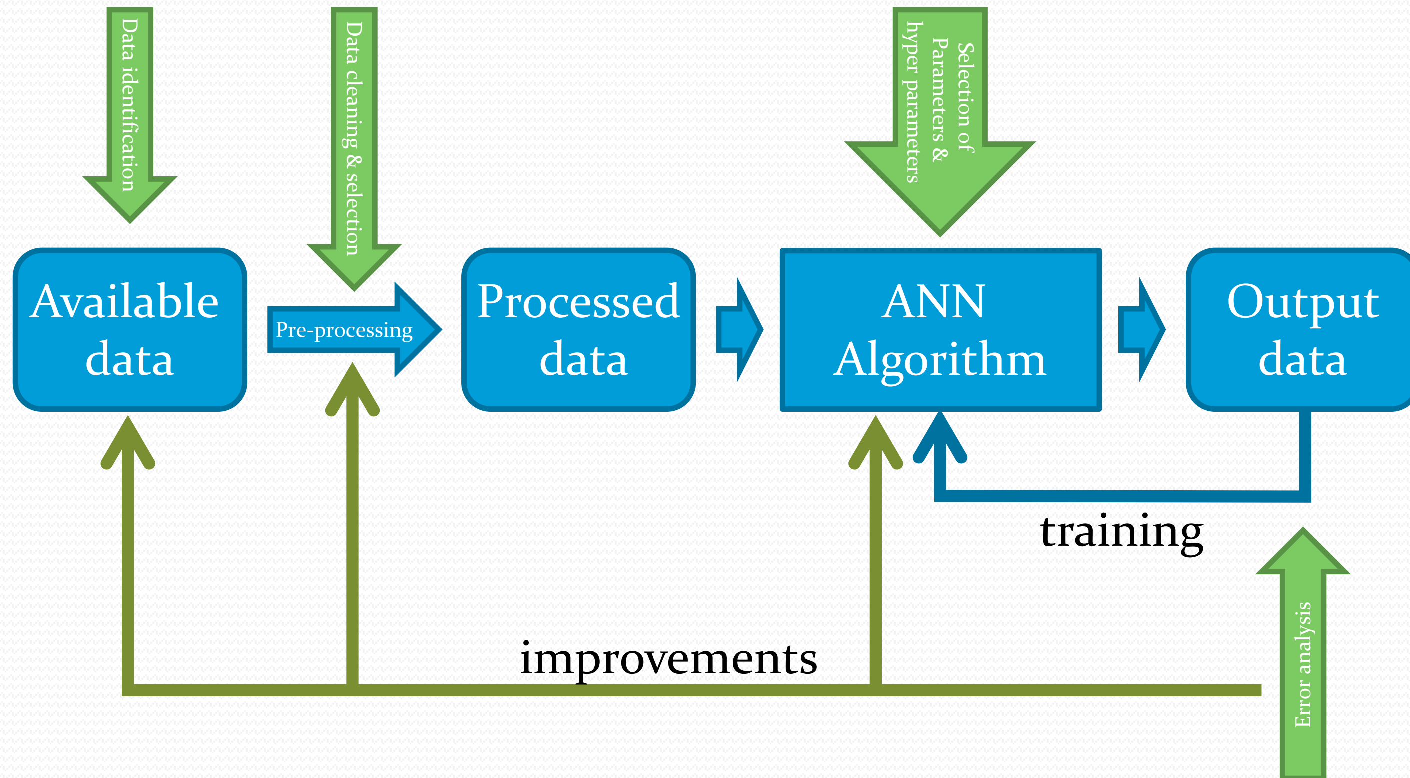
- The **training set** is used to train the ANN (for the back propagation purpose)
- The **validation set**, which is composed of unseen cases, is used to check during the training if the system is facing over-fitting or over-training issues
- The **test set** is used when the training is complete to verify the performances of the trained ANN. Try to have a representative set for the final validation (points well distributed in the domain)
- The distributions between these sets are generally following these rates: **60% / 20% / 20%**

