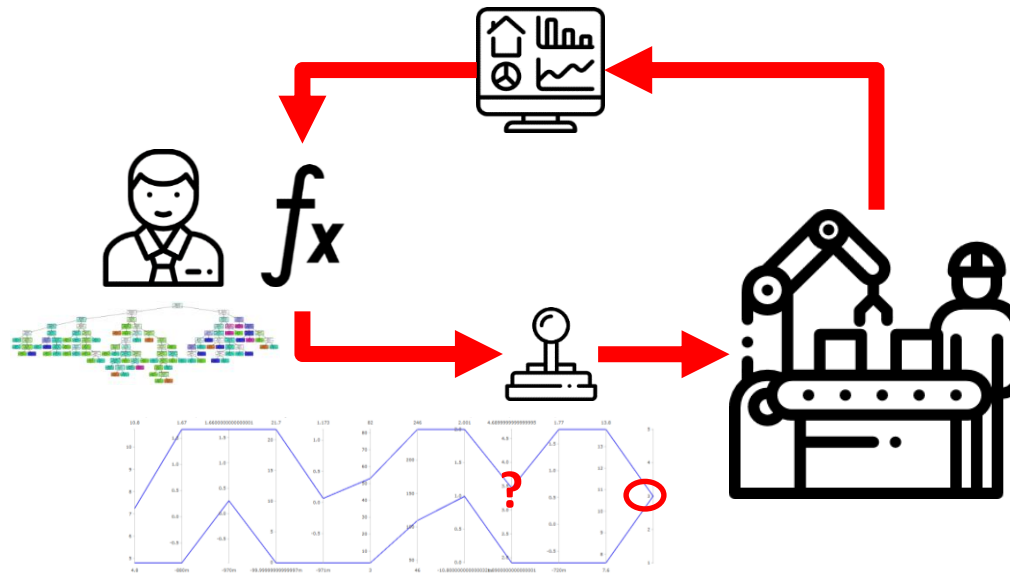


Comment améliorer votre production via les techniques de Machine Learning?

Data science pour la performance des systèmes industriels



Jean-Yves DANTAN
Lazhar HOMRI
Wahb ZOUHRI
Alain ETIENNE

A QUOI CA SERT ?

Découvrir des relations entre variables.

Clustering
Regrouper un ensemble d'objets en fonction de leurs caractéristiques.

Classification
Prédire la catégorie (classe) d'une nouvelle observation.

Boîte à outils de Data science

Trouver une fonction qui modélise les données.
Prédire

A quoi ça sert ?

- ✓ Finalités des techniques de machine learning

COMMENT DEPLOYER ?

Exemple:

41 Paramètres Process (PP)
12 Caractéristiques Produit (KC)
809 points
42877 données

COMMENT DEPLOYER ?

Identifier les causes d'une non-conformité Produit

Comparison

Paramètres discriminants de la spécification ?

Paramètres influents KC7

Pourquoi et comment déployer des techniques d'IA ? Comment exploiter les résultats ? avec des applications

- ✓ Techniques d'association: ACP,...
- ✓ Techniques de clustering: K-MEANS, KNN, ...
- ✓ Techniques de classification: Decision tree, Random Forest, SVM, ...
- ✓ Techniques de régression: régression polynomiale, RFR, ANN, ...

Comment ça marche ?

avec des mises en œuvre

Comment ça marche ?

Analyse de données - Analyse en Composantes Principales

Matrice des données (individus selon les lignes et grandeurs selon les colonnes)

Matrice des données réduite

Matrice des corrélations

Valeurs propres (ordonnées dans l'ordre décroissant)

Vecteurs propres correspondants

Matrice de passage des colonnes vers les vecteurs propres

Matrice des données dans la nouvelle base des vecteurs propres (possibilité de normaliser e_i)

$$M = \begin{pmatrix} x_{1j} \\ \vdots \\ x_{mj} \end{pmatrix}$$

$$m = \begin{pmatrix} x_{1j} - \bar{x}_j \\ \vdots \\ x_{mj} - \bar{x}_j \end{pmatrix}$$

$$r = \begin{pmatrix} \sum_{j=1}^n x_{1j} x_{1j} \\ \vdots \\ \sum_{j=1}^n x_{mj} x_{mj} \end{pmatrix}$$

$$A = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_p \end{pmatrix} \quad \det(A - \lambda I) = 0$$

$$e_i = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_p \end{pmatrix} \quad r e_i^2 = \lambda_i e_i^2$$

$$P = \begin{pmatrix} | & & | \\ e_1 & & e_p \\ | & & | \end{pmatrix} \quad m^2 = m P$$

$$m' = \begin{pmatrix} | & & | \\ m & & m \\ | & & | \end{pmatrix}$$

Comment exploiter les résultats

Prédire les non-conformités Produit

Traitement Random Forest

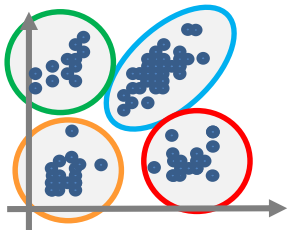
Analyse

Résultats



Clustering

Regrouper un ensemble d'objets en fonction de leurs caractéristiques.

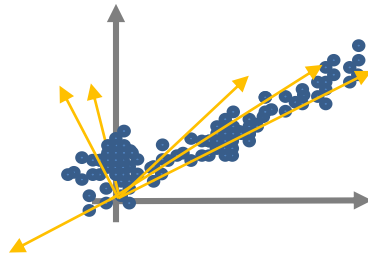


K-moyennes ✓



Association

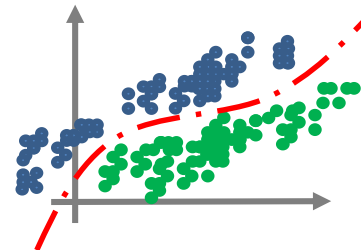
Découvrir des relations entre variables.



ACP ✓

Classification

Prédire la catégorie (classe) d'une nouvelle observation.



Arbre de décision ✓

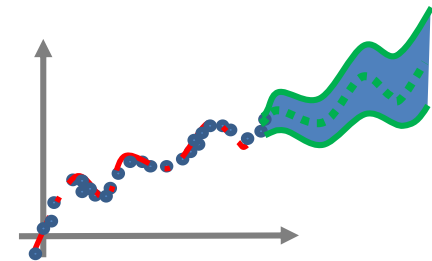
Logistic regression ✓

KNN, SVM ✓

Réseaux de neurones ⌚

Régression

Trouver une fonction qui modélise les données.
Prédire



Réseaux de neurones ⌚

1. Neural Network - Introduction

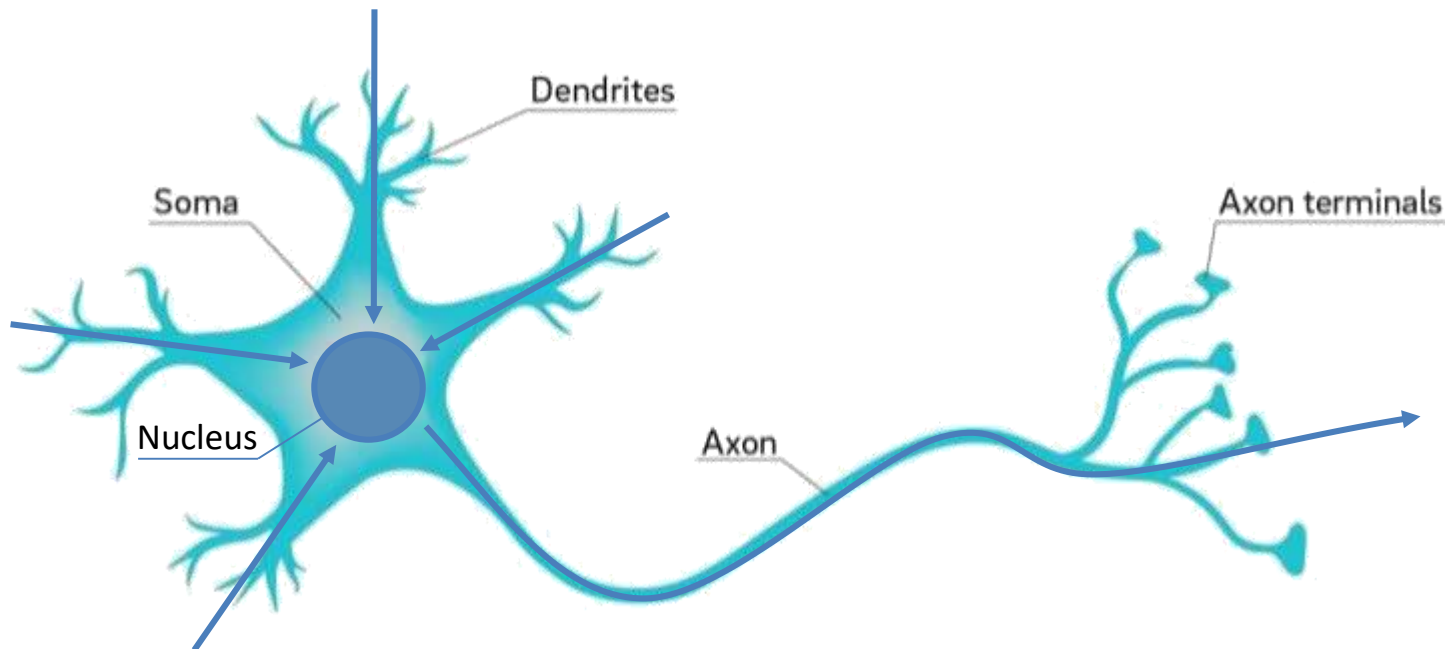




This training is a small introduction of ANN behaviour and the way to design and play with simple ones. To design real solutions a lot of steps need to be mastered !

