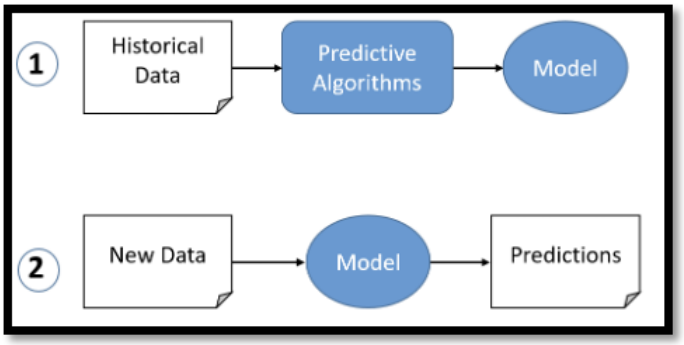
Lab 1 --- Introduction

1) STATISTICAL CONCEPTS FOR MACHINE LEARNING

- Definitions:

> **Artificial intelligence**: Techniques enabling computers to replicate human intelligence reasonning and decision taking methodologies.
>
> **Machine learning**: Subset of AI relying on statistical tools to enable computers to gain the ability to learn from data, and then make predictions and decisions.
>
> **Predictive modeling:** The AI paradigm can be seen as a problem of developing a model using historical data to make a prediction on new data where we do not have the answer. Machine learning models will often aim to learn a hypothetical function Y=f(X) via specific parameters or distributions in order to make predictions of Y for new X. In fact, we are not really interested in the shape and form of the function (f) that we are learning, only that it makes accurate predictions.
>
> 
>

- Empirical modeling:

> It is important to understand that all machine learning models can be classified as empirical models, also called statistical model or data-driven models. Indeed, in the field of science & engineering, those are also called phenomenological models, as they are constructed based on experimental data only without using a priori information about a phenomena or system S.
>
> > In simple terms:
> >
> > - Data are observations of real-world phenomenas (or systems) at different time steps.
> > - From those data, we can extract important features characterizing the system dynamics (or behaviours).
> > - From those features, we can often find important relations between them and assign them as variables in a model.
> > - Modeling those relations allows us to make classification decisions and predictions on future behaviours.
> >
> > To remember:
> >
> > - A “model” in machine learning is the output of a ML algorithm run on data. In other words, the model represents what was learned by the algorithm, or simply the “file” saved after running the algorithm on the training data. For a computer, it represents the instructions needed to make the prediction. Hence, a model is the specific representation learned from data and the algorithm is the process for learning it.
> >
> > Algorithms: **Parametric** vs **Non parametric**
> >
> > - Algorithms associated with **linear models** that simplify the function to a known form by summarizing the data with a set of **fixed size parameters** that is **independant of the number of training examples**. Feeding throught more data is just susceptible to change the coefficients in the equation but will not add complexity to a model. Hence, by making assumptions, the learning process is greatly simplified but is also limited to what can be learned...

- Algorithms associated with **non-linear models** that do not make strong assumptions about the form of the mapping function, allowing them to be more free to learn any functional form from the training data. As the **number of parameters grows with the number of training examples**, those methods are prefered when there is a lot of data and no prior knowledge.

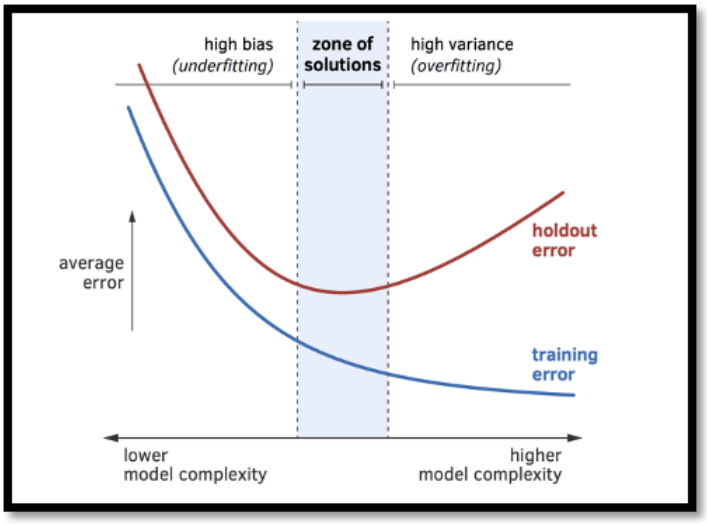
- Variance-Biais trade-off:

An important concept when using empirical models is the Variance-Biais trade-off. Indeed, this can be resumed by the fact that when trying too long to build a model that performs accurately on a specific training dataset, we often end up with a model that performs poorly on the other data like the test and validation sets. In other words, by trying to reduce the variance of our model, we increase its bias, and vice versa. The most important factors in that is of course the complexity of the model, as more complex models that are trained too long can learn to memorize all training examples and their labels, hence mapping almost perfectly the target function.

In the field of statistics, a fit is seen as how well a target function is approximated. The problem of overfitting or underfitting the data is often seen in supervised learning methods where the training is done to approximate the unknown underlying mapping function for the output variables given the input variables. And when it comes to parametric models, they are often said to have smaller variance and larger bias. In the other case, non-parametric models possess smaller biases but larger variance.

- Underfitting: high bias (model performance is average on training data and testing data)
- Overfitting: high variance (model perform almost perfectly on training set but poorly on other data)

In order to control this trade-off, we can tweak what are called the hyperparameters of a model, which can be seen as model parameter that are controlled manually in order to manipulate the learning process of the data. If we desire to reduce the biais of our model, we need to increase the model complexity for better data fitting. This can be done by switching to higher-order polynomials, increase the depth of a decision tree, or increase the number of nodes and/or layers in case of a neural network. Now in the other hand, if we desire to reduce the variance of our model as it is overfitted, we need to simplify the model by using regularizing techniques. (see next section)

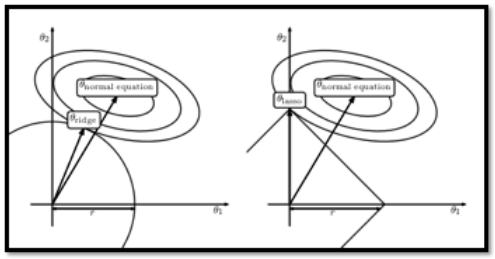


- Regularization:

In mathematical modeling, regularization is a technique where a regularization term (or penalty) is applied to an objective function of a model in order to force a learning algorithm to train a model with a lower degree of complexity, hence reducing its variance. Two common regularization techniques are L1 and L2 regularization, which both modify the objective function in order to penalize complex models and avoid overfitting. As those methods are known to reduce the magnitude of the weighted parameter values in a model, they are also called weight decay methods.

- L1 regularized objective: Penalizes the sum of absolute value of weights.
- L2 regularized objective: Penalizes the sum of square weights.

In addition to being very common when it comes to tuning linear models, L1 and L2 are also used in other type of model architecture like neural networks which may combine those with other type of regularization techniques like layer dropout and batch-normalization.



---

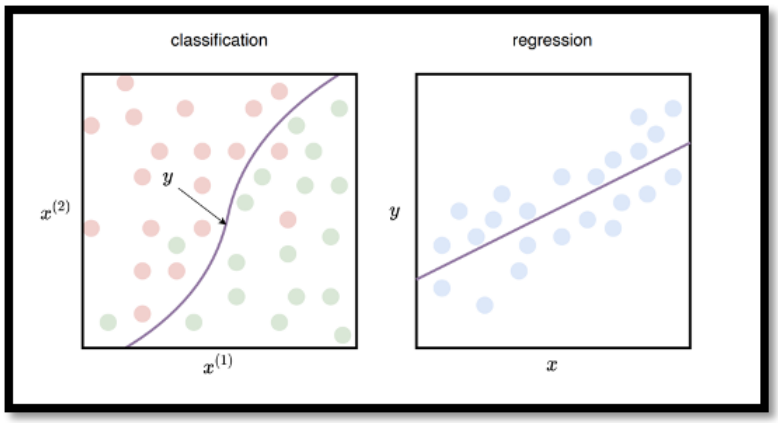
2) TYPES OF MACHINE LEARNING

- Supervised learning:

Type of learning that deals with **labeled data** that allow algorithms to predict the output variable (Y) from the input data (X). Accurate predictions are made by successfull approximation of the mapping function throught a training process where the model make predictions and is corrected when those are wrong. Hence, the learning phase continues until the model achieves a desired level of accuracy on the training data. In simple words, the goal is to produce a model that allows us to **deduce a label for the input feature**.

- Classification: Problem when the output variable is a category, such as “yellow” or "green, “positive” or “negative”. Therefore, the taks consist mainly of predicting a **discret label** but in some cases, a continuous value may be predicted in the form of a probability for a class label.

- Regression: Problem when the output variable is a real value, such as “dollars". In this case, the task consist mainly with predicting a **continuous quantity**.

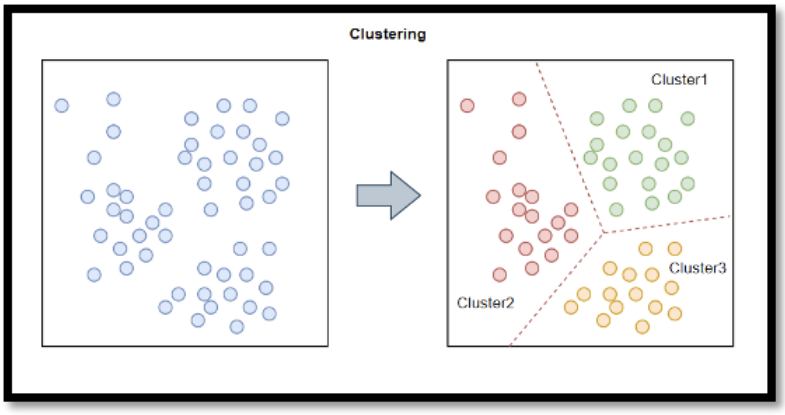


- Unsupervised learning:

Type of learning where the **data is unlabeled** and the algorithm task is to **learn the inherent structure** of a datset. Because there is no corresponding output variables to an input data (X), a model may try to extract general rules or attempts to find natural clustering groups among the data.

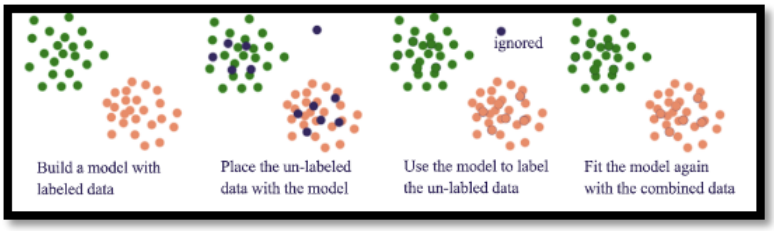
- Cluster analysis: Find the intrinsic groupings in the data such as grouping similar objects in a large collection of articles or grouping customers by purchasing behaviors.

- Association: Discovering rules that describe a large portions of a dataset.



- Semi-Supervised learning:

Problems where there is a large amount of input data (X) and **only some of them are labeled (Y)**. This category is often seen in real life problems as labeling large amount of data is time & ressource consuming. Consequently, using the available labeled training data to build a model that will convert the non-labeled data to labeled data represent often the best solution.