Neural Network - Complete Algorithm



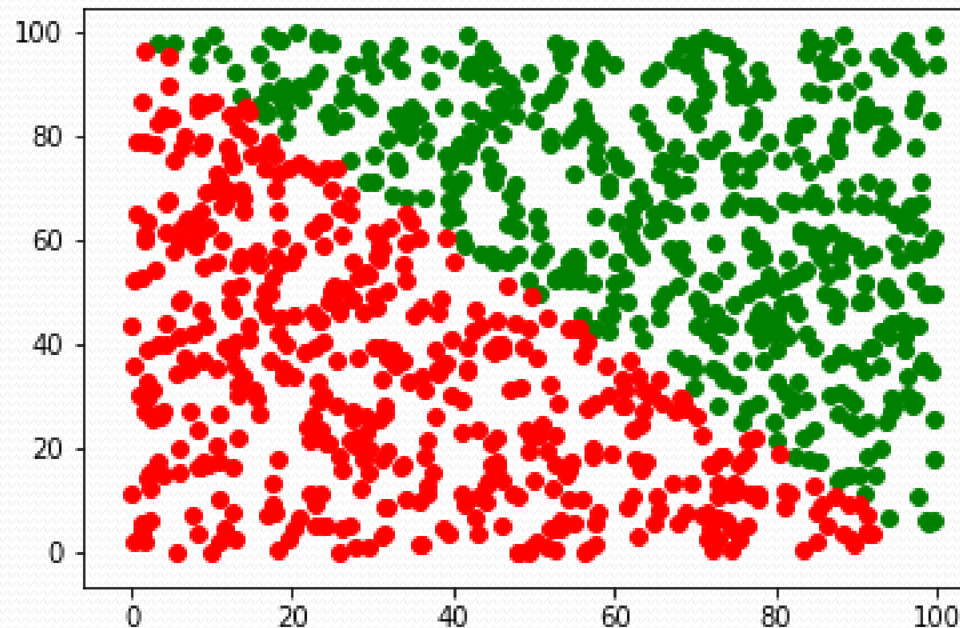
You and the Machine Learning process

You have to take a lot of decisions to build and improve the ANN meeting your expectations and the type of data you want to process:



Exercice : A simple classification Perceptron

Design and implement in Python (without any ANN oriented lib) a simple perceptron (no hidden layer) being able to make classification of a set of points.



The points are considered class_0 if the sum or their coordinate is less than 100 (in red), class_1 in the other case (in green). The range of the analysed points is [0,100] for both x-axis and y-axis.

Exercice : A simple classification Perceptron

Implement a simple Perceptron in python (with numpy and matplotlib) meeting these requirements:

- 2 input neurons (x and y values)
- 1 output neuron. Since the goal is to make classification between two classes, use a Boolean parameter.
- Activation function has to be selected among: sigmoid or hyperbolic tangent. Prefer sigmoid with is easier to derivate.
- The initial weights and biases are generated randomly, as the dataset used to train the system. Take a training set of 250 points.
- Don't forget to normalize the input data ! (use `numpy mean` and `std`)

After implementing your Perceptron **validate it** by:

- Plot in a graph the error of your ANN at each step of its learning phase
- Define the criteria to stop the training process (error less than an a threshold, number of trainings performed…)
- Test the accuracy of your model with a set or randomly generated points in the same range [0, 100] which doesn't belong to the training set

Exercice : structure of the program

In order to guide your work, you can follow these different steps/goals:

- Generate of the training set (use `random` package)
- Define the variables needed to describe the parameters of the ANN
- Define the forward propagation function
- Define the loss function
- Define the back propagation function (you can choose the type of gradient descent)
- Include all this stuff into a loop to generate several updates of the ANN parameters