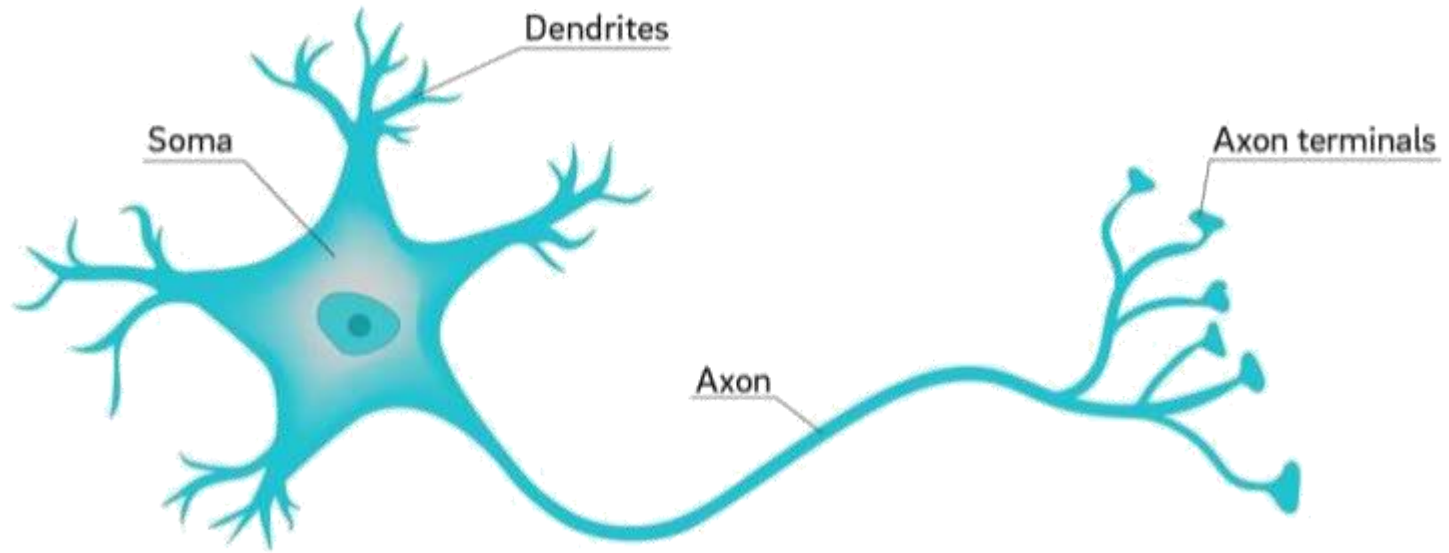
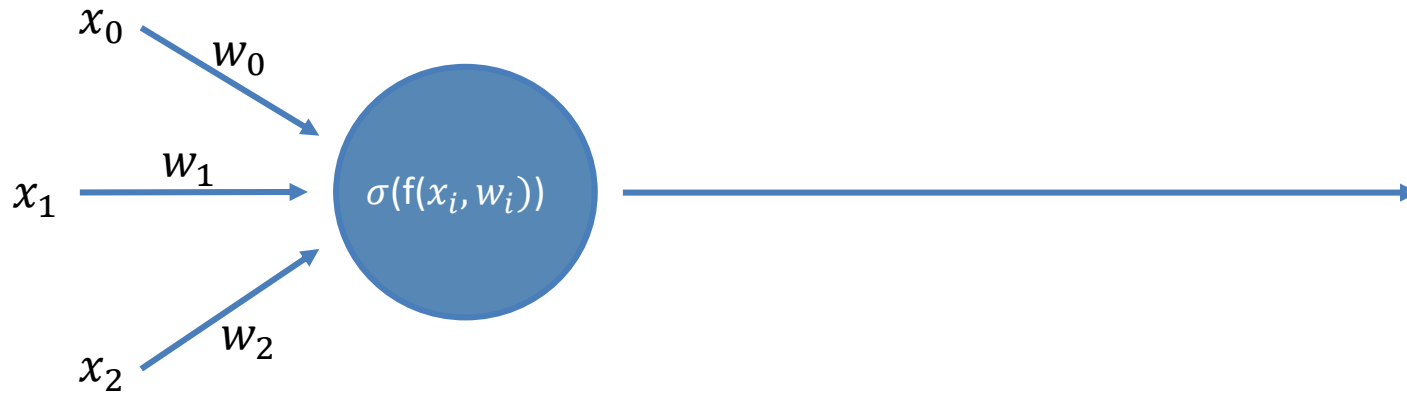
We have about 100 billions neurons. Each is connected to about 100 000 others. How does it work ?

- 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



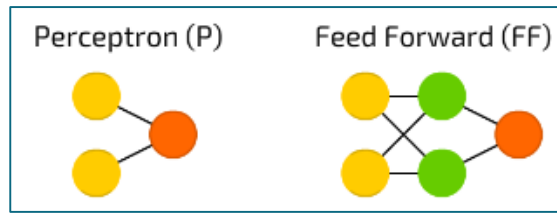
Neuron: From biological to artificial

The artificial neurone has the same structure and behaviour:

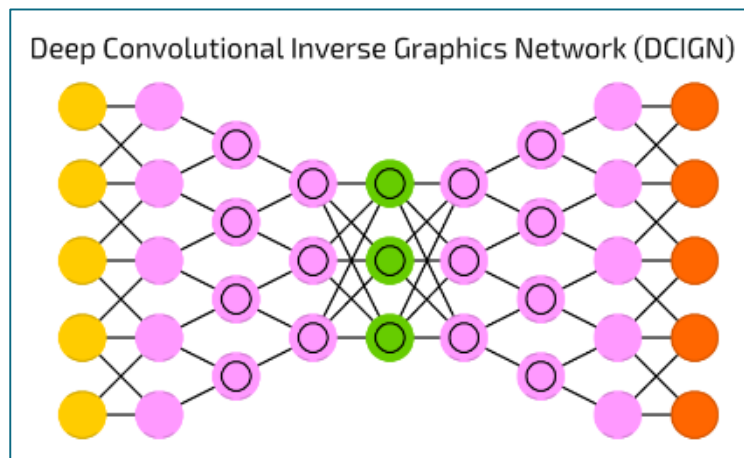


A lot of different structure of Neural Networks are available in the literature:

- From the simple and/or classical ones:



- To more complex ones:



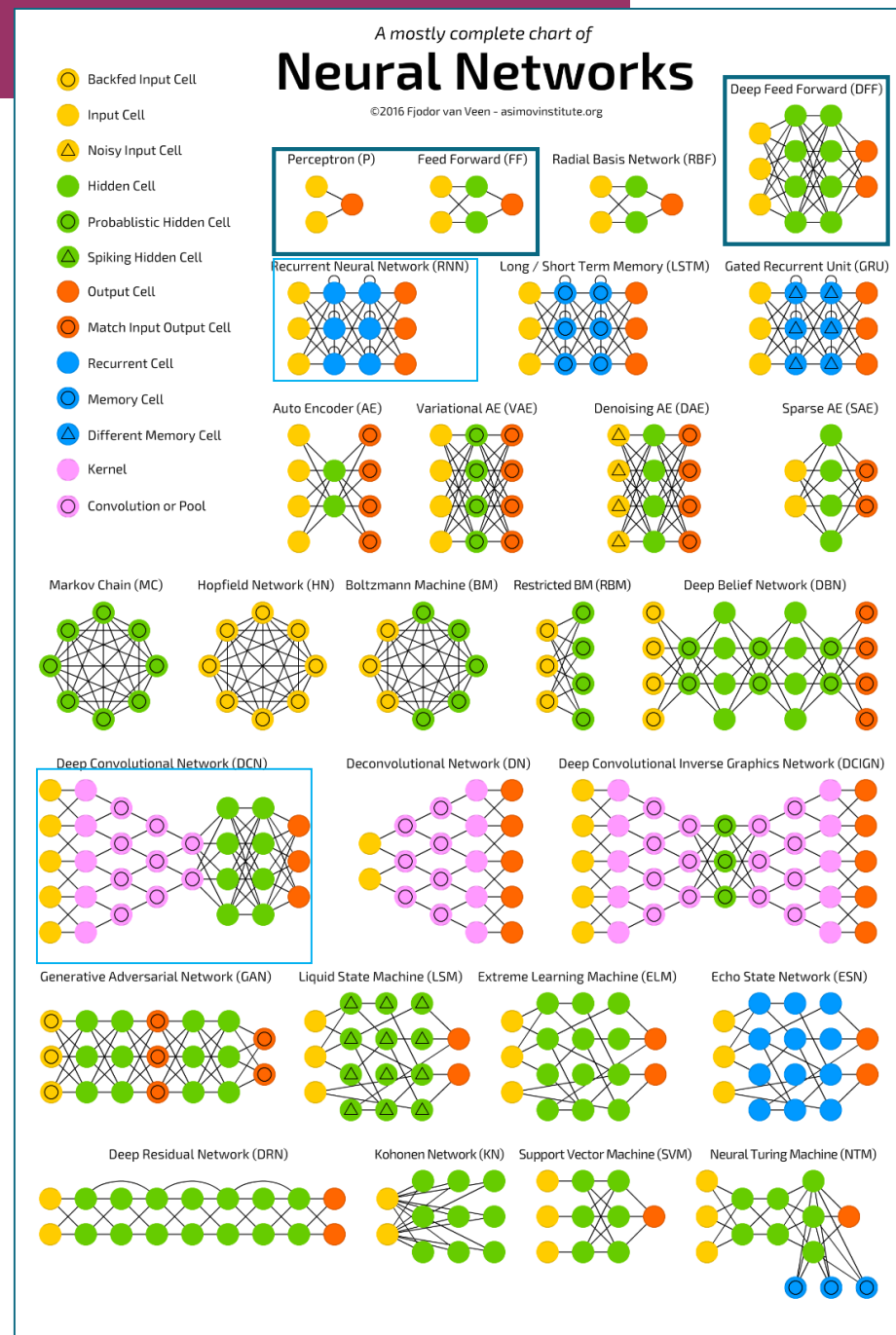
Zoo of neural networks

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

Several are dedicated to a particular goal, for instance:

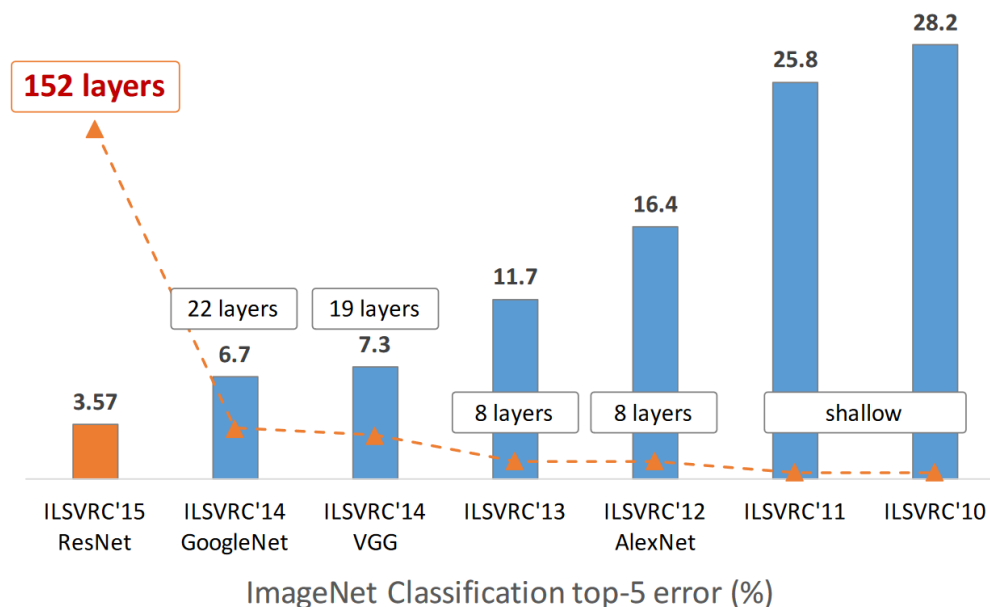
- **Recurrent Neural Network (RNN)** are often used to manage sequence prediction problems (text, language...)
- **Convolutional Neural Network (CNN)** are mainly used to process images