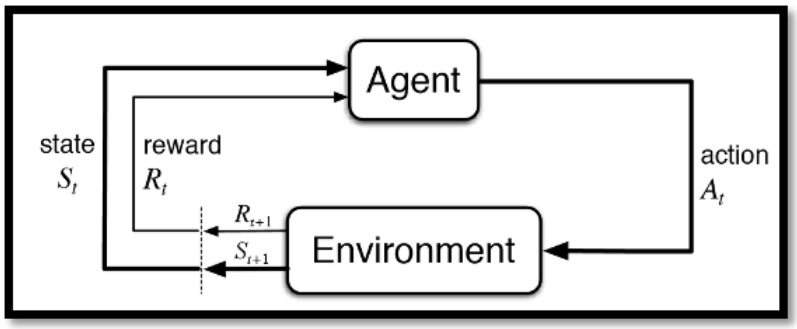


- Reinforcement learning:

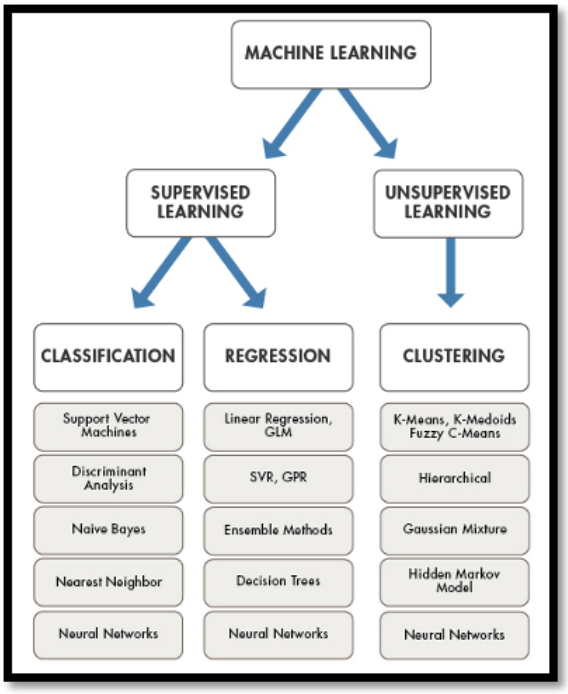
Techniques that enables an agent living in a pre-defined environnement to learn by trial and error using feedback loops from its own actions and experiences. Beacuse different actions lead to different reward situations, the main goal of the algorithm is to find the optimal policy that will maximise its overall long-term reward.

Most reinforcement learning problems can be viewed as **Markov decision processes**, where the agent is faced with a set of finite environment states S, a set of possible actions A(s) in each state, a real valued reward function R(s) and a transition model P(s', s | a).

- Policy: Function that takes the feature vector of a state as input and outputs an optimal action to execute.
- Environment: World in which the agent operates
- State: Current situation of the agent
- Reward: Feedback from the environment



- EXAMPLES:



---

3) ARCHITECTING A GOOD MACHINE LEARNING MODEL

- Step 1: Defining the problem

The first step consist mainly of **establishing the problem / question to solve** and its implications. This may include planning the steps related to the **kind of data** you need or which **type of models** are the most adapted to the task.

- Step 2: Data collection

The second step involve **finding the data** you need, often from a single or multiples sources (internet, databases...), and transfering it to your own data storage space where it will be easy for you to access it later. This is often done throught the intermediary of databases (SQL or NoSQL).

- Step 3: Data preparation

The third step incorporate any action related to transforming the raw data that was collected into a format that can be used in modeling and accurate enough for obtaining a good performing model. As a guideline, here are the most common **propreties of good dataset**:

- Contains enough information that can be used for modeling,
- Has good coverage of what you want to do with the model,
- Reflects real inputs that the model will see in production,
- Is as unbiased as possible,
- Is not a result of the model itself,
- Has consistent labels, and
- Is big enough to allow generalization.

Without going to much in the details, tasks in this section may include:

- **Data Cleaning**: Identifying and correcting mistakes or errors in the data.
- **Feature Selection**: Identifying those input variables that are most relevant to the task.
- **Data Transforms**: Changing the scale or distribution of variables.
- **Feature Engineering**: Deriving new variables from available data.
- **Dimensionality Reduction**: Creating compact projections of the data.

- **Data segragation**: Split subsets of data for the training, testing, and validation phase.

A feature can be defined as **individual measurable property or characteristic of a phenomenon** and often classified as informative, discriminating or independent. The initial set of raw features extracted from a dataset can be redundant and too large to be managed. Therefore, a preliminary step in many applications of machine learning consists of selecting a subset of features: **this is called dimensionnality reduction**.

Feature selection can be defined as the process of **reducing the number of input variables when developing a predictive model**. It is desirable to reduce the number of input variables to both reduce the computational cost of modeling and, in many cases, to improve the performance of the model. Statistical-based feature selection methods involve evaluating the relationship between each input variable and the target variable using statistics and **selecting those input variables that have the strongest relationship with the target variable**.

- Step 4: Model training and tuning

The fourth step consist of the **application of the machine learning algorithms** to the training subset of the data in order to **optimize the model internal parameters**. Additionnaly, this phase may involve training multiple models at the same time with different techniques in order to diversify the model choices later. This refers often to fine-tuning a model in order to improve its performances.

A popular set of methodologies for fine-tuning a model are called feature scaling techniques. Indeed, when using gradient based optimization, it is often necessary to scale the data for optimal convergence towards the minima of the objective function. Normalization is a simple feature scaling technique where numerical features are scaled in a range of 0 to 1. An other technique called standardization compute new values for the data by making them have zero mean and one unit of variance. Many other techniques can be implemented but when it comes to finding the best one, some prefer to fit the model to raw data, then to normalized and finally to standardized data in order to compare their performances.

Other tuning techniques used to get the best model possible involve using different regularization penalizers, trying multiple hyperparameters by varying the different optimizer used to train the model. But often, traditionnal optimizer like the Adam one for neural networks are used as they are often recognized to perform well in most cases.

- Step 5: Model Evaluation

The fifth step involve summarizing the performances of each trained model. This is mainly done by designing a **robust testing methodology** on specific subsets of the dataset in order to assess the level of predictive performance for each model. After this phase, the selected model is often ready to be deployed and used.

Model evaluation may involve sub-tasks such as:

- Select a **performance metric** for evaluating model predictive skill.
- Select a model evaluation procedure.
- Tune algorithm hyperparameters.
- Boost the accuracy by combining multiple models into ensembles.
- Using data visualisation tools to understand results

When assessing the performance of a model, multiples metrics are used depending on the context and the preferances. The most common one are often R2, MSE, RMSE, or the Confusion Matrix. Finally, using out of sampling techniques like Cross-Validation is often key when it comes to testing the model's ability to generalize by testing it on independent samples of data.

---

4) PYTHON AS MACHINE LEARNING TOOL

- Description:

- Python is a widely-used, interpreted, object-oriented, and high-level programming language that is becoming the go-to when it comes to learning the field of data sciences.

- Over the years, the growing number of users and open-source packages turned Python into the most used ecosystem available for data analysis and machine learning tools.
- Because this course is not about mastering Python from end to end, you will only be required to understand and use the basic methodologies related to engineering a machine learning model. Watching the related videos and examples will help you getting familiar with this important tool.

- Important Librairies:

Libraries can be seen as pre-written lines of code that can be imported in our program in order to provide us the ability to perform many tasks without writing our own code. Inside librairies, you can find multiple modules with different functionnalities, acting as a layer of abstraction in order to provide you direct access to reusable lines of code that you may want to include in your program. One of the main librairy we will use is called SciPy, which is a famous ecosystem containing multiple Python modules for mathematics, science and engineering. Here are some of the packages we will use for this course:

- NumPy: Allow us to work efficiently with data in arrays.
- Matplotlib: Allow us to create data visualisation tools.
- Pandas: Tools and data structures to load, organize, and analyze data.
- SciPy: Ecosystem of Python libraries for mathematics, science and engineering
- Scikit-learn: ML and deep learning models & training algorithms.
- Kera: ML and deep learning models & training algorithms.
- Tensorflow: ML and deep learning models & training algorithms.

- Required methodologies:

- How to import the right librairies & packages
- How to load Machine Learning datasets
- How to generate statistics related to data
- How to visualise data
- How to generate, train and evaluate machine learning models
- How to make predictions with the model

(For the purpose of this course, the phases related to data pre-processing and feature engineering will not be required. Additionnaly, many layers of abstraction are made on certain concepts in order to provide a fast track to the student for implementing the models directly on real world datasets.)

- Modeling template:

Across all laboratories, your task of modeling a dataset through machine learning techniques will often take the same strucuture. Of course, datasets, models and algorithms will always change, but as the complexity of the task will grow, the strucuture below will never change and you should understand it well:

#1. Prepare Problem: (Load libraries, Load dataset)

#2. Summarize Data: (Descriptive statistics, Data visualizations)

#3. Prepare Data: (Data Cleaning, Feature Selection, Data Transforms, Split validation & test set)

#4. Model initiation and training: (Import model, Set-up hyperparameters & optimizers)

#5. Evaluation & Accuracy: (Algorithm Tuning, Performance review)

#6. Visualisation and prediction: (Predictions on validation dataset, Plot data, Save model)

---------- LINKED VIDEO TO WATCH ----------

- Machine learning fundamentals:

1) https://www.youtube.com/watch?v=z-EtmaFJieY&list=PLIrbp3VWo-GIDEXNG9zZrW5reQPmPnXwk&index=5&ab_channel=CrashCourse

2) https://www.youtube.com/watch?v=Rt6beTKDtqY&list=PLIrbp3VWo-GIDEXNG9zZrW5reQPmPnXwk&index=4&ab_channel=ZachStar

- Python fundamentals:

1)

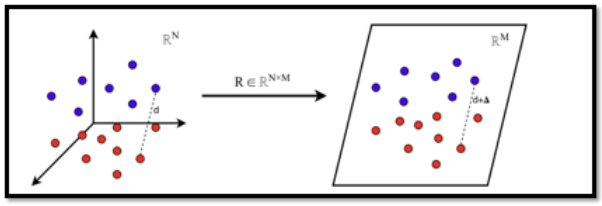
2)

Lab 2 --- Dimensionality reduction

- Introduction on subject:

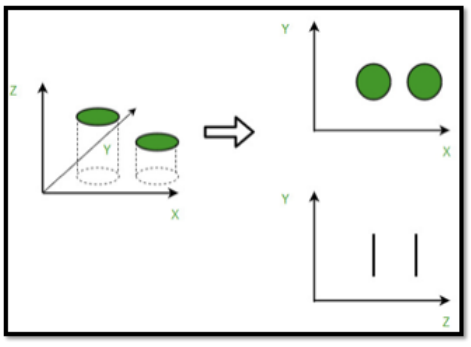
When starting in the field of Machine learning, one of the key concept to master before anything else is Dimensionality reduction. Indeed, before modeling and training a predictive model, it is often recommanded to apply this **data preparation technique** in order to keep only the most relevant variables from the dataset, which will improve the performance of the machine learning algorithms.

In fact, the dimensionnality of data can be seen as the **number of input variables** (also called features) in a dataset. As those variables grow in number, the task of predictive modeling become more challenging as high amount of training data is required in order to maintain an accurate representation of the data space. This phenomena was initially called by the founder of Dynamic Programming (Richard Bellman): the **curse of dimensionality**.



Therefore, dimensionality reduction techniques can simply be seen as ways to **reduce the number of input variables** in a dataset. The first approach to this problem is called **feature selection**, where we keep only the most relevant features from the dataset and discard the others. This is done by analyzing the properties of all existing features through statistical methods in order to find which variables are the most relevant to keep for explaining a specific phenomena, hence maximizing the quality of our model.

A second approach to this problem is using mathematical transformations like **Matrix Factorization** (called feature projection) to create a smaller set of new variables, which are a combination of the different input features of our dataset, while keeping almost all of the information necessary for our model. This is done by taking the original feature vector and transforming it into a new vector with lower dimensionality.



As the era of Big Data progresses, multiple other approaches of different complexity are being developped to reduce the dimensionnality of datasets. For example, **Autoencoders** are a type of Deep learning neural networks architecture that is seen as a very promising one when it comes to more complex datasets. Of course, we will not see all of them, but autoencoders are one of the architecture that we will review in our Deep learning laboratory.