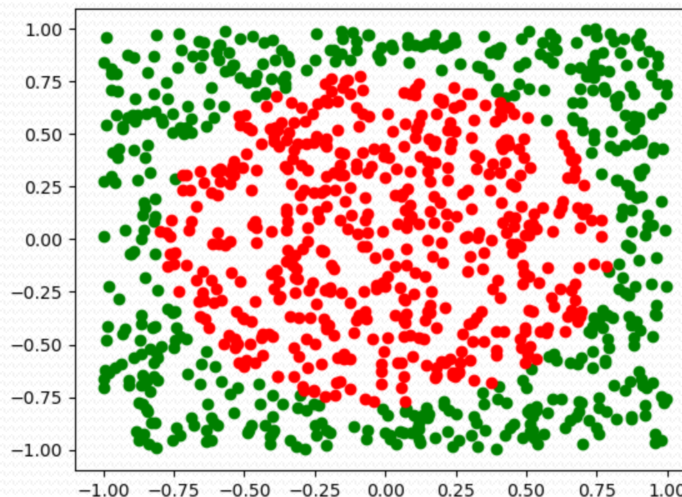
Exercice: Tips and real cases

With the sigmoid function and with the MSM, the updating of weights and biases in a perceptron (no hidden layer) follow this mathematical relation:

$$wi = wi \; - learning_rate * (prediction_error) * g'(z).xi$$

$$bi = bi \; - learning_rate * (prediction_error) * g'(z).1$$

As soon as your model is correct, try to train your perceptron on a new case, where the boundary is not linear (for instance $x^2 + y^2 < 1$). Analyse the efficiency of the perceptron in this new case.

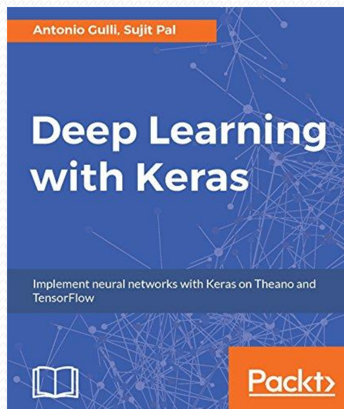


Useful Python Libraries - Keras

Several Python packages are available to design ANN and train them. Among them, *Tensorflow*, *sci-kit learn*, *Theano* and *keras* (with can be seen as an interface to ease the definition of an ANN) are very powerful (direct use of the computing power of GPU Graphic Power Unit) and easy to handle.

You can find several resources on Internet:

- Installation of Keras with Anaconda framework: Link and on SAVOIR
- Several examples of the definition and implementation of MLP (called sequential models in keras): Link (keras website), Link and Link



Several books are available (mainly in English) explaining how to use these Python libs.

Structure of a Keras program

The structure of an ANN designed in Keras follow this structure:

Import all the mandatory packages

Load and process the dataset (normalize + split into the 3 sets (training, validation and test))

Define the structure of the ANN (layers and their characteristics)