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



Two main improvements ease the performances of ANN:

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

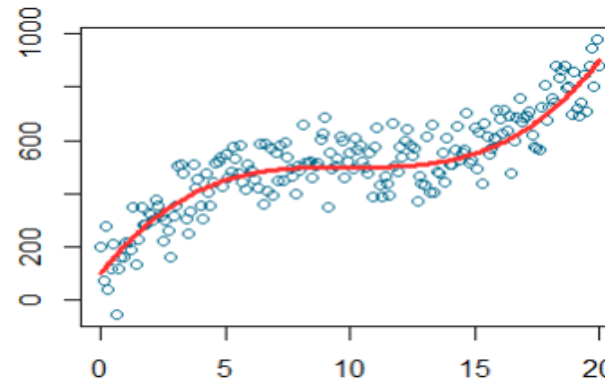


Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

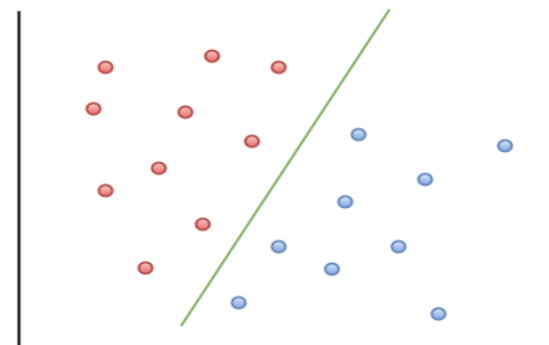
- The amount of data available is strongly increasing (more captors, Internet, IoT, connected devices...) => more cases to learn by the ANN which is a supervised learning approach
- New algorithms and approaches

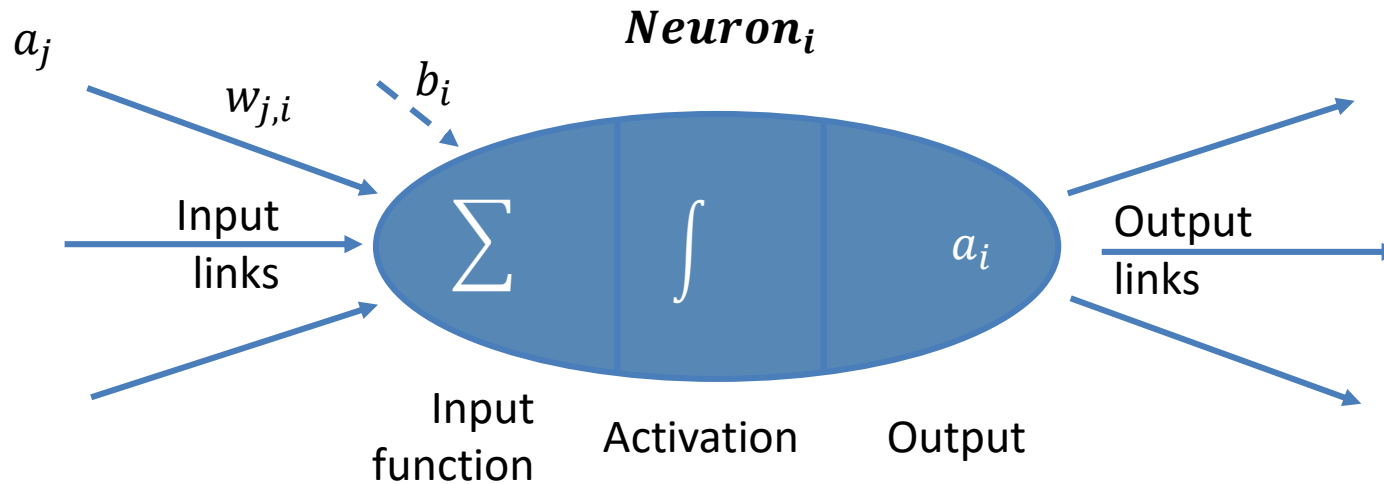
MLP are commonly selected to solve classical data problems:

- **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)



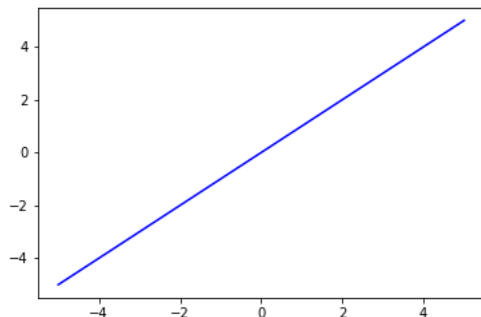


$$a_i = g(b_i + \sum a_k \cdot w_{k,i})$$

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

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

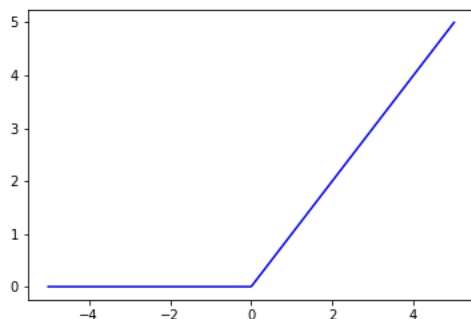


$$u = x$$

Linear function

Dedicated to regression

No transformation

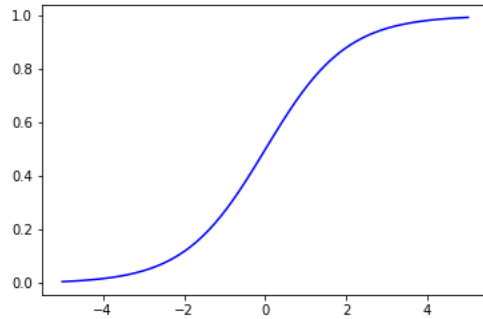


$$u = \max(0; x)$$

ReLU (Rectified Linear Units)

Filter negative values

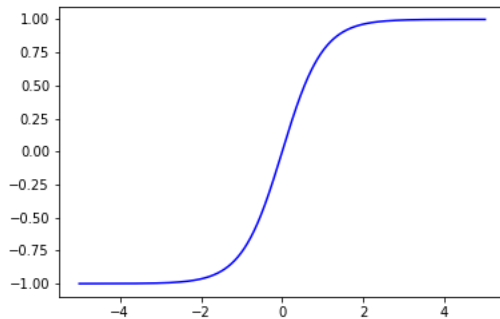
Use in Deep Learning (easy to calculate (evaluation and derivation))



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

Sigmoid

Move the activation in [0, 1] range
Use for Classification, not relevant for
hidden layers



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

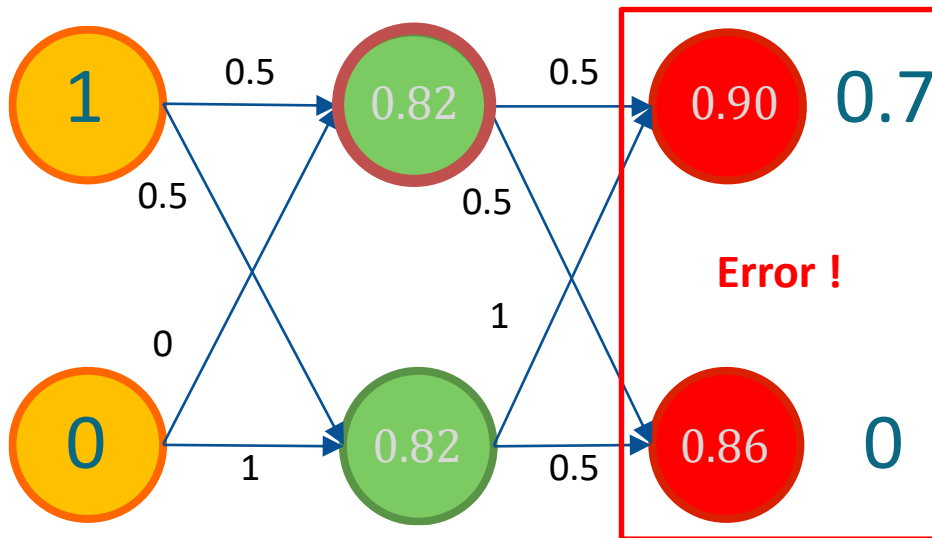
Hyperbolic Tangent

Move the activation in [-1, 1] range
Use for Classification, not relevant for
hidden layers

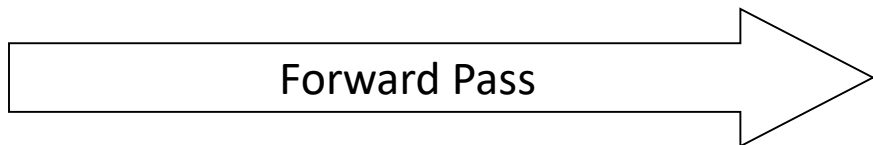
Forward Pass

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



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

The loss function ($L(\theta) = f(\widehat{Y}_i, Y_i)$) 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**:

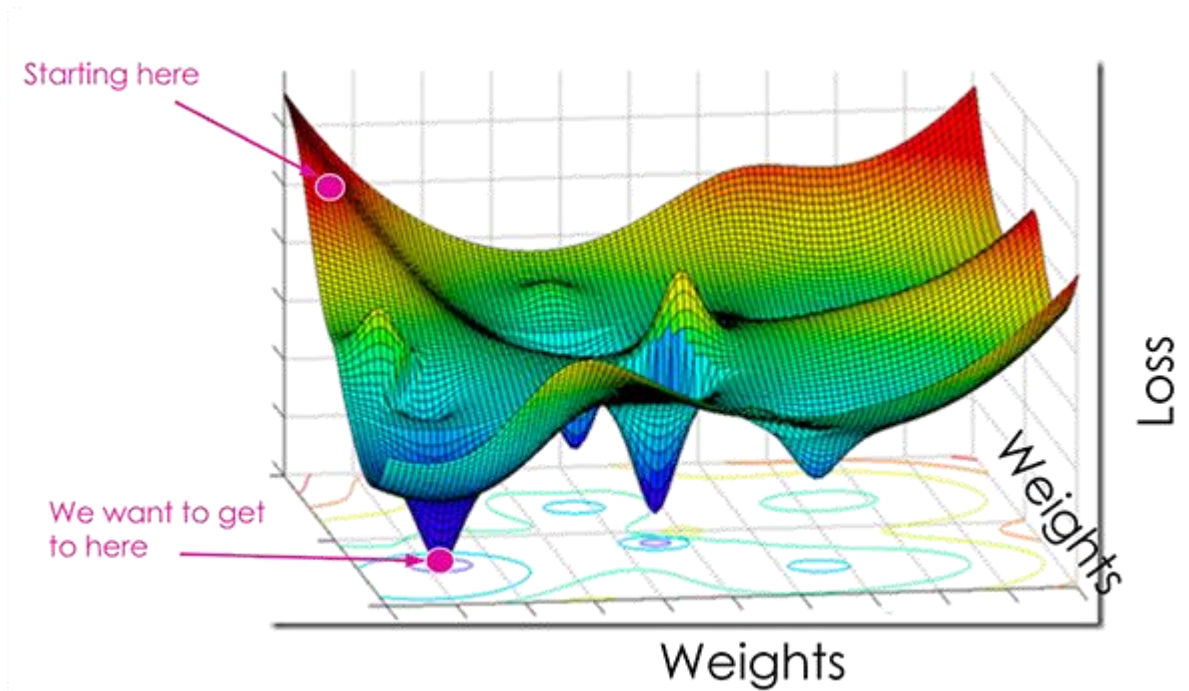
$$\theta_{best} = \text{Argmin}_{\theta} (f(Y_{learning}, \widehat{Y}_{learning}(\theta)))$$