In short, it is important to understand that dimensionality reduction techniques are **critical** when it comes to simplifying a classification & regression tasks, increasing the learning speed of algorithms, adding generalization capabilities to models & avoiding overfitting, and finally improving data visualization for humans.

- List of methods presented in this laboratory:

1. Principal Component Analysis (PCA)
2. Independant Component Analysis (ICA)
3. Linear Discrimant Analysis (LDA)

- Instructions:

Completed laboratory need to be sent to the teacher before midnight

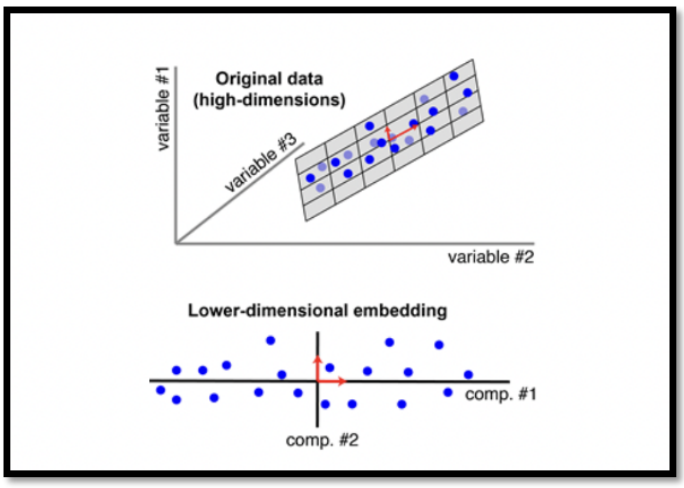
1) PRINCIPAL COMPONENT ANALYSIS (PCA)

- Model overview:

The principal component analysis (PCA) method is known to be one of the most used unsupervised algorithms to perform dimensionality reduction tasks. Indeed, by identifying and detecting the correlation between variables, it can reduces the dimension of a d-dimensional dataset by projecting it onto a (k) - dimensional subspace (where k < d). By rotating the axes, the PCA algorithm will try to showcase more variability from a dataset.

Using linear algebra methods like matrix decomposition (Eigendecomposition or SVD), PCA produce an orthogonal projection of the data onto a principal subspace, where the variance of the projected data is maximized for optimal retention of the essence of the original data. By reducing a dataset matrix into its main constituent parts, it is possible to rank them and select only those that best captures the salient structure of the matrix that can be used to represent the dataset. In simpe words, the PCA algorithm will try to find a list of the principal axes in the data for better description. The algorithm will output the main components found (can be seen as vectors axis) and their explained variance, which relate how the data will vary if projected onto that axis.

When applying it to dimensionnality reduction, PCA will simply remove one or more of the smallest principal components that do not explain well the data, hence resulting in a lower-dimensional projection that preserves a high degree of variance. In other words, by leaving only the components of the data with the highest variance, we remove the information along the least important axis, allowing us to reduce the dimension of the data while keeping a high degree of relationship between the observations.



One of the most important steps in the PCA algorithm is the Matrix factorization (or "decomposition") phase, where a given matrix is described using its constituent elements. In general, many methods like the Singular-Value Decomposition (SVD) or an eigendecomposition are used, which often generate the same result at the end. To perform the PCA, we use **eigenvectors and eigenvalues**, which are measures used to quantify the direction and the magnitude of the different variation captured by each axis. Logically, the correlation between each principal component (or any pair of eigenvalue/eigenvector) should be zero, producing **mutually orthogonal axes** that are perpendicular to each other.

Now when projecting data of datasets with dozens dimensions, the first principal axis will be the line which **maximizes the variance**, then each additional component added to the space expresses less and less variance than its predecessor (more noise), therefore it is important to choose the right number of components. This can be done by computing the cumulative explained variance ratio as a function of the number of components and using techniques like the elbow method to choose the right number.

Of course, using PCA required us to have a sufficient number of observations, a high ratio of observation to features, correlated features, linearity, and few to none significant outliers in order to produce good results. Nonetheless, the principal component analysis method is widely used when it comes to dimensionnality reduction, feature extraction, noise filtering, and data visualization.

- Algorithmic steps:

-Step 1: Pre-process the data by computing the mean-subtracted values of your observations, also called data normalization.

-Step 2: Compute the covariance matrix for Eigen decomposition, which will give us the eigenvalues and eigenvectors of the matrix.

-Step 3: Sort eigenvalues in descending order and choose the k eigenvector that correspond to the k largest eigenvector where k is the number of dimensions of the new feature subspace. The optimal number of principal components is often determined by looking at the cumulative explained variance ratio as a function of the number of components, which can be resumed as making a tradeoff between dimensionality reduction and the information loss.

-Step 4: Construct the projection matrix W from the selected k eigenvectors.

-Step 5: Transform the original dataset X via W to obtain a k-dimensional feature subspace Y.

- Example 1:

Below is an example of a **Principal Component Analysis** algorithm that was applied in a classification task in order to reduce the number of dimensions of a dataset containing observations of different wines with 14 features each. With PCA applied, we successfully reduced the 14 dimensions of this dataset to only 2 dimensions, hence we found the optimal stretch and rotation that allows us to see the layout of those wines in much lower dimensions. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

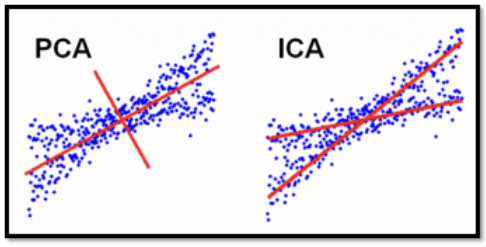
```
# Importing packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


# Importing dataset
dataset = pd.read_csv("Wine.csv")
x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values


# Data pre-processing
    # Splitting the dataset into the training & testing sets
    # Dividing training and test set (ratio 80 - 20 for trainging and testing respectively)
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)


    # Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()


    # Fitting and Transforming
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)



# Applying PCA
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
x_train = pca.fit_transform(x_train)
x_test = pca.transform(x_test)



# Training the Logistic Regression model on the Training Set
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(x_train, y_train)


# Performance review (Confusion Matrix)
from sklearn.metrics import confusion_matrix, accuracy_score
y_pred = classifier.predict(x_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
print("Accuracy Score of PCA is",accuracy_score(y_test, y_pred))



# Visualising the Training Set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
```

```
x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)
plt.title('Logistic Regression (Training set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()




# Visualising the Test Set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_test, y_test
x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()
```

---

2) INDEPENDANT COMPONENT ANALYSIS (ICA)

- Model overview:

> Now that we have seen the PCA algorithm, it is time to see its extension called Independant Component Analysis (ICA). Unlike PCA where the goal is to extract correlations by maximizing the variance, the ICA algorithm seek to maximize the indepedance by finding the optimal linear transformation of the feature space in order to produce new features that are mutually independent statistically. Thus, the different latent variables are assumed to be nongaussian and mutually independent, forming what is called the independent components of the data (or the sources).
>
> This situation is often well represented by the blind source separation (BSS) problem, also called the "cocktail party problem", where a source signal is extracted in the presence of noise even if little information is known about this signal. To extract the independant source signals, a specific unmixing matrix W need to be found. Estimating the unmixing matrix (W) can be done by many different algorithms, where the most popular methodologies are the maximum likelihood estimation, the minimization of mutual information or the maximisation of non-gaussianity.
>
> > 
> >
>
> Assuming the data is generated by the process $X = As$ where A denotes the mixing matrix and s denotes the original signals, it is possible to designate W as the inverse of mixing matrix A if we make the assumptions that A is a square matrix and s is generated by non-Gaussian process. Therefore, approximating W can be done using gradient descent in order to minimize the log likelihood.

- $W = W + \alpha[(1 - \frac{2}{1+e^{-W^T X}})X^T + (W^T)^{-1}]$

Many assumptions are taken into account when applying ICA. Those will not be detailed here as the goal is only to provide your with a framework to understand those algorithms and their importance when it comes to data analysis. Hence, the important thing to remember is that ICA is an algorithm that produce estimates of a source signals by analysing the observed signal mixtures through a linear transformation that maximizes the mutual independence (non-gaussianity) of the mixtures. To conclude, ICA is widely seen as a very powerfull noise removing & feature extraction tool that is constantly used in many fields like Biomedical analysis, Audio signal processing, or Image processing.

- Example 2:

Below is an example of a **Independant Component Analysis (ICA)** algorithm that was applied to 3 noisy signals in order to extract the independant components from them. Like you can see, the ICA output correspond to musical wave forms successfully extracted from the signals. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:
```
# Import packages
import numpy as np
import wave

# Read the wave file
mix_1_wave = wave.open('ICA mix 1.wav','r')


# Extract Raw Audio from Wav File
# List of ints representing the sound
signal_1_raw = mix_1_wave.readframes(-1)
signal_1 = np.fromstring(signal_1_raw, 'Int16')


# Visualisation
import matplotlib.pyplot as plt
fs = mix_1_wave.getframerate()
timing = np.linspace(0, len(signal_1)/fs, num=len(signal_1))

plt.figure(figsize=(12,2))
plt.title('Recording 1')
plt.plot(timing,signal_1, c="#3ABFE7")
plt.ylim(-35000, 35000)
plt.show()



# Add 2 others wave files for our dataset
# + Extracting Raw Audio from each Wav File
mix_2_wave = wave.open('ICA mix 2.wav','r')
signal_raw_2 = mix_2_wave.readframes(-1)
signal_2 = np.fromstring(signal_raw_2, 'Int16')

mix_3_wave = wave.open('ICA mix 3.wav','r')
signal_raw_3 = mix_3_wave.readframes(-1)
signal_3 = np.fromstring(signal_raw_3, 'Int16')



# Create dataset X by zipping signal_1, signal_2, and signal_3 into a single list
X = list(zip(signal_1, signal_2, signal_3))


# ICA algorithm to retrieve the original signals
from sklearn.decomposition import FastICA
ica = FastICA(n_components=3)
ica_result = ica.fit_transform(X)


# Splitting into separate signals
result_signal_1 = ica_result[:,0]
result_signal_2 = ica_result[:,1]
result_signal_3 = ica_result[:,2]


# Data visualisation
# Plot Independent Component #1
plt.figure(figsize=(12,2))
plt.title('Independent Component #1')
plt.plot(result_signal_1, c="#df8efd")
plt.ylim(-0.010, 0.010)
plt.show()
```

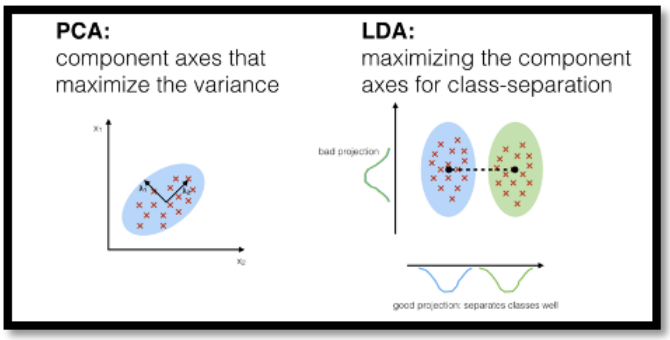
```
# Plot Independent Component #2
plt.figure(figsize=(12,2))
plt.title('Independent Component #2')
plt.plot(result_signal_2, c="#87de72")
plt.ylim(-0.010, 0.010)
plt.show()

# Plot Independent Component #3
plt.figure(figsize=(12,2))
plt.title('Independent Component #3')
plt.plot(result_signal_3, c="#f65e97")
plt.ylim(-0.010, 0.010)
plt.show()
```

---

3) LINEAR DISCRIMINANT ANALYSIS (LDA)

- Model overview:

> Finally, the last dimensionality reduction technique presented is LDA, a technique that differ slightly from PCA as it allows us also to find the optimal axes that maximize the separation between multiple classes. Indeed, this linear transformation is know to project the feature space into a smaller subspace k while maintaining the class-discriminatory information.
>
> In order to choose the value k, it is good to check the degree of similarity between the computed eigenvalues. If they seem all similar, this possibly mean that the actual feature space is optimal. And in case of eigenvalues much larger than others, this often means that keeping those eigenvectors is the optimal decision to take for dimensionality reduction.