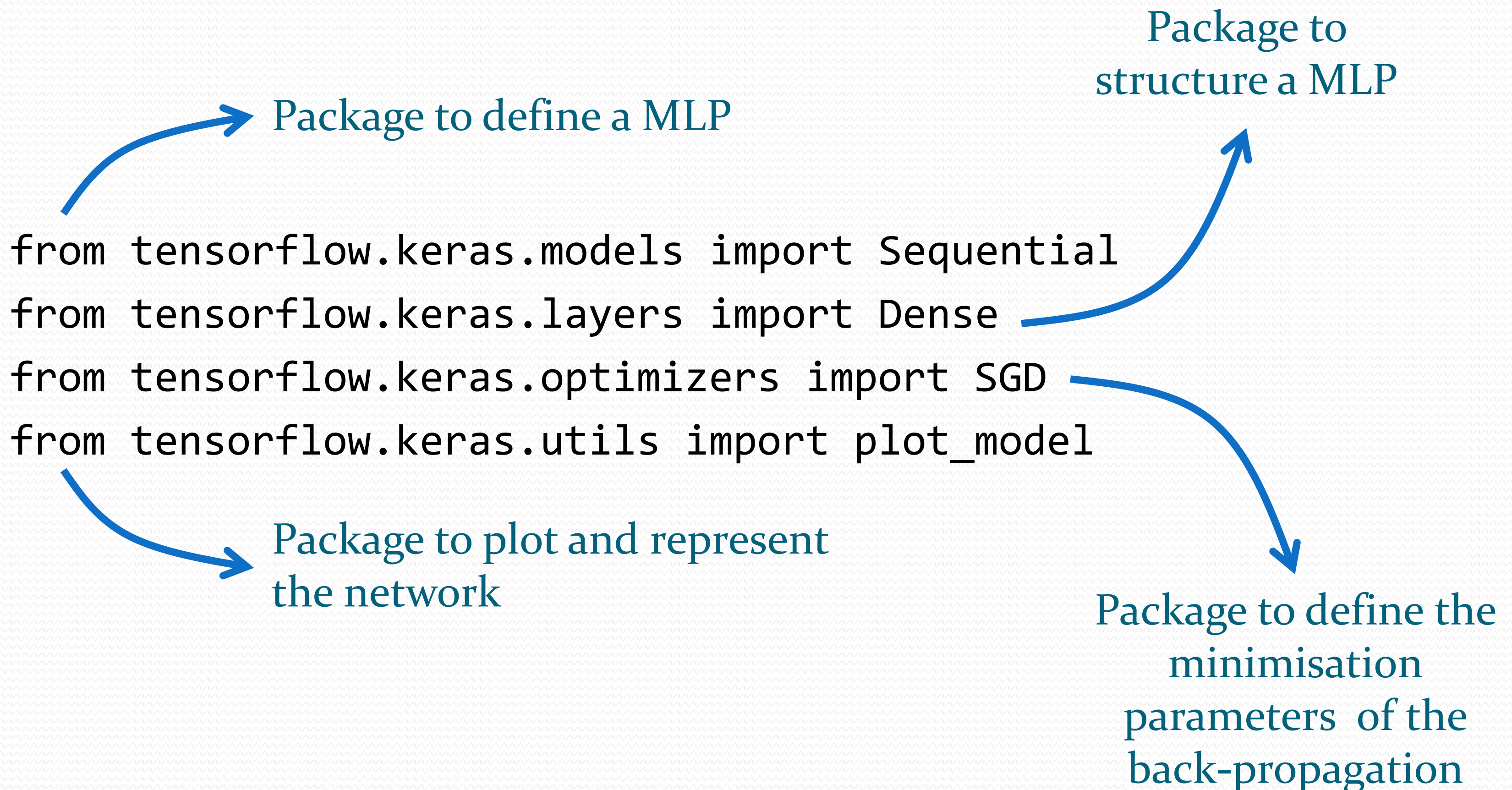
Run the learning phase (after defining the loss function, and the optimisation behaviour)

Consult the results and the ANN final configuration (weights and biases) and save the configuration

Use the ANN final configuration to new cases

Packages import

As usual in python, it is necessary to import a set of packages needed to define, train and use an artificial neural network:



Structuring the layer(s)

As soon as the MLP is defined (`Sequencial` Neural Network), you can easily add one by one its layers by describing their key parameters:

- its name
- the number of neurons allocated to the layer (first input parameter, **mandatory**)
- for the first layer (only), the number of input neurons (`input_dim`)
- the type of activation function (`activation` to select among: `'relu'`, `'sigmoid'`, `'linear'`, `'tanh'`,...) => Only one activation function can be allocated to all neurons of a layer
- the activation of bias or not (`True or False`)

Example:

```
my_network = Sequential()
my_network.add(Dense(20, input_dim = 10, activation = 'relu'))
my_network.add(Dense(4, activation = 'sigmoid'))
```

It's important to check that the model is consistent, for instance that `input_dim` is equal to the dimension of training data, or in the case of classification that the final layer has an activation function compatible (sigmoid, tanh...)

Check the structure

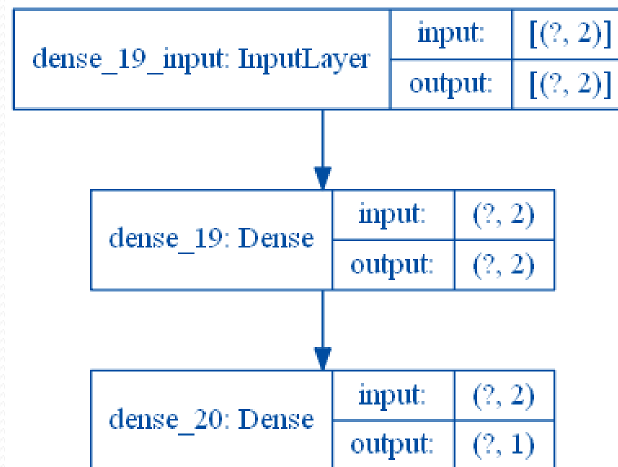
Several methods are available to access to the structure of the network:

- `Summary()`: generates the structure and print it on the debug console

```
In [71]: ANN.summary()
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_17 (Dense)             (None, 2)                 6
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
```

- If you want to have a graphical feedbacks of the structure of your ANN, it is possible to generate an image of it. To do so, import `plot_model` from `tensorflow.keras.utils`.

```
plot_model(my_network, to_file='model_plot.png', show_shapes=True)
```



Back-propagation parameters

As soon as the structure defined, it is necessary to define the learning parameters :

- The selection of the gradient parameters (learning rate and the momentum), describing the optimisation parameters:

  `Op_params = SGD(learning_rate, momentum)`, both are float numbers

- The selection of the loss function among the ones available
  - For regression: `'mean_squared_error'`, `'mean_absolute_error’`
  - For classification: `'binary_crossentropy’` (2 classes), `'categorical_crossentropy’` (more than 2 classes)

- The choice of the performance indicator (analysis of the accuracy of the ANN) among the ones available (`'accuracy'` and `'binary_accuracy'`)

Example:

```
op_params = SGD(0.1 , 0.9)
my_network.compile(loss='binary_crossentropy', optimizer= op_params,
metrics=['binary_accuracy'])
```

Learning definition and running

The network is now ready to be trained. To do so, the next step aims at defining and running the training. The method to use is `fit`, which is waiting for a set of input parameters:

- The input dataset (as a `numpy` or a python list of list) - **mandatory**
- The output dataset (same constraint than for the input) - **mandatory**
- The epochs (integer the number of iteration) – mandatory - **mandatory**
- The batch_size (integer defining the size of the mini-batch)
- The validation_split, float number representing the part of the dataset to use for the validation of the network. It is possible to give through the input parameter validation_inputs the value to consider as validation set.
- Verbose (0,1,2) to define the type of feedbacks given during the training (from nothing to complete)

Example:

```
my_network.fit(Inputs, Outputs, epochs = 1500, batch_size = 250)
```

Training run

During the training, if verbose parameter is defined to 2, you can see the evolution of the loss function and the accuracy function if you selected or defined them.

Current epoch (iteration of the training phase)

```
Epoch 394/1450
809/809 [==============================] - 0s 6us/sample - loss: 0.7102
Epoch 395/1450
809/809 [==============================] - 0s 5us/sample - loss: 0.7097
Epoch 396/1450
809/809 [==============================] - 0s 5us/sample - loss: 0.7095
Epoch 397/1450
809/809 [==============================] - 0s 5us/sample - loss: 0.7083
Epoch 398/1450
809/809 [==============================] - 0s 5us/sample - loss: 0.7101
Epoch 399/1450
809/809 [==============================] - 0s 5us/sample - loss: 0.7129
Epoch 400/1450
809/809 [==============================] - 0s 6us/sample - loss: 0.7072
Epoch 401/1450
809/809 [==============================] - 0s 6us/sample - loss: 0.7103
Epoch 402/1450
300/809 [==========>...................] - ETA: 0s - loss: 0.7323
```

Evolution of the batch processing

Evaluation of the loss function on the epoch

Training - Evolution of loss and accuracy 1/2

It is possible to have access to the evolution of the loss (of the training and validation sets) and metrics (accuracy selected on the parameters of the `fit` function) after the training of a network, since `Keras` stores all this data in an history.

For example:

```
my_history = my_network.fit(Inputs, Outputs, epochs = 1500, batch_size = 250)
```

To access this information, store into a variable the result of the fit function. From this variable you can have access to this history. Before requesting any data, you can firstly check the ones available by using the keys() parameter of the history.