Examples of loss functions:

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

$$CEL = \frac{1}{n} \cdot \sum_{i,j} -y_{ij} \log(\widehat{y}_{ij}(\theta)) + (1 - y_{ij}) \log(1 - \widehat{y}_{ij}(\theta))$$

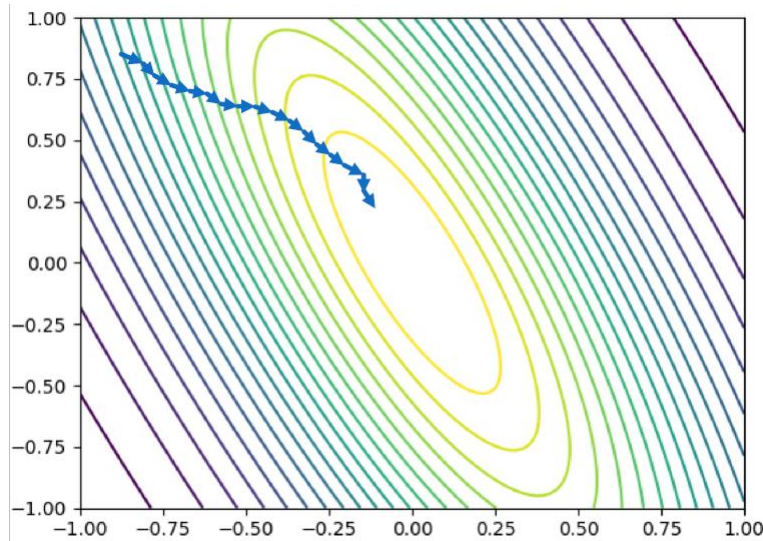
To find the best configuration of the network, the goal is to find the minimum of the loss function in the domain space of all parameters ($\theta = (W, b)$).
Since no mathematical expressions are available, numerical approaches are mandatory (gradient descent algorithms).



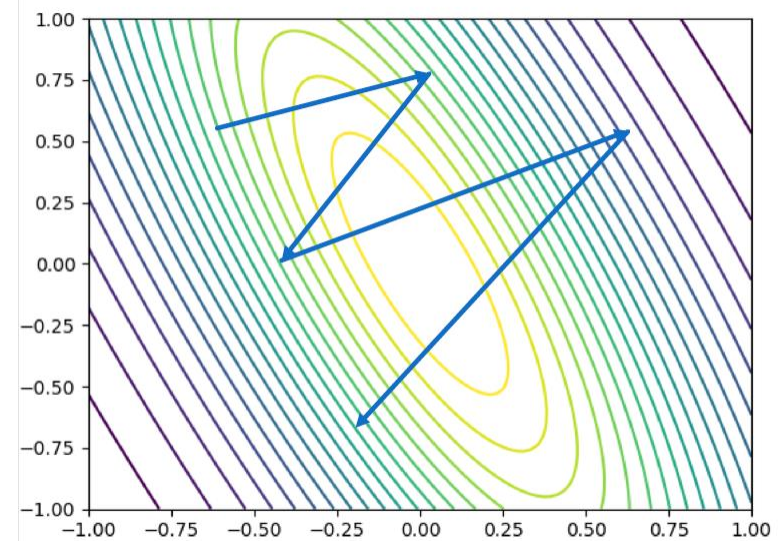
The goal of the learning phase is to minimize the loss function (due to all weights and biases of the network) by using a gradient approach:

$$\theta = \theta - \lambda \cdot \Delta L(\theta)$$

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



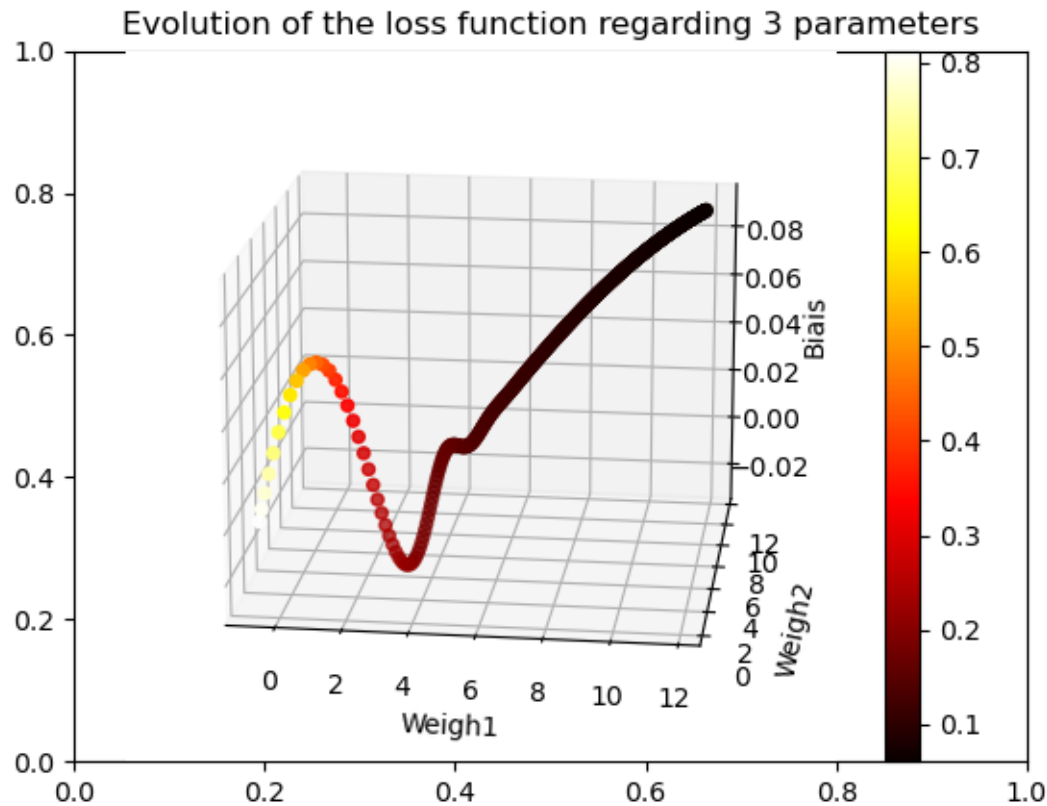
λ too low : long time to converge
and be trapped in local optimum



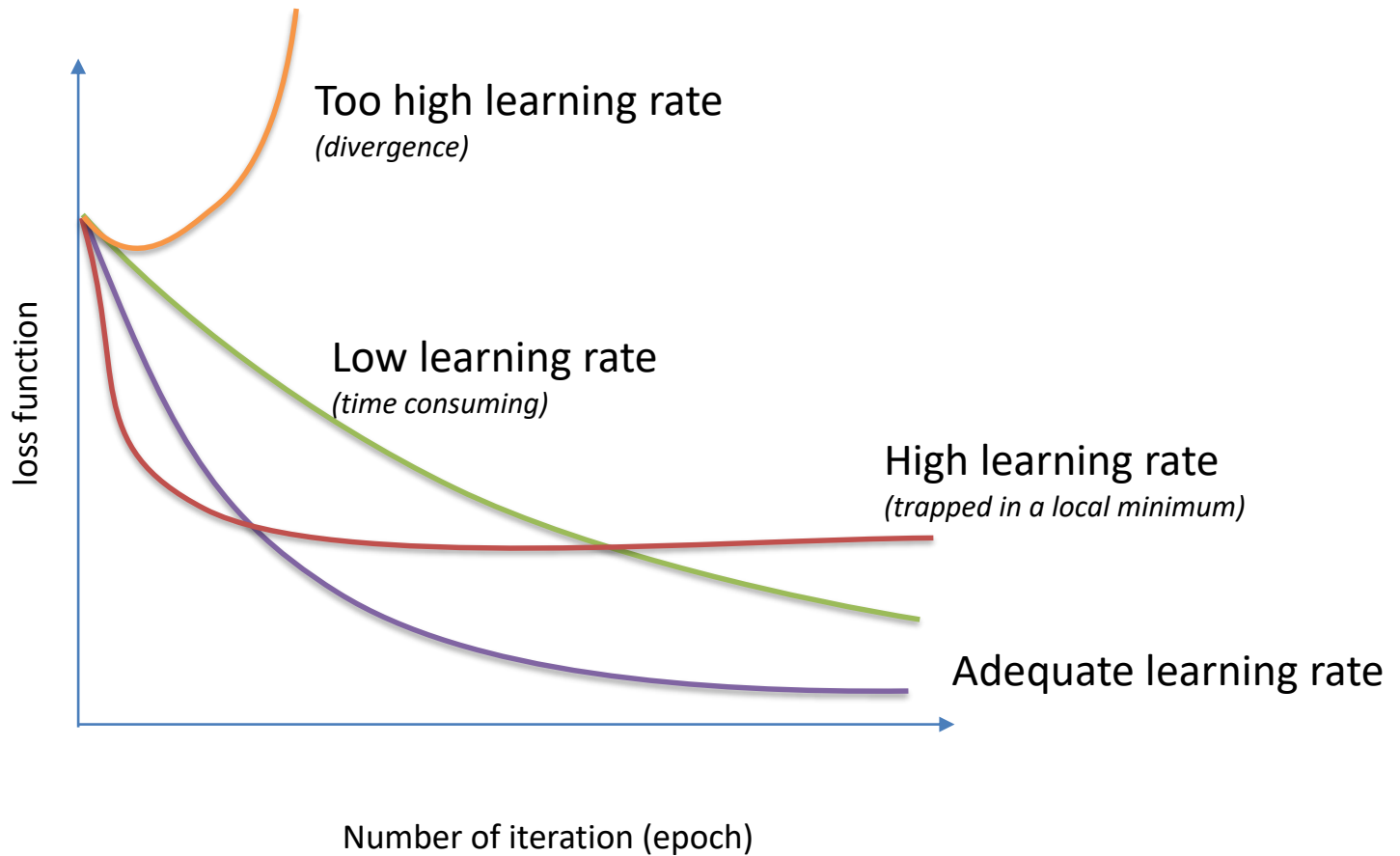
λ too big : chaotic exploration with
big risk of divergence!