- Here are the main steps for LDA:

> **1)** Compute the d-dimensional mean vectors for the different classes from the datasets.
>
> **2)** Compute the scatter matrices (in-between class and within-class scatter matrix)
>
> **3)** Computer the eigenvector and corresponding eigenvalues for the scatter matrices.
>
> **4)** Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues
>
> **5)** Use this (d k) eigenvector matrix to tranform the sample onto the new subspace.

- Example 3:

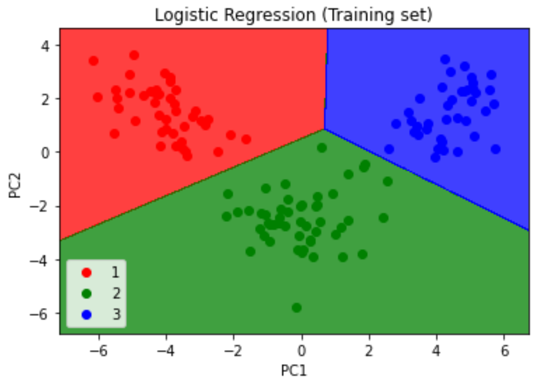
> Below is an example of a **LDA** algorithm that was applied in a classification task in order to reduce the number of dimensions of a dataset containing observations of different wines with 14 features each. With LDA applied, we successfully reduced the 14 dimensions of this dataset to only 2 dimensions, hence we found the optimal stretch and rotation that allows us to see the layout of those wines in much lower dimensions. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [1]:
```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


# Import dataset
dataset = pd.read_csv("Wine.csv")
x = dataset.iloc[:, :-1].values
```

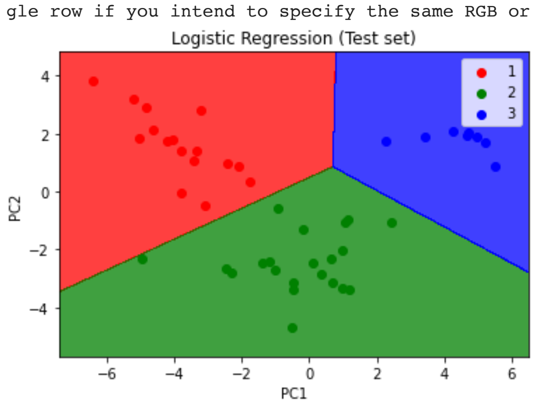
```
y = dataset.iloc[:, -1].values


# Data pre-processing
# Dividing training and test set (ratio: 80 - 20)
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)

# Feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# Fitting and Transforming
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)




# Applying LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components = 2)
x_train = lda.fit_transform(x_train, y_train)
x_test = lda.transform(x_test)




# Using a Logistic Regression model on the new sub-space
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)

# Fitting the regression
classifier.fit(x_train, y_train)




# Model performance review + predictions
from sklearn.metrics import confusion_matrix, accuracy_score

y_pred = classifier.predict(x_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
print("Accuracy Score of PCA is",accuracy_score(y_test, y_pred))




# Visualising the Training Set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)
plt.title('Logistic Regression (Training set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()




# Visualising the Test Set results
from matplotlib.colors import ListedColormap
x_set, y_set = x_test, y_test
x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(np.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())
```

```
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()
```

```
[[16  0  0]
 [ 0 21  0]
 [ 0  0  8]]
Accuracy Score of PCA is 1.0
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
```




```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precede
nce in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array with a sin
gle row if you intend to specify the same RGB or RGBA value for all points.
```



- Student section:

Hypothesis

Dataset

In [ ]:

Model coding

In [ ]:

In [ ]:

In [ ]:

In [ ]:

------------------------------ VIDEOS TO WATCH ------------------------------

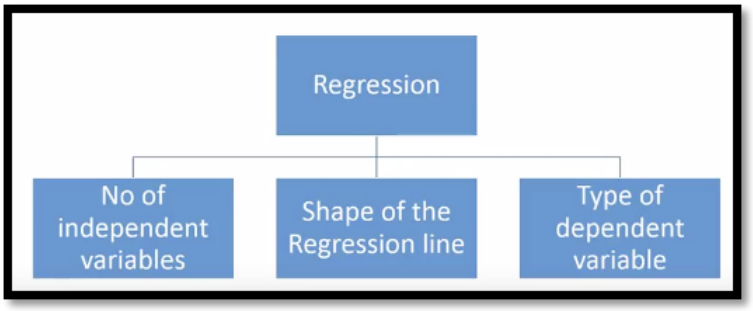
> PCA: https://www.youtube.com/watch?v=g-Hb26agBFg&ab_channel=LuisSerrano
>
> PCA: https://www.youtube.com/watch?v=fkf4IBRSeEc&ab_channel=SteveBrunton
>
> PCA: https://www.youtube.com/watch?v=kw9R0nD69OU&list=LL&index=11&ab_channel=Udacity

>

>

Lab 3 --- Supervised learning (Regression methods)

- Introduction on subject:

> Like we saw in the introduction, supervised learning is a kind of approach that deals with labeled data that allow models to make accurate predictions by approximating the mapping function throught a training process.
>
> We then briefly introduced the concept of regression by stating that it consist of a predicting a continuous quantity. Indeed, the regression problem is resolved by a regression learning algorithm that takes a collection of labeled examples as inputs and produces a model that can take an unlabeled example as input and output a target. Hence, this predictive modelling technique can be seen as a way to understand the relationship between a dependent variable (target) and independent variable (predictor).
>
> In the field of machine learning, regression analysis is often seen as a simple and effective tool to tackle problems involving prediction & forecasting, time series modelling, and finding the causal effect relationships between variables. Many type of regression methods can be used. Factors that differentiate them are: the shape of the regression line, the number of independant variables, and the nature of the dependant variable. The most common class of regression methods are the linear models, which depend linearly on their unknown parameters, making them easier to fit than models which are non-linearly related to their parameters. But during this laboratory, we will explore both as they are critical for an optimal understanding of machine learning methodologies.
>
> > 
> >

- List of methods presented in this laboratory:

> 1. Linear regression
> 2. Polynomial regression ("non-linear")
> 3. Decision Tree Regression
> 4. Random Forest Regression

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

---

1) Linear regression

- Model overview:

> Known as one of the simplest representation in machine learning, a linear regression model can be seen as a way to predicts a target as a weighted sum of its feature inputs. When we are facing only a single (X) and a single (Y), this is known as a **simple linear regression**. Otherwise, when there is multiple input (X) for a single (Y), we call it a **multiple linear regression**.
>
> Linear models can be used to model the dependence of a regression target (Y) on some features (X). The learned relationships are linear and can be written for a single instance i as follows:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon$$

> The predicted outcome of an instance is a weighted sum of its p features. The betas (β_j) represent the learned feature weights or coefficients. The first weight in the sum (β_0) is called the intercept and is not multiplied with a feature. The epsilon (ϵ) is the error we still make, i.e. the difference between the prediction and the actual outcome. These errors are assumed to follow a Gaussian

distribution, which means that we make errors in both negative and positive directions and make many small errors and few large errors. Now for making predictions, it is as simple as solving the equation for a specific set of inputs.

Before further explanation, it is important to understand that the **set of coefficients represent the model** and the **method used to estimate those coefficient is the algorithm**. Hence, learning a linear regression model can be understood as estimating the values of the coefficients. Various methods can be used to estimate the optimal weight. For simple linear regressions, computing statistics related to the dataset can be used. In the case of multiple linear regression, the **ordinary least squares method** is usually used to find the weights that minimize the squared differences between the actual and the estimated outcomes:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} \left(y^{(i)} - \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_j^{(i)} \right) \right)^2$$

- Example 1:

Below is an example of a **simple linear regression model** that is used to make predictions on the salary of an individual (Y) in function of their years of experience (X). You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [9]:

```
# Importing librairies
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
dataset.head()


# data preprocessing
X = dataset.iloc[:, :-1].values  #independent variable array
y = dataset.iloc[:,1].values  #dependent variable vector



# splitting the dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1/3,random_state=0)



# fitting the regression model
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train,y_train) #actually produces the linear eqn for the data



# Predicting the test set results
y_pred = regressor.predict(X_test)

print("- The predicted salaries are", y_pred)

print("- The real salaries of the testset are", y_test)




# Visualizing the results

# plot for the TRAIN

plt.scatter(X_train, y_train, color='red') # plotting the observation line
plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
plt.title("Salary vs Experience (Training set)") # stating the title of the graph

plt.xlabel("Years of experience") # adding the name of x-axis
plt.ylabel("Salaries") # adding the name of y-axis
plt.show() # specifies end of graph
```

```
# plot for the TEST

plt.scatter(X_test, y_test, color='red')
plt.plot(X_train, regressor.predict(X_train), color='blue') # plotting the regression line
plt.title("Salary vs Experience (Testing set)")

plt.xlabel("Years of experience")
plt.ylabel("Salaries")
plt.show()
```

```
- The predicted salaries are [ 40835.10590871 123079.39940819  65134.55626083  63265.36777221
 115602.64545369 108125.8914992  116537.23969801  64199.96201652
  76349.68719258 100649.1375447 ]
- The real salaries of the testset are [ 37731. 122391.  57081.  63218. 116969. 109431. 112635.  55794.  83088.
 101302.]
```

- Example 2:

Below is an example of a **multiple linear regression model** that is used to make predictions on the energy content of a powerplant (Y) in function of multiple factors (X) like temperature, pressure, humidity... Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [4]:

```
# Importing librairies
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


#Importing the dataset
dataset = pd.read_csv('Combined_Cycle _Powerplant.csv')
x1 = dataset.iloc[:, :-1].values
y1 = dataset.iloc[:, -1].values


# Splitting the dataset into the Training set and Test set (ratio used is 80/20 for trainging/testing)
# Importing Package
from sklearn.model_selection import train_test_split
x1_train, x1_test, y1_train, y1_test = train_test_split(x1, y1, test_size = 0.2, random_state = 0)



# Training the Multiple Linear Regression model on the Training set
# Importing Package
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
```

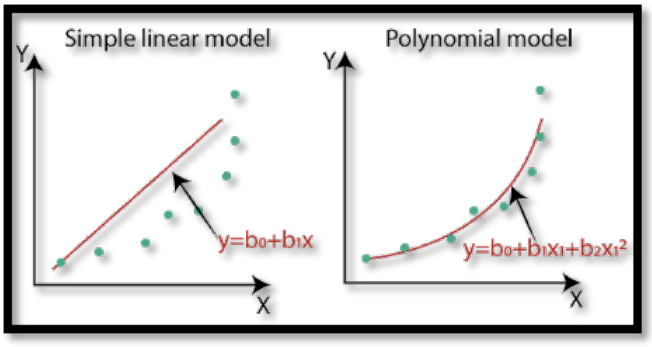
```
# Fitting the model
reg.fit(x1_train, y1_train)



# Predicting the test set result
y1_pred =  reg.predict(x1_test)

# Getting only 2 decimal
np.set_printoptions(precision = 2)

# Concatenating and Reshaping
print(np.concatenate((y1_pred.reshape(len(y1_pred), 1), y1_test.reshape(len(y1_test), 1)), axis = 1))

# First column is the vector prediction and Second vector is the real predication.
# As we can see, prediction are very close.



# Evaluating the Model Preformance
# Importing Package
from sklearn.metrics import r2_score


# R-squared Coefficient (Important to interpret it)
print("The R Squared Coefficient of the Multiple Linear Regression is",r2_score(y1_test, y1_pred))
```

```
[[431.43 431.23]
 [458.56 460.01]
 [462.75 461.14]
 ...
 [469.52 473.26]
 [442.42 438.  ]
 [461.88 463.28]]
The R Squared Coefficient of the Multiple Linear Regression is 0.9325315554761303
```