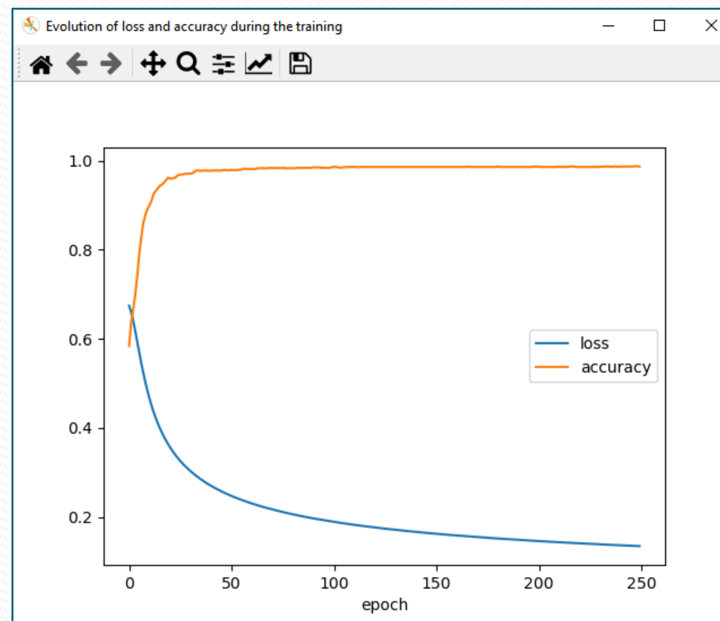
For example:

```
print(my_history.history.keys())
```

Training - Evolution of loss and accuracy 2/2

As soon as checked the data available in the history, you can access or plot them. As an example, this code will plot the evolution of the accuracy and loss function of the training defined on the previous slide:

```
plt.figure("Evolution of loss and accuracy during the training")
plt.plot(my_history.history['loss'], label="loss")
plt.plot(my_history.history['binary_accuracy'], label="accuracy")
plt.xlabel('epoch')
plt.legend(['loss', 'accuracy'], loc='right')
```



Use a trained network

As soon as the network is trained and its accuracy is considered satisfactory, it is possible to use it on new cases to predict their output or to classify them.

The instructions available in `keras` depends on the type of ANN:

- For regression problems: the method `predict` takes as inputs a batch of cases/points and generates the output vector (as a `numpy` matrix or vector) regarding the current configuration of the network
- For classification problems: the method `predict_classes` takes as inputs a batch of cases/points and generates the output vector regarding the current configuration of the network

Example:

```
my_network.predict(Inputs_to_test)
```

Classification – Confusion Matrix

In classification problems, it is possible to calculate the confusion matrix that synthetises the accuracy (good prediction) of the classifier and underlining the main risks :

- ✓ **Alpha**: predicting False something that is True
- ✓ **Beta**: predicting as True something that is False

		Prediction	
		True	False
True label	True		α
	False	β	

Classification – Confusion Matrix

To easily handle the confusion matrix, the sci-kit learn package is relevant and fully compatible with keras trained neural networks.

```
from sklearn.metrics import confusion_matrix
predictions = ANN.predict_classes(Inputs)
c = confusion_matrix(y_true, y_pred)
print(c)
```

```
[[44375     0]
 [    0 88750]]
```

Access to the parameters of the ANN

It is possible to have a look at the parameters (weights and bias) of each neuron of the network, to check the relative importance of a neuron (input or intermediate one). To do so, use the method `weights`.

In the case of a MLP composed by 1 hidden layer of 2 neurons and one output neuron, the result has this shape:

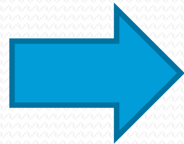
```
In [78]: ANN.weights
Out[78]:
[<tf.Variable 'dense_21/kernel:0' shape=(2, 2) dtype=float32, numpy=
 array([[1.6364022 , 7.908386  ],
        [0.50241685, 7.989181  ]], dtype=float32)>,
 <tf.Variable 'dense_21/bias:0' shape=(2,) dtype=float32, numpy=array([-1.0894496, -8.029093 ], dtype=float32)>,
 <tf.Variable 'dense_22/kernel:0' shape=(2, 1) dtype=float32, numpy=
 array([[ 0.5413507],
        [13.857591 ]], dtype=float32)>,
 <tf.Variable 'dense_22/bias:0' shape=(1,) dtype=float32, numpy=array([-6.8311167], dtype=float32)>]
```