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

In the graph below, the evolution of the quantification of the characteristics of a very simple network (2 weights and one bias) is proposed. The colour models the loss function (the darker is the better).



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



Three main strategies are used to calculate the loss gradient $\Delta L(\theta)$ and so to train/correct 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 k^{th} weight in the last layer, with the i^{th} case, the gradient is defined:

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

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

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:

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

Then the new weight is calculated:

$$w_k^{[n]} = w_k^{[n]} - \lambda \cdot (\widehat{y}_{ik} - y_{ik}) \cdot (\widehat{y}_{ik} \cdot (1 - \widehat{y}_{ik})) \cdot a_k^{[n-1]}$$

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:

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

With $a_j^{[0]}$ is the x_{ij} (the j^{th} element of the i^{th} case)

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

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

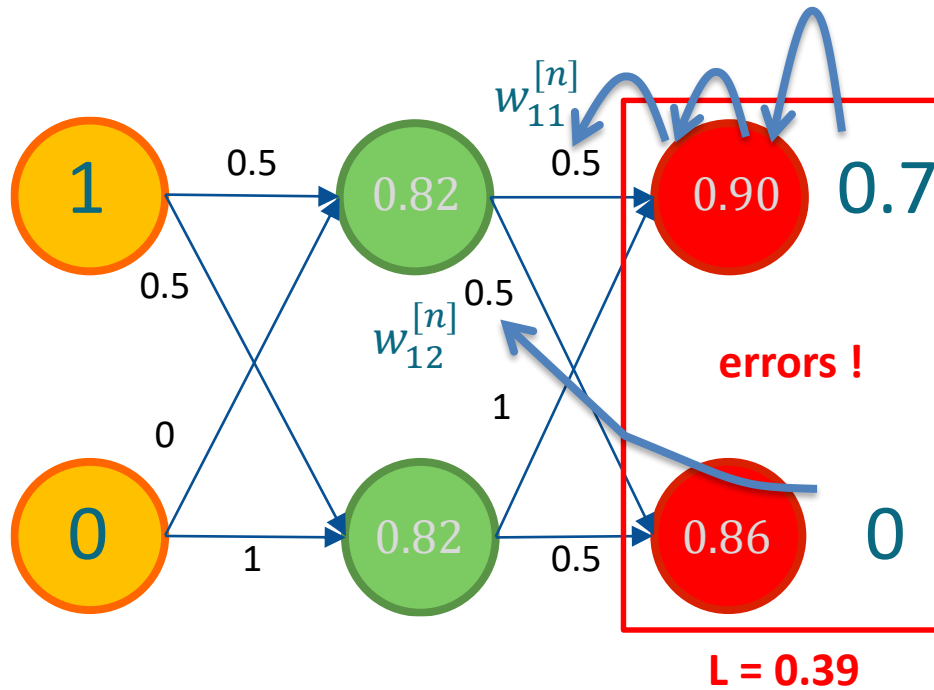
$$e_j^{[n]} = (\widehat{y}_{ij} - y_{ij}) \cdot g'(a_j^{[n]})$$

Back propagation - Example

Considering the total error L is calculated with:

Learning rate $\lambda = 0.1$

$$L = \frac{1}{2} \cdot \sum_{i,j} (\widehat{y_{ij}}(\theta) - y_{ij})^2$$



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

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

$$\text{New_}w_{11}^{[n]} = \mathbf{0.499}$$

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

$$\text{New_}w_{12}^{[n]} = \mathbf{0.491}$$

Back Propagation

Back propagation - Example

Considering the total error L is calculated with:

Learning rate $\lambda = 0.1$

$$L = \frac{1}{2} \cdot \sum_{i,j} (\widehat{y_{ij}(\theta)} - y_{ij})^2$$

$$\text{New_}w_{11}^{[n-1]} = w_{11}^{[n-1]} - \lambda \cdot e_0^{[1]} \cdot a_0^{[0]}$$

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

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

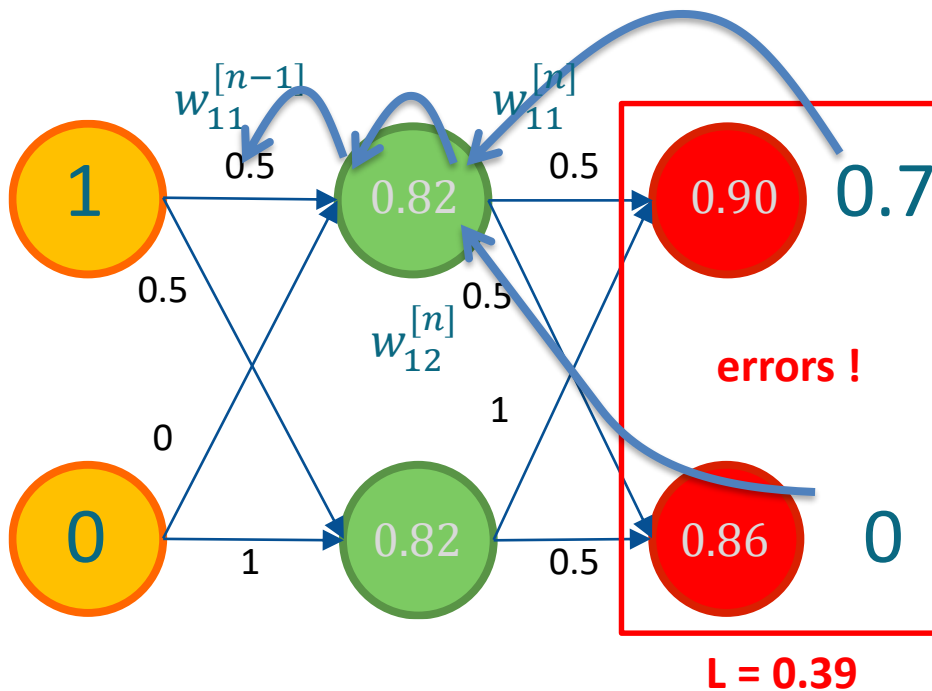
Or $e_k^{[l]}$ were already calculated on the previous step:

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

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

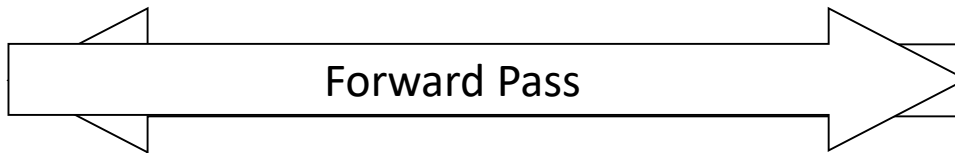
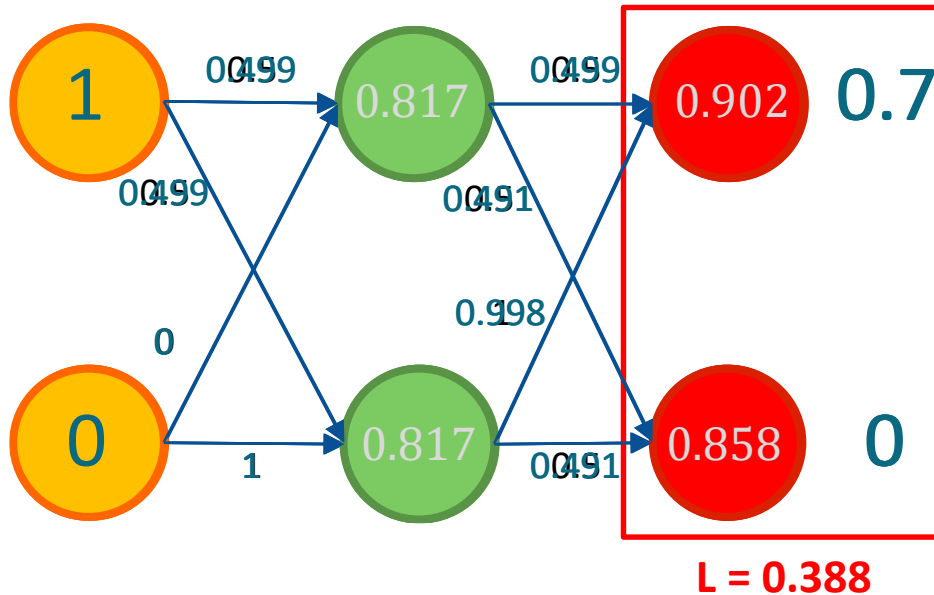
$$e_0^{[1]} = 0.148 \cdot (0.018 * 0.5 + 0.103 * 0.5) = 0.009$$

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



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...

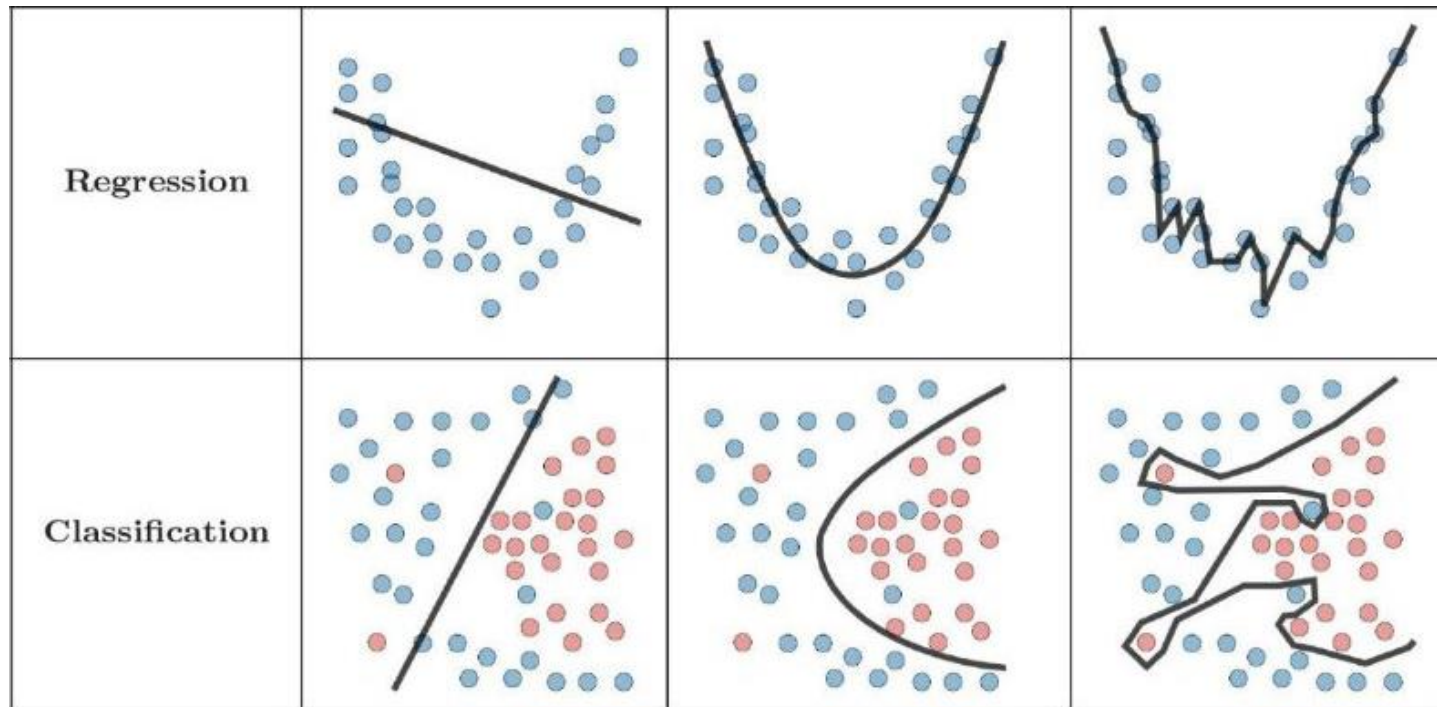
Evaluate the new weights and biases through back propagation

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

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

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).