---

2) Polynomial regression

- Model overview:

> The polynomial models are often used to approximate more complex nonlinear relationship as the application of a higher order polynomial gives you the ability to fit your regression line to a dataset more precisely. The main difference between the Linear and Polynomial Regression is the fact that the ladder does not require the relationship between the independent and dependent variables to be linear in the data. Hence, if your data points clearly will not fit a linear regression (a straight line through most of the data points), it might be ideal for polynomial regression. A simple analogy to understand the intuition behind the model can be: Adding more features to our model, which are themselves powers of already present features.
>
> The main category that we should pay attention here is the more simple **polynomial model in one variable**, or often called second-order model (or quadratic) for a simple explanatory variable, which are represented by:
>
> $$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_k x^k + \epsilon$$
>
> The coefficients B1 and B2 are called the linear effect parameter and quadratic effect parameter, which represent the weights in the equation. B0 is your biais and k is your polynomial degree. Like you can see, the linear regression model seen previously is often said to include the polynomial regression model with it. Thus, the techniques for fitting linear regression model can be used for fitting the polynomial regression model.
>
> > 
> >

- Finding the right degree of polynomial:

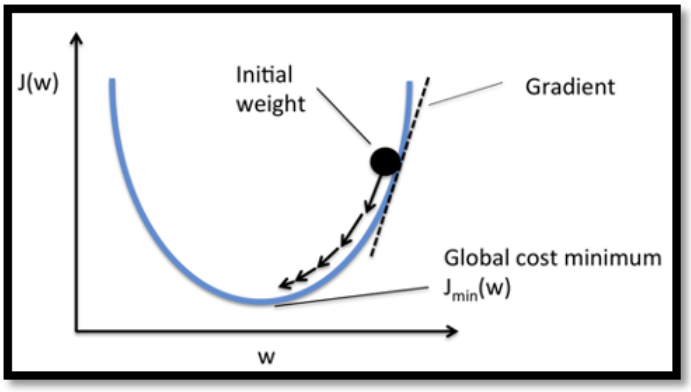
It is important to optimally choose the degree of polynomial of our model as it is based on the relationship between target and predictor. The 1-degree polynomial is a simple linear regression. But as we increase this degree, the complexity of the model also increases. Therefore, the value of n must be chosen precisely. If this value is low, then the model won't be able to fit the data properly (underfitting) and if high, the model will overfit the data easily.

- **Forward selection procedure**: Successively fit the models in increasing order and test the significance of regression coefficients at each step of model fitting. Keep the order increasing until t -test for the highest order term is nonsignificant. This is called a forward selection procedure.
- **Backward elimination procedure**: Another approach is to fit the appropriate highest order model and then delete terms one at a time starting with the highest order. This is continued until the highest order remaining term has a significant t - statistic. The forward selection and backward elimination procedures do not necessarily lead to the same model.

- Fitting the model:

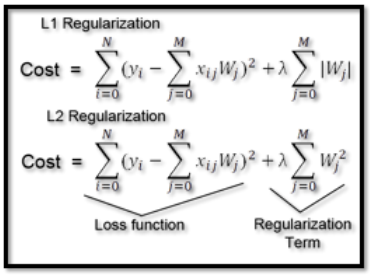
Now for fitting the model and improving its accuracy, minimising our loss function (mean square error) throught an algorithm called **gradient descent** is often done. Indeed, by using the the partial derivative of the cost function to computes the gradient of a model, the gradient descend algorithm allow us to obtain the slope of the cost function at our current position, thus indicating in which direction (gradient) we should move.

This gradient is then multiplied by a tuning parameters named the learning rate in order to control the pace of iteration while moving toward the minimum of our cost function. Finally, the result of this multiplication is then substracted from the weights to decrease the loss of further predictions.



- Regularized regressions:

- Lasso Regression (L1): Penalty term correspond to the absolute weights.
- Ridge Regression (L2): Penalty term correspond to the square of weights



- Example 3:

Below is an example of a **polynomial regression model** that is generated on a fictive dataset. A interesting comparaison is made between a simple linear regression model and a polynomial one. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [4]:
```
# Importing librairies
import numpy as np
```

```
import matplotlib.pyplot as plt

%matplotlib inline
```

```
# Generating the random dataset
m = 100
x1 = 5 * np.random.rand(m, 1) - 2
x2 = 0.7 * x1 ** 2 - 2 * x1 + 3 + np.random.randn(m, 1)

plt.scatter(x1, x2)
plt.show()
```

```
# Applying a Linear Regression to see the results
from sklearn.linear_model import LinearRegression

lr = LinearRegression()

lr.fit(x1, x2)

preds = lr.predict(x1)

plt.plot(x1, x2, 'b+', label='Datapoints')
plt.plot(x1, preds, 'r-', label='Linear Reg Line')
plt.legend()
plt.show()
```

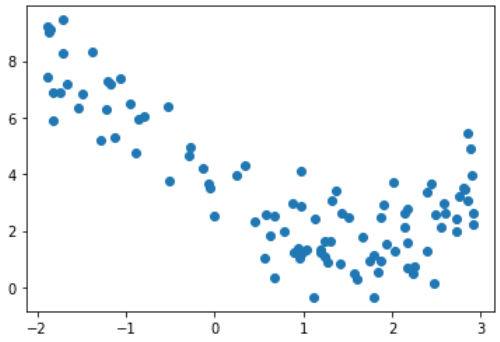
```
# Model evaluation
from sklearn.metrics import r2_score, mean_squared_error
print('RMSE: {:.3f}\nR2: {:3f}'.format(
    np.sqrt(mean_squared_error(x2, preds)), r2_score(x2, preds)))
```

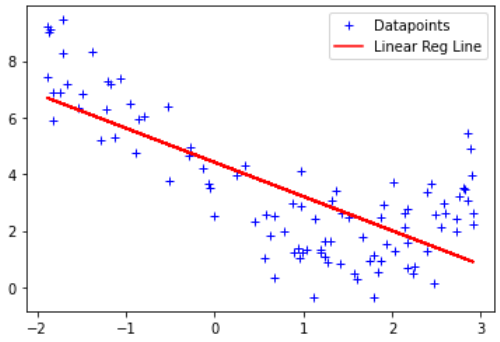
```
# Switching on to the polynomial regression
# Adding the new features through the "PolynomialFeatures" module of scikit-learn
# Then training it
from sklearn.preprocessing import PolynomialFeatures

poly_feats = PolynomialFeatures(degree=2)
x1_poly = poly_feats.fit_transform(x1)

lr = LinearRegression()
lr.fit(x1_poly, x2)

preds = lr.predict(x1_poly)

plt.plot(x1, x2, 'b+', label='Datapoints')
plt.plot(sorted(x1[:, 0]), preds[np.argsort(x1[:, 0])], 'r', label='Polynomial Reg Curve')
plt.legend()
plt.show()
```

```
# Model evaluation
print('RMSE: {:.3f}\nR2: {:3f}'.format(
    np.sqrt(mean_squared_error(x2, preds)), r2_score(x2, preds)))
```

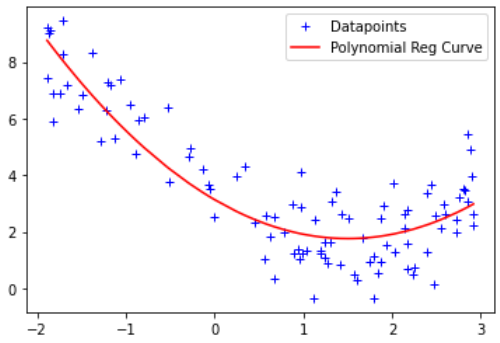
```
# (BONUS) Merging the two steps (PolynomialRegression + LinearRegression) in one with sklearn's pipeline
from sklearn.pipeline import Pipeline

model = Pipeline([
    ('poly_feats', PolynomialFeatures(degree=2, include_bias=False)),
    ('lr', LinearRegression())
])

model.fit(x1, x2)
preds = model.predict(x1)

plt.plot(x1, x2, 'b+', label='Datapoints')
plt.plot(sorted(x1[:, 0]), preds[np.argsort(x1[:, 0])], 'r', label='Polynomial Reg Curve')
plt.legend()
plt.show()
```

```
# (BONUS) Visualising higher degree of polynomial features
def get_preds(x1, x2, degree):
    '''A helper function that will compute Polynomial Features
       and compute predictions
    '''
    poly_feats = PolynomialFeatures(degree=degree, include_bias=False)
    x1_poly = poly_feats.fit_transform(x1)
    lr = LinearRegression()
    lr.fit(x1_poly, x2)
    return lr.predict(x1_poly)

    # degrees we will be working with and corresponding colors
degrees = [1, 2, 8, 16, 30]
cs = ['r', 'g', 'c', 'y', 'k']

    # Politting for degrees
plt.figure(figsize=(10, 6))
plt.plot(x1, x2, 'b+', label='Datapoints')

for degree, c in zip(degrees, cs):
    # Make predictions for degree
    preds = get_preds(x1, x2, degree)
    # Plot
    plt.plot(sorted(x1[:, 0]), preds[np.argsort(x1[:, 0])], c, label='Degree: {}'.format(degree))
    # Print Metrics
    print('Degree: {}\tRMSE: {:.3f}'
          '\tR2: {:.3f}'.format(degree, np.sqrt(mean_squared_error(x2, preds)), r2_score(x2, preds)))

plt.legend()
plt.show()
```

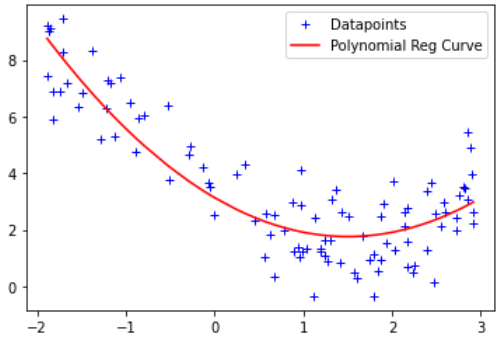




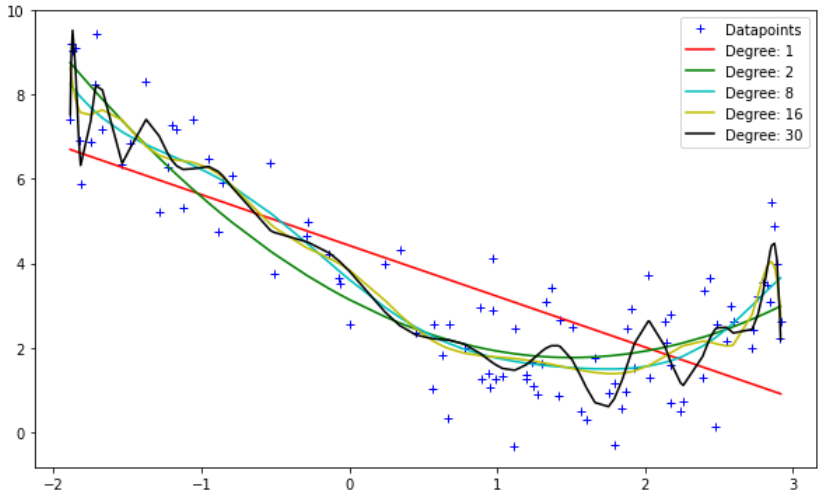

```
RMSE: 1.615
R2: 0.557912
```




```
RMSE: 1.090
R2: 0.798389
```




```
Degree: 1       RMSE: 1.615     R2: 0.558
Degree: 2       RMSE: 1.090     R2: 0.798
Degree: 8       RMSE: 1.012     R2: 0.826
Degree: 16      RMSE: 0.988     R2: 0.835
Degree: 30      RMSE: 0.879     R2: 0.869
```