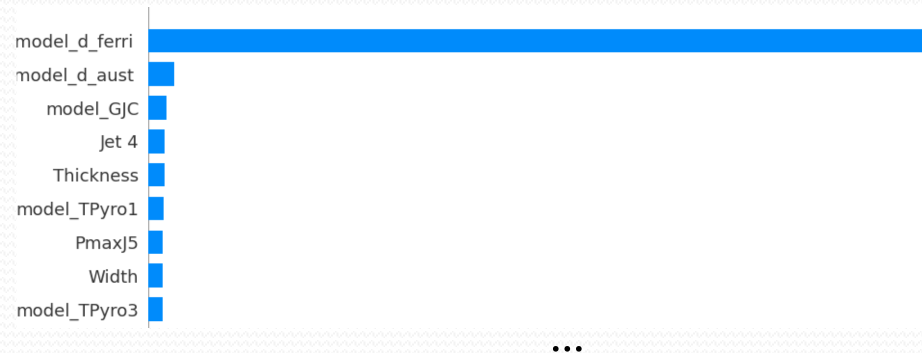
Input weights of each neuron composing the first hidden layer

Biases of each neuron composing the first hidden layer

Analyse features importance - Shap

Since the analysis of the weights is quite complicate, even more on complex model, several tools are available to analyse the importance of input neurons on the predictions.