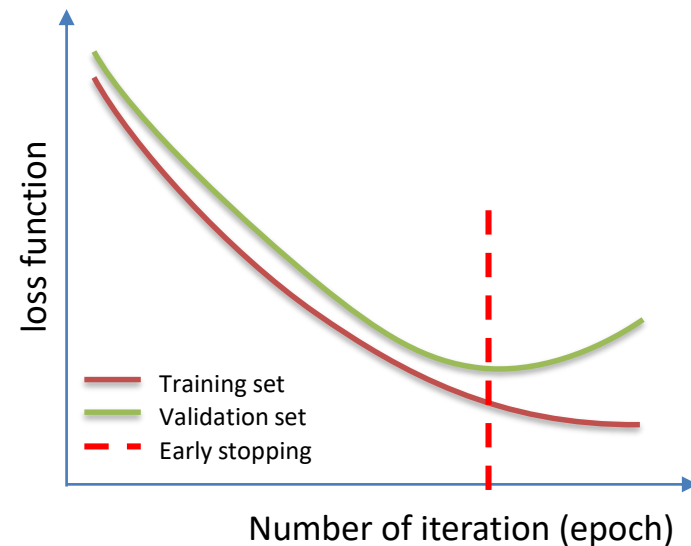


Underfitting

Overfitting

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.

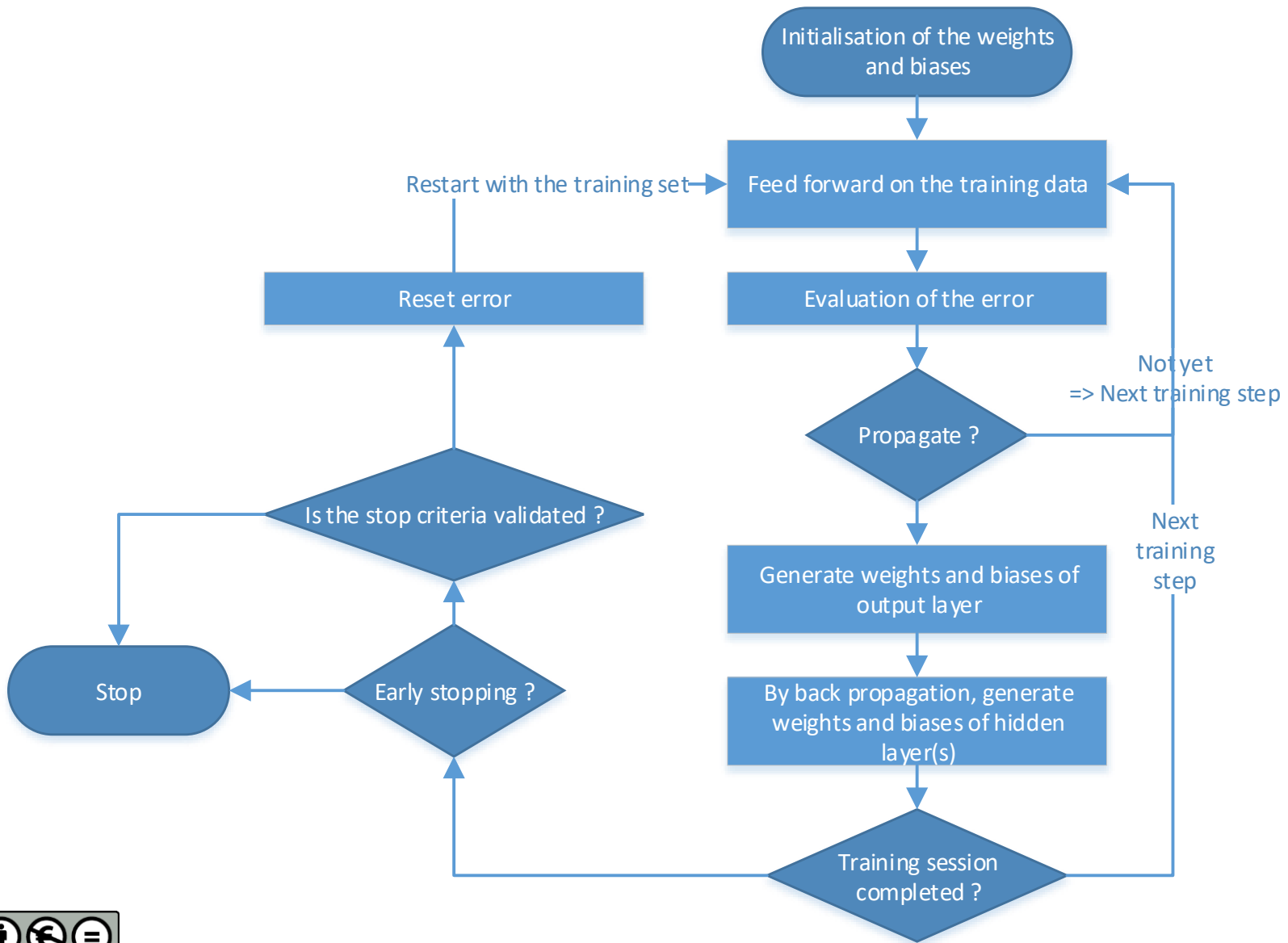
Then, how to split the dataset ?

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



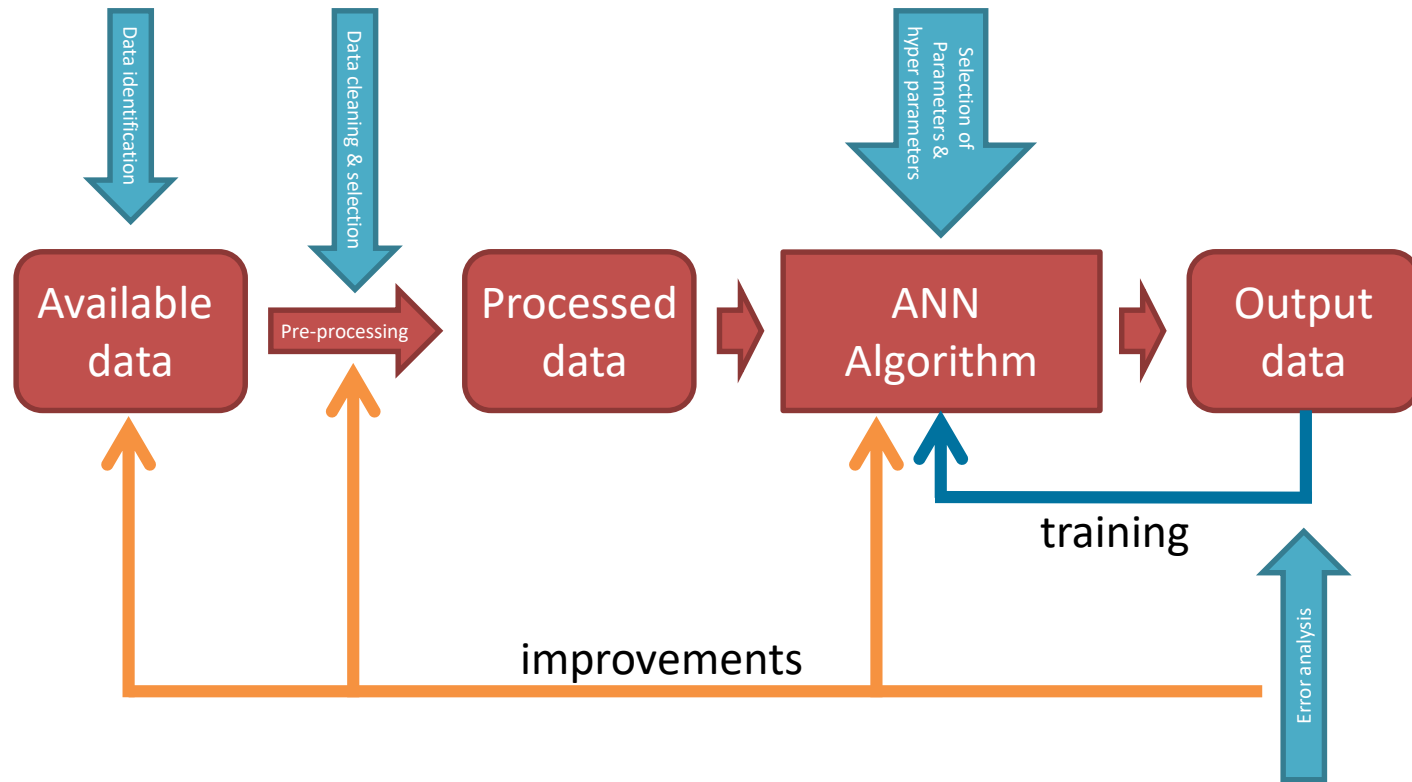
Where:

- 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: **80% / 15% / 5%**



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:



2. Designing a neural network

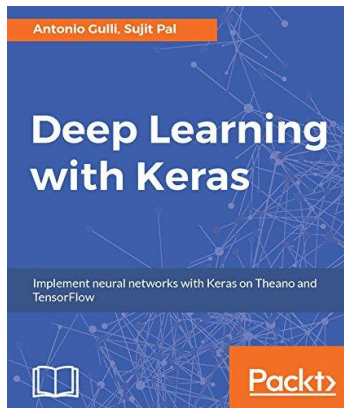
Using python with Keras packages



Several Python packages are available to design ANN and train them. Among them, [sci-kit learn](#), [pytorch](#) and [Tensorflow/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 at the [end of this course](#)
- 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. Several packs are proposed at discount price at www.humblebundle.com

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

Import all the mandatory packages

Load and prepare 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 and analyse the results and the ANN final configuration and save the configuration