> ## - Example 4:
>
> > Below is an other example of a **polynomial regression model** that is trained on the previous dataset related to a powerplant in order to make predictions. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [6]:

```
# Importing librairies
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


# Importing the dataset
dataset = pd.read_csv('Combined_Cycle _Powerplant.csv')
x2 = dataset.iloc[:, :-1].values
y2 = dataset.iloc[:, -1].values



# Importing Package
from sklearn.model_selection import train_test_split

# Dividing training and test set.
# The best ratio is 80 - 20 for trainging and testing respectively.
x2_train, x2_test, y2_train, y2_test = train_test_split(x2, y2, test_size = 0.2, random_state = 0)



# Training the Polynomial Linear Regression model on the Training set
# Importing Package
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Tranforming Matrix
poly_reg = PolynomialFeatures(degree = 4)
x2_poly = poly_reg.fit_transform(x2_train)

# Polynomial Model
regressor = LinearRegression()
```

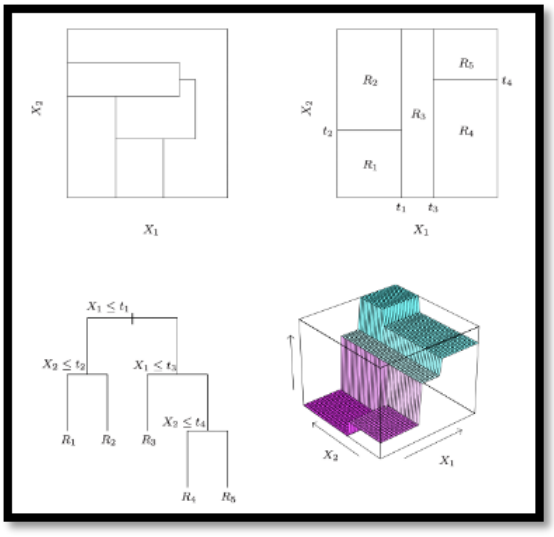
```
regressor.fit(x2_poly, y2_train)



# Predicting the Test set results
# Predicting Regression
y2_pred = regressor.predict(poly_reg.transform(x2_test))
np.set_printoptions(precision=2)

# Concatenating and reshaping
print(np.concatenate((y2_pred.reshape(len(y2_pred),1), y2_test.reshape(len(y2_test),1)),1))




# Evaluating the Model Preformance
# Importing Package
from sklearn.metrics import r2_score

# R-squared Coefficient
print("R Squared Coefficient of Polynomial Linear Regression is",r2_score(y2_test, y2_pred))
```

```
[[433.94 431.23]
 [457.9  460.01]
 [460.52 461.14]
 ...
 [469.53 473.26]
 [438.27 438.  ]
 [461.67 463.28]]
R Squared Coefficient of Polynomial Linear Regression is 0.9458193033689702
```

---

3) Decision Tree Regression

- Model overview:

> The method we are interested in here is known as Classification and Regression Trees **(CART)**, which are powerful decision tree algorithms that can be used for predictive modeling problems. Indeed, by partitionning the feature space into a set of regions (rectangles), the model is fitted in each one by effectively learning simple **decision rules** inferred from the data features.
>
> The CART model may be represented as a **binary tree** where each node represents a single input variable (x) and a split point on that variable (assuming the variable is numeric). The leaf nodes of the tree contain an output variable (y) which is used to make a prediction. To make a prediction with a new input, the tree is simply traversed by evaluating the specific input started at the root node of the tree. Hence, creating a binary decision tree is actually a process of **dividing up the input space in multiple dimensions** based on the similarities of your dataset.
>
> To grow a regression tree, the algorithm needs to automatically decide on the splitting variables and split points, and also what topology (shape) the tree should have, which is optimally done by finding the best binary partition in terms of minimum sum of squares.
>
> > 
> >

- Greedy algorithm (recursive binary splitting):

> The training procedure for this model generally consist of the alignement of all the values where different split points are tried and tested using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner (e.g. the very best split point is chosen each time). For regression predictive modeling problems **the cost function that is minimized to choose spit points is the sum squared error** across all training samples that fall within the rectangle.
>
> By scanning through all of the inputs, the optimal split point is found. We then partition the data into the two resulting regions and repeat the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions. How large should we grow the tree? Clearly a very large tree might overfit the data, while a small tree might not capture the important structure.
>
> In fact, an important **tuning parameter** for this model is the **tree size**, which directly determines the model's complexity. One approach would be to split tree nodes only if the decrease in sum-of-squares due to the split exceeds some threshold. This strategy is too short-sighted, however, since a seemingly worthless split might lead to a very good split below it. The preferred strategy is to grow a large tree T0, stopping the splitting process only when some minimum node size (say 5) is reached.

- Example 5:

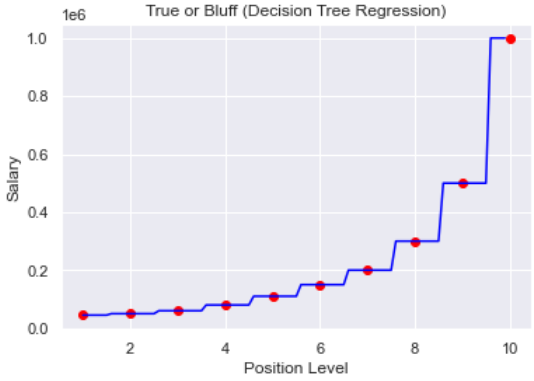
> Below is an example of a **Decision Tree Regression model** trained on a dataset relating the position level of an employee and correspondant salary. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

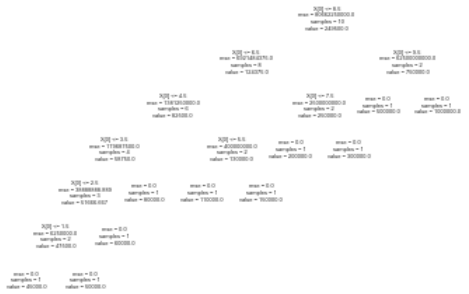
In [39]:

```
# Importing the Libraries
# Numpy allows us to work with array.
import numpy as np

# Maptplotlib which allows us to plot some chart.
import matplotlib.pyplot as plt

# Pandas allows us to not only import the datasets but also create the matrix of features(independent) and
# dependent variable.
import pandas as pd

# Importing dataset
# Independent variable (first columns) / Dependent variable (last columns)
dataset = pd.read_csv("Position_Salaries.csv")
x = dataset.iloc[:, 1:-1].values      # iloc is index of location
y = dataset.iloc[:, -1].values

# Visualizing and describing with statistics the dataset
# DATASET IS NOT SPLIT BETWEEN TRAINING & TEST (BUT IT IS REQUIRED)
print(dataset)
print(dataset.describe())

# Training the Decision Tree Model on the whole dataset
# Importing Library for model
from sklearn.tree import DecisionTreeRegressor

reg = DecisionTreeRegressor(random_state = 0)

# Fitting
reg.fit(x, y)


# Predicting a new Results.
print(reg.predict([[6.5]]))

# ADD MODEL PERFORMANCE R2

# Visualising the Decision Tree Regression result (for higher resolution and smoother curve)
# Increasing the density.
x_grid = np.arange(min(x), max(x), 0.1)
x_grid = x_grid.reshape(len(x_grid), 1)

# Plotting (chart)
plt.scatter(x, y, color = "red")
plt.plot(x_grid, reg.predict(x_grid), color = "blue")

plt.title("True or Bluff (Decision Tree Regression)")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()

#Plotting the tree itself
from sklearn import tree
import matplotlib.pyplot as plt
```

```
tree.plot_tree(reg)
plt.show()
```

```
            Position  Level   Salary
0   Business Analyst      1    45000
1  Junior Consultant      2    50000
2  Senior Consultant      3    60000
3            Manager      4    80000
4    Country Manager      5   110000
5     Region Manager      6   150000
6            Partner      7   200000
7     Senior Partner      8   300000
8            C-level      9   500000
9                CEO     10  1000000
          Level          Salary
count  10.00000       10.000000
mean    5.50000   249500.000000
std     3.02765   299373.883668
min     1.00000    45000.000000
25%     3.25000    65000.000000
50%     5.50000   130000.000000
75%     7.75000   275000.000000
max    10.00000  1000000.000000
[150000.]
```





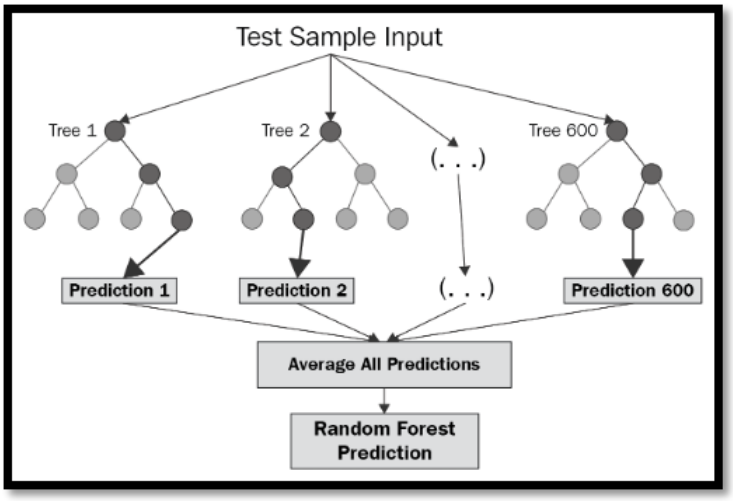
---

4) Random Forest Regression

- Model overview:

> Random forest is an important ensemble learning methodology where models are combination of several hundred to thousands of weaker models to produce a very strong one. It makes many random samples of the training set and then trains a different statistical model on each sample. Indeed, by building multiple decision trees and merging them together, a Random Forest model can get more accurate and stable prediction than many other supervised learning algorithms. In fact, the "forest" it builds is the ensemble of decision trees.
>
> Because every decision tree has high variance, when we decide to combine them in parallel, then the variance become much lower as each decision tree gets perfectly trained on a particular sample of data and hence the output doesn't depend on one decision tree but multiple decision trees. In other words, combining multiple models bring better performance because when several uncorrelated models agree, they are more likely to agree on the correct outcome.
>
> In decision trees (CART), when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split-point. Over time, it will formulate some set of rules, which will be used to make the predictions. The random forest algorithm changes this procedure so that the learning algorithm create individual tree from a different sample of rows and at each node, a different sample of features is selected for splitting. Then, each of the trees makes its

own individual prediction, which are then averaged to produce a single result. Predictions are made by taking the majority vote (for classification) or an average (for regression) of all models, thus reducing the variance.



It is important to understand that Random forest is considered as a bagging technique where you bootstrap and aggregate. In other words, the trees in the algorithm are run in parallel, hence there is no interaction between them while the building process is done. The opposite method is often called boosting.

When it comes to tuning the model, the critical hyperparameter is the number of random features to consider at each split point. Across the literature, a good heuristic for regression is to set this hyperparameter to 1/3 the number of input features. Also, another important hyperparameter when it comes to tuning is the depth of the decision trees. Indeed, the deeper are the trees, the more the data is overfitten but also less correlated, therefore improving the performance of the ensemble. Finally, when using stochastic learning algorithm, it is recommanded to review the performance of your model by averaging performances across multiple runs of cross-validation.