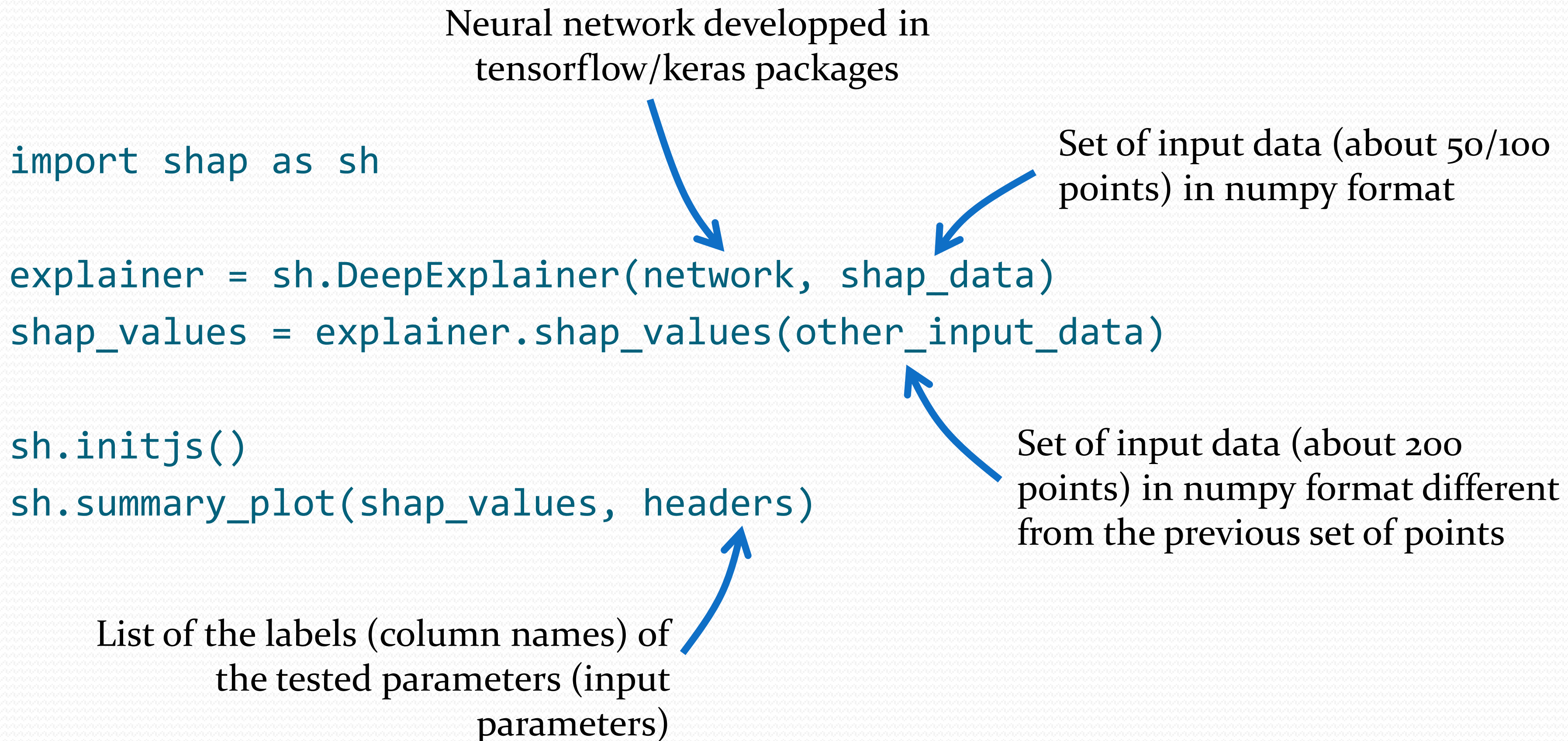


…

It can be relevant to compare this analysis with statistical analysis of the input data before training to detect some biases due to the network structure.

Analyse features importance - SHAP

Code example:



Save a trained network (to re-use after)

As soon as an ANN is trained and considered accurate, it is fortunately not necessary to re-train at each use. It is then possible to save and load a network already trained and configured.

First of all, import the loading functions to your python program:

```
from tensorflow.keras.models import load_model
```

From now, you can easily:

- Save your trained model with the function `save(filename)`
- Load your trained model with the function `load_model(filename)`

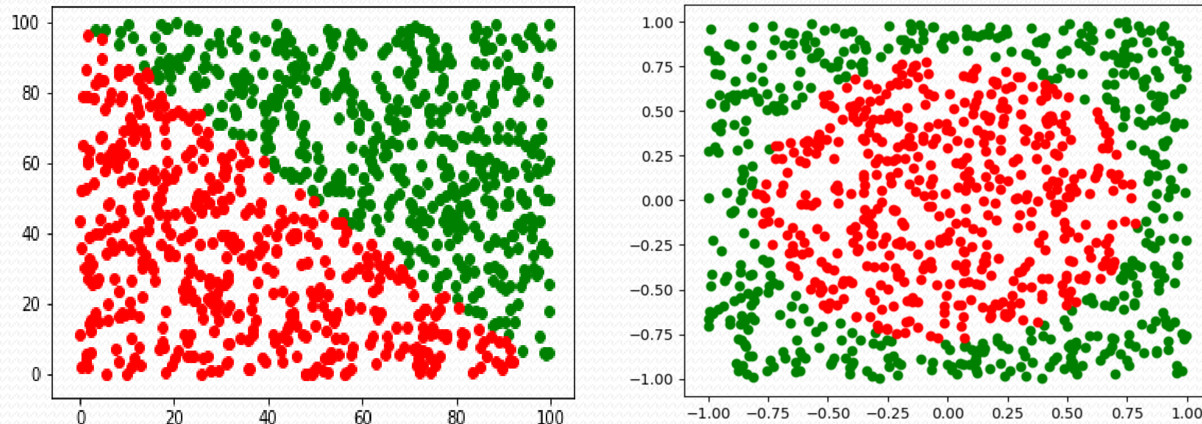
Example:

```
my_network.save("My_network.keras")
my_new_network = load_model("My_network.keras")
my_new_network.summary()
…
```

Exercices

By using `keras` + `numpy` packages, please complete these exercises:

- Design a simple perceptron where the separator is linear
- Design a perceptron where the separator is not linear (see the previous exercise with the separator is a circle)
- From this perceptron, add hidden layers to increase the accuracy of this classification neural network



For these 3 cases, check the structure of the design network, see the value of the weights and biases and save them.

To go further… documents and videos

Videos on YouTube:

- 3blue1brown propose an interesting an well animated explanation of the working of a neural network for hand writing recognition): Link
- Open course proposed by MIT: See more particularly videos 12a and 12b in this playlist: Link

Blogs and websites:

- Keras webpage: Link
- Complete guide to ANN (not only MLP): Link
- Blog about ANN and deep learning: Link
- Another blog about machine and deep learning: Link
- Clear explanation of backpropagation on a example: Link
- Several datasets proposed: Link

Several books are available in the library (or online see ENI edition)!