Use the ANN final configuration to new cases

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

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.utils import plot_model
```

Package to define a MLP

Package to structure a MLP

Package to plot and represent the network


Package to define the minimisation parameters of the back-propagation

As soon as the MLP is defined (Sequential 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...)

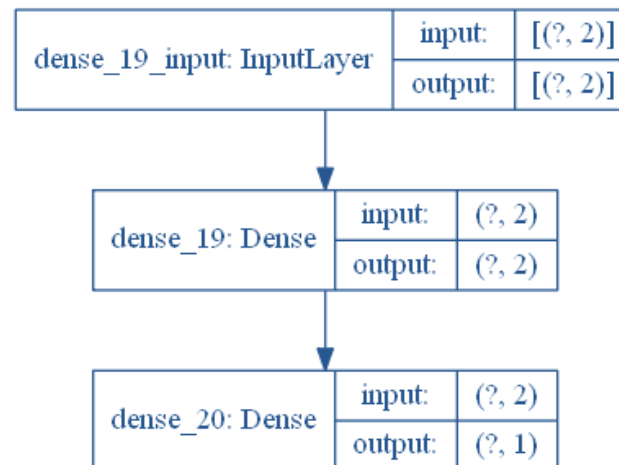
Several methods are available to access to the structure of the network:

- `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)
```



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'])
```

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,  
validation_split = 0.15)
```

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.

```
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
```

Current epoch (iteration of the training phase)

Evolution of the batch processing

Evaluation of the loss function on the epoch

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, validation_split = 0.15)
```

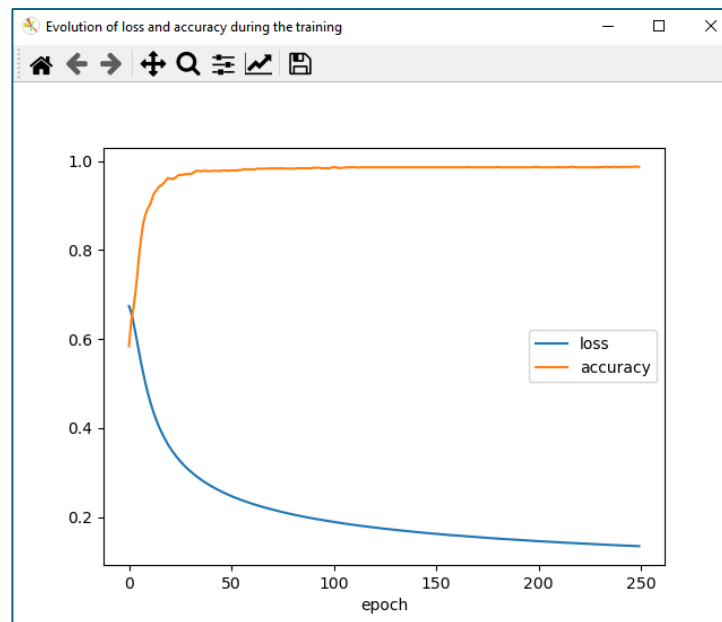
To access this information, store into a variable the result of the fit function. From this variable you can have access to this history. Before requesting any data, you can firstly check the ones available by using the keys() parameter of the history.

For example:

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

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')
plt.show()
```



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)
```

Or you can use `evaluate`, or a simple call of the network if the goal is just to work on only one evaluation (not a batch)

In classification problems, it is possible to calculate the confusion matrix that synthesises 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

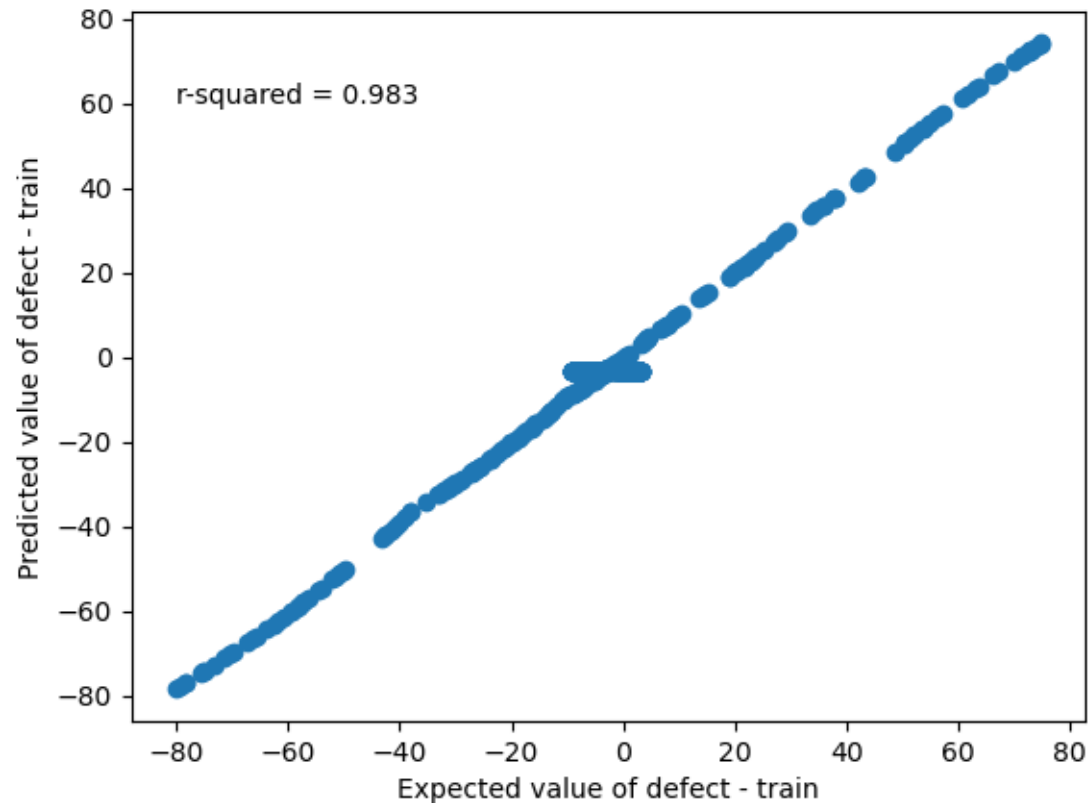
		Prediction	
		True	False
True label	True		α
	False	β	

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, predictions)
print(c)
```

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

In regression problems, it is possible to assess the relevance of the regression by using r^2 indices on the data composed by the prediction of the network and the expected value.



To plot and calculate this metric, you can adapt these programming lines, regarding your dataset and the prediction of your network:

```
from sklearn.metrics import r2_score

plt.figure("Comparison between predicted and expected value - calculated on
train_dataset")
plt.scatter(outputs_train, mon_reseau.predict(inputs_train))
plt.xlabel("Expected value of defect - train")
plt.ylabel("Predicted value of defect - train")
plt.annotate("r-squared = {:.3f}".format(r2_score(outputs_train,
mon_reseau.predict(inputs_train))), (-80, 60))
plt.show()
```

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:

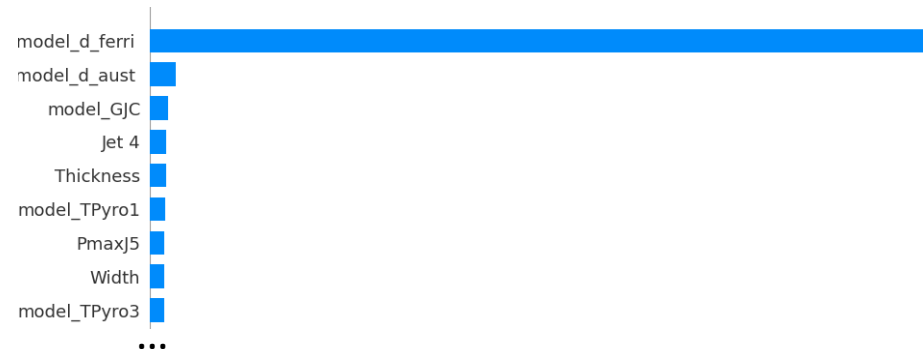
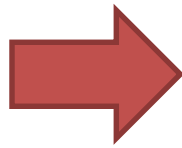
```
In [78]: ANN.weights
Out[78]:
[<tf.Variable 'dense_21/kernel:0' shape=(2, 2) dtype=float32, numpy=
array([[1.6364022, 7.908386],
       [0.50241685, 7.989181]], dtype=float32)>,
<tf.Variable 'dense_21/bias:0' shape=(2,) dtype=float32, numpy=array([-1.0894496, -8.029093 ], dtype=float32)>,
<tf.Variable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

Code example:

```
import shap as sh
```

```
explainer = sh.DeepExplainer(network, shap_data)  
shap_values = explainer.shap_values(other_input_data)
```

```
sh.initjs()  
sh.summary_plot(shap_values, headers)
```

List of the labels (column names) of the tested parameters (input parameters)

Neural network developed in tensorflow/keras packages

Set of input data (about 50/100 points) in numpy format

Set of input data (about 200 points) in numpy format different from the previous set of points

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()
```

...

Videos on YouTube:

- 3blue1brown propose an interesting and 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 an example: [Link](#)
- Several datasets proposed: [Link](#)

3. Examples and exercices

Using python with Keras packages



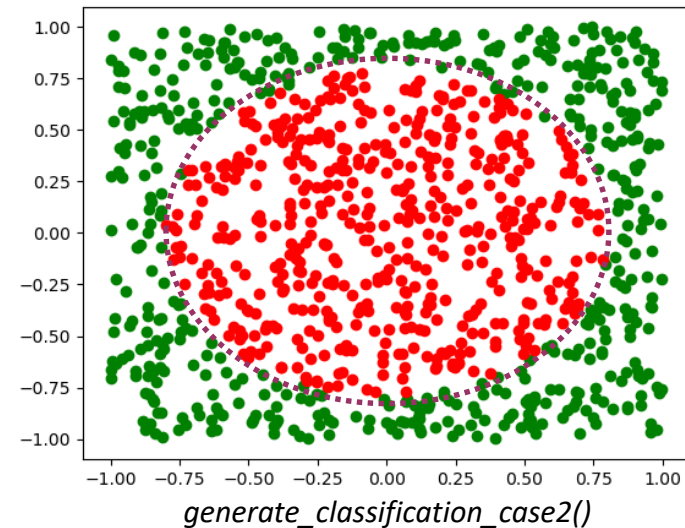
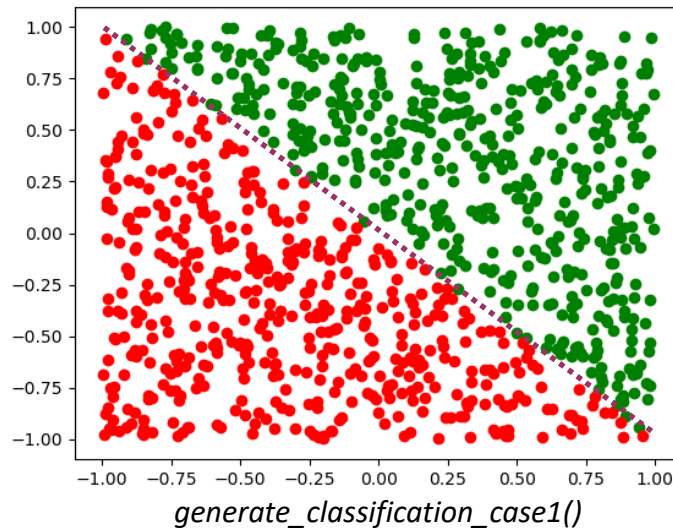


generate_classification_case
_case

Exercice – Classification

By using keras + numpy packages, three examples are proposed:

- Designing a simple perceptron where the separator is linear
- Designing a perceptron where the separator is not linear
- Improving the previous network by increasing its degree of freedom by adding one new hidden layer



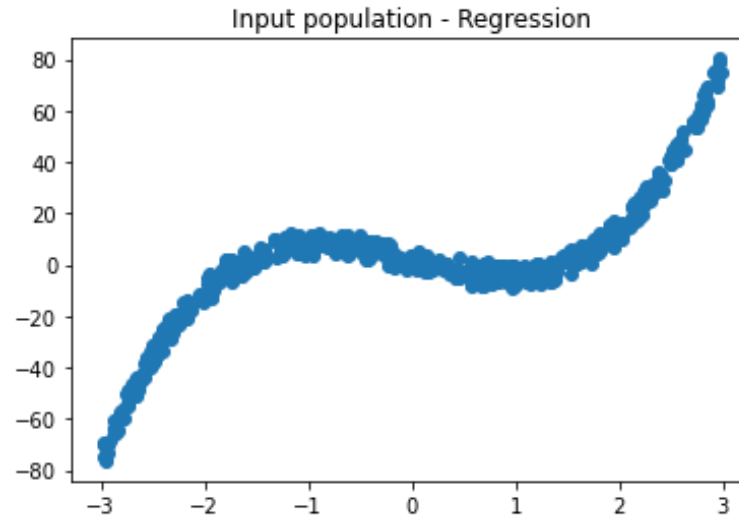
For these 3 examples, the structure of the designed network, its loss evolution and prediction analysis are proposed.



`generate_regression_case`

Exercice – Regression (with noise or not)

By using keras + numpy packages, design a regression neural network. To do so, the python function named `generate_regression_case` gives you a way to generate randomly and with a controlled noise the point set to work on:



After completing the definition and training of the network on a case without any noise, analyse the effect of the noise on the efficiency of the network.

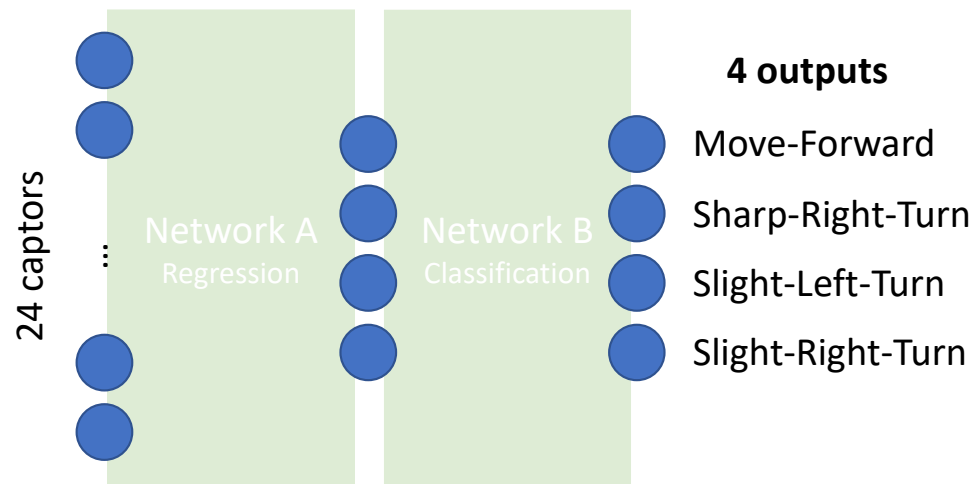
Problem 1: Regression + Classification



`load_dataset_networkB`

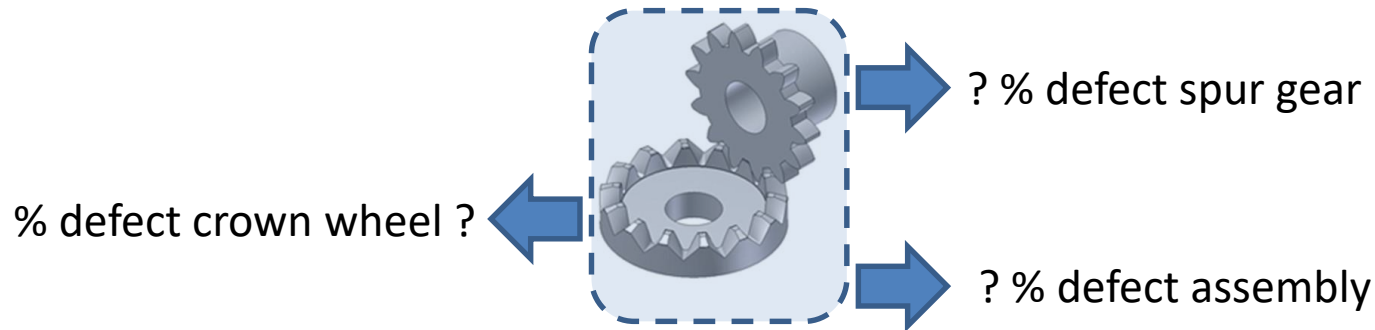
From the data given as two csv files, design, train and analyse the performances and results of two neural network types:

- A regression one (model reduction) – Network A
- A classification one – Network B



The code to load the dataset for the Network B is given as an example. Don't hesitate to copy/paste/modify or improve it for loading the datasets.

Try to evaluate the efficiency of the complete ANN process and compare it to a complete one.



The provided dataset, is the result of many gears simulations based on complex gears models. The time needed to perform these simulations is too huge to be directly integrated into an optimization algorithm. That is why an ANN is considered.

- It contains **15 inputs parameters** (colored in light blue in the Excel file), some are tolerances defined by designers and applied on the gears characteristics, and some are deviations on key parameters of these gears mainly due to their manufacturing. The last input parameter is the Kinematic Transmission Error (KTE).
- **Three outputs** are available (colored in gold in the provided file): the defect of each type of produced gears and the most important output the amount of final assembled defects (the most expensive ones).

The main goals of this project are:

- Firstly to check what are the parameters involved in the spur gear and crown wheel defects
- Secondly, to design a reasonable complex ANN accurately modelling the assembly defects.

4. Keras installation

Simple process to install Keras and other relevant packages



The first step is to download Anaconda package that includes a lot of packages interesting for engineers and scientists. In addition, it comes with several programming environment such as Spyder or Jupyter. To download this package, go to its [website](#) and select the version fitting your operating system. **Don't download nor install older version!**



As soon as the package is installed, find and run the “**Anaconda prompt**” to open the conda terminal. Notice that this will open on the base Anaconda environment.



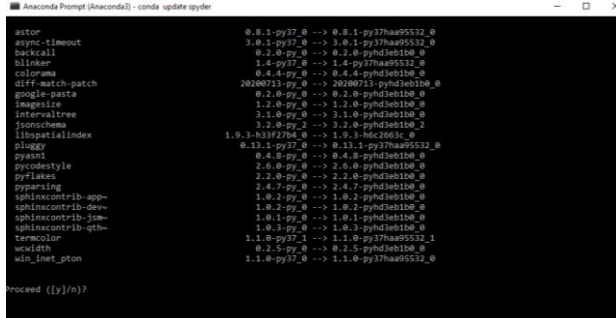
The next step is to create a dedicated environment for ANN to avoid any conflict with the current installation of your python environment (called (base)) or avoid breaking your python installation on your computer.

To create an environment and to activate it, please type these two programming commands:

- ✓ `conda create -n ann_env`, where *ann_env* is the name of the environment you want to create.
- ✓ `activate ann_env`, where *ann_env* is the name of the environment you want to activate.

At this step, update anaconda environment :

- ✓ `conda update -n base -c defaults conda`



```
anaconda Prompt (anaconda) - conda update spyder
astor 0.8.1-py37_0 --> 0.8.1-py37haa95532_0
async_timeout 1.0.1-py37_0 --> 3.0.1-py37haa95532_0
backcall 0.2.0-py_0 --> 0.2.0-pyh43eb1b0_0
blinker 1.4-py37_0 --> 1.4-py37haa95532_0
colorama 0.4.4-py_0 --> 0.4.4-pyh43eb1b0_0
diff_match_patch 20200715-py_0 --> 20200715-pyh43eb1b0_0
google-pasta 0.2.0-py_0 --> 0.2.0-pyh43eb1b0_0
imgviz 1.2.0-py_0 --> 1.2.0-pyh43eb1b0_0
intervaltree 3.1.0-py_0 --> 3.1.0-pyh43eb1b0_0
jsonschema 3.2.0-py_2 --> 3.2.0-pyh43eb1b0_2
libspatialindex 1.0.3-h33f278c_0 --> 1.0.3-h6c2682c_0
pluggy 0.13.1-py37_0 --> 0.13.1-py37haa95532_0
pyasn1 0.4.8-py_0 --> 0.4.8-pyh43eb1b0_0
pycodestyle 2.6.0-py_0 --> 2.6.0-pyh43eb1b0_0
pyflakes 2.2.0-py_0 --> 2.2.0-pyh43eb1b0_0
pyrsistent 2.4.7-py_0 --> 2.4.7-pyh43eb1b0_0
1.0.2-py_0 --> 1.0.2-pyh43eb1b0_0
1.0.2-py_0 --> 1.0.2-pyh43eb1b0_0
1.0.1-py_0 --> 1.0.1-pyh43eb1b0_0
1.0.3-py_0 --> 1.0.3-pyh43eb1b0_0
1.1.0-py37_1 --> 1.1.0-py37haa95532_1
0.2.5-py_0 --> 0.2.5-pyh43eb1b0_0
1.1.0-py37_0 --> 1.1.0-py37haa95532_0
Proceed ([y]/n)?
```

When the correct environment is activated (this is the one in parenthesis in the beginning of the prompt of the terminal), you can start installing the relevant packages by typing all these programming instructions one by one and by confirming the installation of the packages needed by pressing the “y” key:

- ✓ `conda install python=3.8`
- ✓ `conda install tensorflow`
- ✓ `conda install pydot`
- ✓ `conda install matplotlib`
- ✓ `conda install scikit-learn`
- ✓ `conda install -c conda-forge shap`
- ✓ `conda install spyder=5` (or Jupyter, pycharm... depending on your loved programming editor)
- ✓ `conda install -c anaconda pandas`