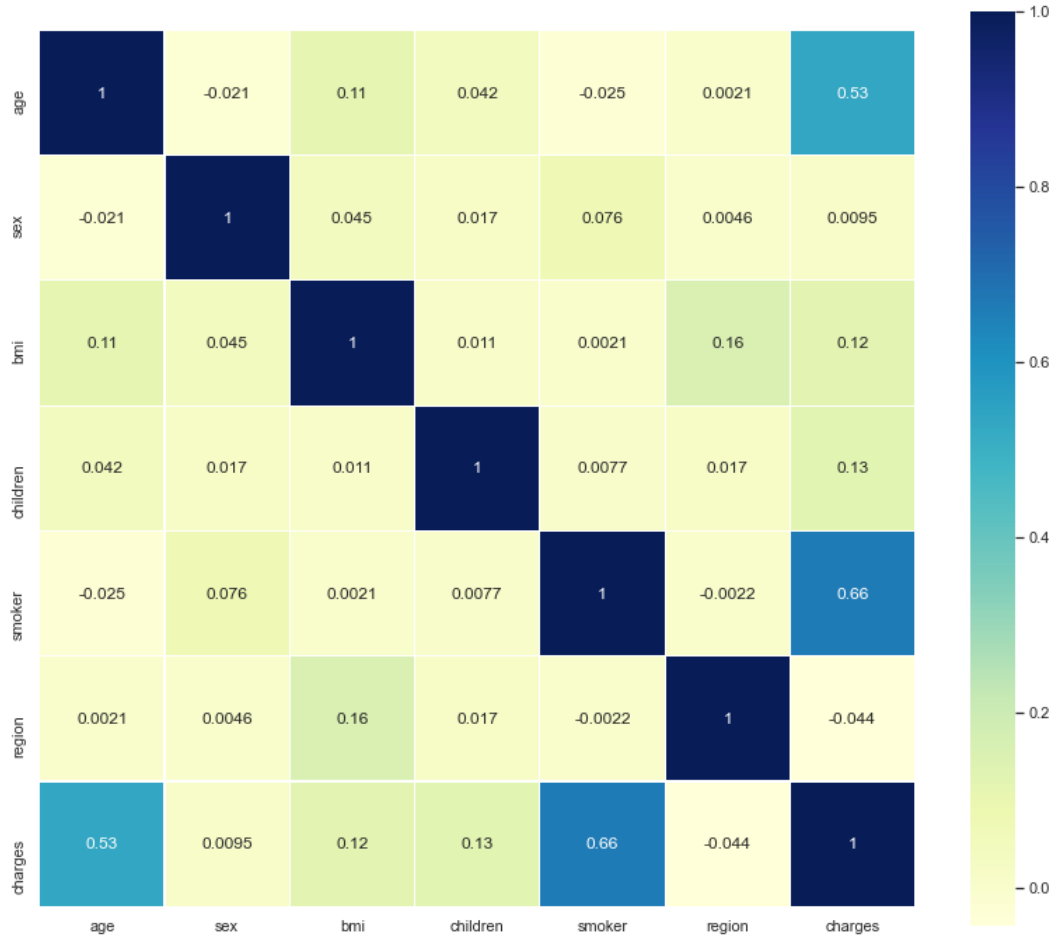
- Example 6:

Below is an example of a **Random Forest Regression** model trained pretty accurately to forecast the possible insurance charges of an individual. For this task, we are using a dataset with the cost of healthcare for a sample of the population given smoking habits, age, sex, bmi, and region. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [40]:
```
# Import librairies
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline



# Import dataset
df = pd.read_csv('insurance.csv')
print(df.head()) # show the first five rows

# Data pre-processing
# Convert string values to integer using the scikit-learn encoder
from sklearn.preprocessing import LabelEncoder
st = df.apply(LabelEncoder().fit_transform)
print(st.head())


# Data visualization (correlation heatmap)
sns.set(color_codes=True)
plt.figure(figsize=(14, 12))
sns.heatmap(st.astype(float).corr(),
            linewidths=0.2,
            square=True,
            linecolor='white',
            annot=True,
            cmap="YlGnBu")
print(plt.show())
```

```
# Data pre-processing
# Converting the vectors of categorical features into a vector-matrix multiplication
# Useful for treating data as numbers
df['age'] = df['age'].astype(float)
df['children'] = df['children'].astype(float)
df = pd.get_dummies(df)
print(df.head())


# Spliting the dataset (10% testing data / 90% training)
from sklearn.model_selection import train_test_split
y = df['charges']
X = df.drop(columns=['charges'])
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, random_state=42)



# Iniating and training the model
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor(n_estimators = 50).fit(X_train, y_train)

rfr_train_pred = rfr.predict(X_train)
rfr_test_pred = rfr.predict(X_test)


# Robust model performance assessment
# MSE = Mean Squared Error
# RMSE = Root Mean Squared Error (More interpretable as same units to vertical axis (Y))
# Cross-validated metric = More accurate estimate of out-of-sample accuracy (All observations used for training/testing)
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.model_selection import cross_val_score

rfr_train_mse = mean_squared_error(y_train,rfr_train_pred)
rfr_test_mse = mean_squared_error(y_test,rfr_test_pred)

print('MSE train data: {:.5}, MSE test data: {:.5}'.format(rfr_train_mse, rfr_test_mse))
print('RMSE train data: {:.5}, RMSE test data: {:.5}'.format(
    np.sqrt(np.absolute(rfr_train_mse)),
    np.sqrt(np.absolute(rfr_train_mse))))
print('R2 train data: {:.5}, R2 test data: {:.5}'.format(
    r2_score(y_train, rfr_train_pred),
    r2_score(y_test, rfr_test_pred)))

rfr_cv = cross_val_score(RandomForestRegressor(n_estimators = 50), X, y, cv = 7) # accuracy +/- 2 standard deviations
print("Accuracy: {:.2} (+/- {:.2})".format(rfr_cv.mean(), rfr_cv.std() * 2))
```

```
   age     sex     bmi  children smoker     region      charges
0   19  female  27.900         0    yes  southwest  16884.92400
1   18    male  33.770         1     no  southeast   1725.55230
2   28    male  33.000         3     no  southeast   4449.46200
3   33    male  22.705         0     no  northwest  21984.47061
4   32    male  28.880         0     no  northwest   3866.85520
   age  sex  bmi  children  smoker  region  charges
0    1    0  197         0       1       3     1005
1    0    1  350         1       0       2       57
2   10    1  331         3       0       2      306
3   15    1   73         0       0       1     1097
4   14    1  223         0       0       1      254
```




```
None
    age     bmi  children      charges  sex_female  sex_male  smoker_no  \
0  19.0  27.900       0.0  16884.92400           1         0          0
1  18.0  33.770       1.0   1725.55230           0         1          1
2  28.0  33.000       3.0   4449.46200           0         1          1
3  33.0  22.705       0.0  21984.47061           0         1          1
4  32.0  28.880       0.0   3866.85520           0         1          1

   smoker_yes  region_northeast  region_northwest  region_southeast  \
0           1                 0                 0                 0
1           0                 0                 0                 1
2           0                 0                 0                 1
3           0                 0                 1                 0
4           0                 0                 1                 0

   region_southwest
0                 1
1                 0
2                 0
3                 0
4                 0
MSE train data: 3.7172e+06, MSE test data: 2.2795e+07
RMSE train data: 1928.0, RMSE test data: 1928.0
R2 train data: 0.97479, R2 test data: 0.83477
Accuracy: 0.83 (+/- 0.087)
```

- Student section:

Hypothesis

Dataset

In [ ]:

Model coding

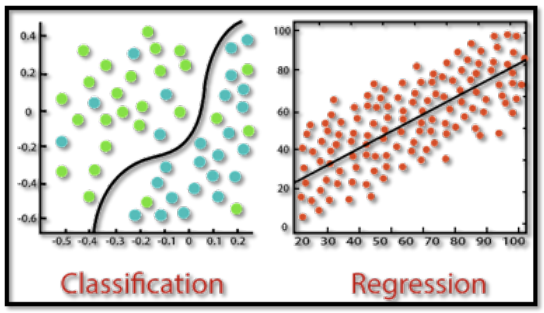
In [ ]:

In [ ]:

In [ ]:

Lab 4 --- Supervised learning (Classification)

- Introduction on subject:

> Now that we have seen the methodologies for supervised learning applied to regression tasks, it is time to study the methodologies for supervised learning applied to classification tasks. Indeed, instead of a real value as the output, we will now have a **category (or class) as the output** for our algorithm, which is often seen as the most used tool in decision making problem. By building a predictive model to forecast class labels, Machine Learning classifiers are able to utilizes some training data to understand the **relation between a given input variable and a specific class**, hence playing a key role when replacing humans for automating part of some complex decision process.
>
> Classification Algorithms can be seen as being part of two category: the **linear** models and the **non-linear** ones. The former include models such as the Logistic Regression and the Support Vector Machines (SVM) while the ladder include models like the K-Nearest Neighbours (KNN), the Kernel SVM or the Naïve Bayes classifier. As the methodologies of Decision Trees and Random Forest were already presented in the past laboratory, their use in classification will not be presented here but is nonetheless instrumental as their performance are very striking.
>
> > > 
> > >

- List of methods presented in this laboratory:

> 1. Logistic regression
> 2. Naive bayes classifier
> 3. Support-vector machine
> 4. K-Nearest-Neighbors (KNN)

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

---

1) LOGISTIC REGRESSION

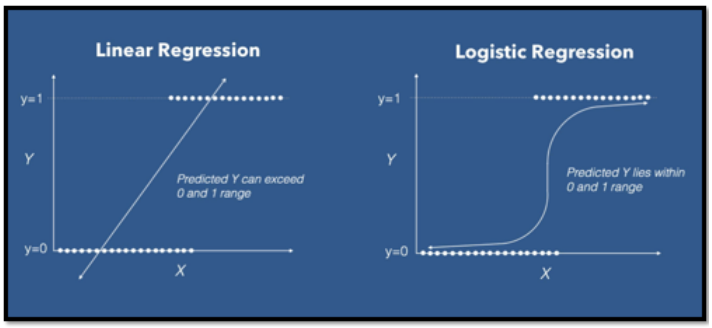
- Model overview:

> When it comes to classification tasks, a linear regression model is often not optimal as it does not output probabilities to the predicted outcome, therefore there is no meaningful threshold at which you can distinguish one class from the other. The solution for a classification task is to use the logistic (or "Sigmoid") function to squeeze the output of a linear equation between 0 and 1, which is now called a logistic regression. Instead of modeling the relationship between outcome and features with a linear equation, we use the weighted sum of our parameters for obtaining probabilities between 0 and 1 as the outcome, hence removing the previous linear influence of our weights on our target.

$$log\left(\frac{P(y=1)}{1-P(y=1)}\right) = log\left(\frac{P(y=1)}{P(y=0)}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

> We call the term in the log() function "odds" (probability of event divided by probability of no event) and wrapped in the logarithm it is called log odds. This formula shows that the logistic regression model is a linear model for the log odds, hence predicting dichotomous variables where the probability of occurence of feature 1 (or "is the feature present?") is estimated.

> Now when it comes to the cost function to minimize the errors, this is not the same as the classical linear regression Mean Squared Error as we are now with a non-linear prediction function and this will end up with many local minimums that will complicate the convergence task toward the global minimum. Instead, we use the famous Cross-entropy loss function, where the loss increases as the predicted probability diverges from the actual label. Finally, the minimization process is done through a traditional gradient descent function on each parameter of our equation. Of course, when we are faced with a classification task that need a binary discrete class (true or false), it is important to select a threshold value above which we will classify values into each of the two classes.



- Example 1:

> Below is an example of a **Logistic regression** algorithm that was applied in a classification task in order to classify which type of customer is buying a specific product based on their age and their estimated salary. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


# Importing dataset
dataset = pd.read_csv('Social_Network_Ads.csv')
x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values



# Data pre-processing
# Dividing training and test set (80 / 20)
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25, random_state = 0)


# Feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# Fitting and Transforming
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)



# Training the Logistic Regression model on the Training set
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)

# Fitting
classifier.fit(x_train, y_train)



# Predicting the Test set results
y_pred = classifier.predict(x_test)

# Concatenating and Reshaping
# First column is the predicted value and Second column is the real value.
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

```
# Model performance (Confusion matrix + Accuracy Score)
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
print(accuracy_score(y_test, y_pred))



# Visualising the Training Set results
from matplotlib.colors import ListedColormap
x_set, y_set = sc.inverse_transform(x_train), y_train
x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 10, stop = x_set[:, 0].max() + 10, step = 0.25),
                     np.arange(start = x_set[:, 1].min() - 1000, stop = x_set[:, 1].max() + 1000, step = 0.25))

plt.contourf(x1, x2, classifier.predict(sc.transform(np.array([x1.ravel(), x2.ravel()]).T)).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)

plt.title('Logistic Regression (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()



# Visualising the Test Set results
from matplotlib.colors import ListedColormap

x_set, y_set = sc.inverse_transform(x_test), y_test

x1, x2 = np.meshgrid(np.arange(start = x_set[:, 0].min() - 10, stop = x_set[:, 0].max() + 10, step = 0.25),
                     np.arange(start = x_set[:, 1].min() - 1000, stop = x_set[:, 1].max() + 1000, step = 0.25))

plt.contourf(x1, x2, classifier.predict(sc.transform(np.array([x1.ravel(), x2.ravel()]).T)).reshape(x1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)

plt.title('Logistic Regression (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

---

2) NAIVE BAYES CLASSIFIER

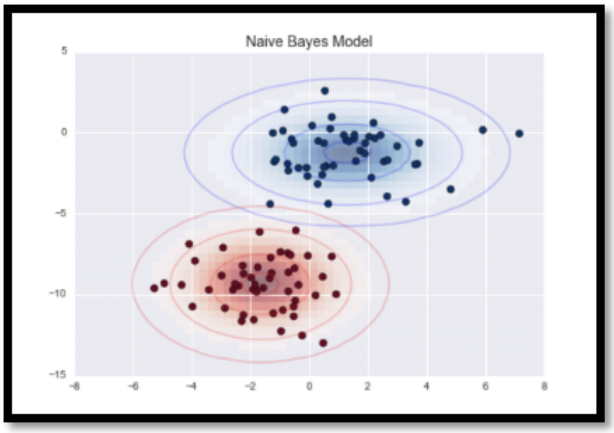
- Model overview:

> The Naive Bayes classifiers is a famous classification method that **rely on Bayes's theorem** in order to find the probability of a label given some observed features. In machine learning we are often interested in selecting the best hypothesis (h) given some data (d). Hence, in a classification task, the hypothesis (h) can be seen as the class to assign for a new observation (d). In this case, **using our prior knowledge** about the problem to select the most probable hypothesis of a given observation is often a good method to follow. Our tool of preference for that is the Bayes' Theorem:

$$P(h \mid \mathrm{d}) = \frac{P(\mathrm{d} \mid h)P(h)}{P(\mathrm{d})}$$

> In this equation, P(h|d) can be seen as the probability of hypothesis h given the data d, or simply the posterior probability. P(d|h) is the probability of data d given that the hypothesis h was true, P(h) is the probability of the hypothesis h being true regardless of the data, also called the prior probability of h. Finally, P(d) is the probability of the data regardless of the current hypothesis we have. To make our decision about which label to attribute, we simply **compute the ratio of the posterior probabilities for each label** (or hypothesis) and we select the label with the highest probability, also called the maximum probable hypothesis (or maximum a

posteriori). Of course, the fact that it is simple is also based on the strong assumption that each hypothesis are assumed to be **conditionally independent** given the target value, which is most unlikely in the real world but works very well when this hold.



As there is no optimization process in the training phase, this method is known to very very fast and only a list of probabilities (Class and conditional) are stored for a trained model. Also, it is important to know that Naive Bayes can be extended to real-valued attributes, often through a Gaussian distribution assumption where you only need to estimate the mean and the standard deviation from your training data. This approach is called **Gaussian Naive Bayes**. Applications of naive Bayesian classifiers are often found where there is a need for a very fast probabilistic predictive model with very few tunable parameters were the naive assumption is thought to hold on. Their performance is also very good for **high-dimensional data** where model complexity is not an issue. Indeed, as the number of informative dimension in a dataset grows, the proximity between observations is diminishing continuously, hence making the task of clustering more feasible for simple classifiers.

- Example 2:

Below is an example of a **Naive Bayes classifier** that was applied for a Natural Language Processing (NLP) task in order to classify if a restaurant review was negative or positive. For that, we trained our model on many past reviews (small sentences of words) that are labeled as positive or negative. In the prediction phase on our test set, the model made 55 correct prediction of negative reviews and 91 correct prediction of positive reviews. On the other hand, it made 42 incorrect prediction of positive reviews and 12 incorrect prediction of negative reviews. For a simple model, this performance seems very acceptable. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:
```
# Importing packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


# Importing dataset
    # The full form of tsv is Tab Sepreated Variable.
    # Delimiter is used to read file in tab form.
    # Quoting is used to avoid Execution error.
dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter = '\t', quoting = 3)



# Data pre-processing (cleaning text)
# Punctuation, different characters, lower case, upper case...
# Symbols of stop words: Words which we dont want to include in our review (All Articles and Pronoun)
# Stemming: taking only the root of a word for its basic meaning (reducing the final dimension of the sparse matrix)

import re
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
corpus = []
for i in range(0, 1000):           # 1000 is used as number of observation in dataset.
    # Remove all punctuation, sub is used replace all the punctuation by space. "^ = not"
    review = re.sub('[^a-zA-Z]', ' ', dataset['Review'][i])
    # Tranform all capital letter to lower case.
    review = review.lower()
    # Splitting the reviews
    review = review.split()
    # Stemming
    ps = PorterStemmer()
```

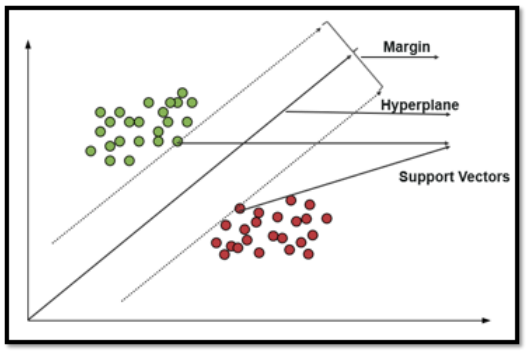
```
    all_stopwords = stopwords.words('english')
    all_stopwords.remove('not')
    review = [ps.stem(word) for word in review if not word in set(all_stopwords)]
    review = ' '.join(review)    # Adding and Spacing in review
    corpus.append(review)



# Creating the Bag-of-Words Model (Used in NLP)
# Creating a List for document where each word is a key + value for number of occurrences
# Tokenization
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features = 1500)
# Fitting and Transforming
x = cv.fit_transform(corpus).toarray()
y = dataset.iloc[:, -1].values



# Splitting the dataset into the Training Set and Test Set (80 / 20)
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 0)



# Training the Naive Bayes Model on the Training Set
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
# Fitting
classifier.fit(x_train, y_train)




# Predicting the Test Set results
# First column is the predicted value and Second column is the real value.
y_pred = classifier.predict(x_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))




# Model performance
# Confusion Maxtrix + Accuracy Score
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
print(accuracy_score(y_test, y_pred))

#55 correct prediction of negative reviews, 91 correct prediction positive reviews
#42 incorrect prediction of positive reviews and 12 incorrect prediction of negative reviews
```

---

3) SUPPORT-VECTOR MACHINE (SVM)

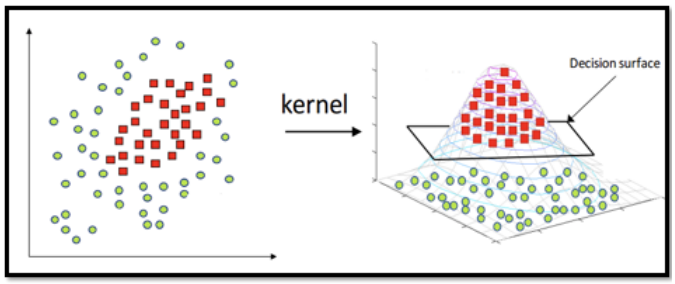
- Model overview:

> The Support-Vector Machine (SVM) algorithm is a type of **discriminative classifier** where a line/curve (2D) or a hyperplane is selected to best separate the points in the input variable space by their classes. Rather than modeling each class by their distribution like the Naives Bayes method (generative classification), we plot each data item as a point in n-dimensional space (where n correspond to the number of features) and we classify the data by finding the **optimal hyper-plane** that differentiates the two classes. Thus, those type of models are called margin classifier, as they are taking decisions for each example based on a specific distance from a pre-determined decision boundary.
>
> To understand this model, it is important to know some definitions. First, **Support Vectors** can be represented as coordinates of individual observation that are closest to the hyperplane and form the most important concept in the construction of the classifier. Second, the **hyperplane** is the decision plane where obervations are separated based on their class. Third, the **margin** can be seen as the perpendicular distance between the line and the closest data points of each classes. The larger the margin, the better it is.



In simple terms, the SVM algorithm will use the training data to **maximizes the margin** through an optimization process. To handle overlapping data with no clear separation boundaries, we can tune the SVM model to be softer. Indeed, a **soft margin classifier** is a type of SVM that loosen up the constraint of maximizing the margin distance between categories by allowing more data to violate the separation line. For this, a specific **tuning parameter C** is introduced that characterize the amount of violation of the margin in a specific model. Hence the larger the value of C, the more violations of the hyperplane are permitted, which itself directly influence the number of support vectors used by the model. Then, we can deduce that a smaller C value will generate a higher sensitivity to the training data (possibly overfitting) while a larger C value will generate a lower sensitivity to the training data (possibly underfitting).

When implementing the SVM algorithm against non-linear datasets, it is often used with a **kernel**, which can be seen as transforming the problem using some linear algebra dot-products equations in order to find the new vector (x) with all the optimal support vectors of the training data. In this case, some people decide to differentiate it from the regular SVM model by calling it the Kernel SVM, but in our case, we will just assume the model as an extension of the regular one. Now by combining kernels with SVM, we can project our data into **higher-dimensional space** and fit a linear classifier model for nonlinear relationships. This is often called the Kernel trick, where we will make use of more complex kernels to separate better spaces into clear different classes, which lead to more accurate classifiers. For linear SVM, the kernel distance is a linear combination of the inputs. Other type of Kernels are used to transform the input space into higher dimensions, where the most common are **Polynomial Kernel** and the **Radial Kernel**.



The Lagrangian optimization process can be understood as a **Quadratic Programming** problem where sub-problems are often solved analytically (by calculating) rather than numerically (by searching or optimizing). As the goal of the learning algorithm is to find values for the **coefficients that best separates the classes**, the problem is often posed with multiple constraints where in some cases, other methods like Subgradient can be used to itterate through the training set in order to find a set of stable coefficients. In simpler terms, the quadratic programming problem for linear data can be formulated as minimizing the objective function in order to find a solution that enable us to classify a vector z with a classification score representing the distance z from the decision boundary. For non linear data where problems do not have a simple hyperplane for optimal separation, the kernel trick rely on computational method for hyperplanes classification that can be resumed as dot products that will help us maximize the margin between the data points and the hyperplane.

When it comes to finding the right hyper-parameters for SVM, trying all combinations to pick the best one is often done. Example of hyper-parameters are the C seen earlier or specific kernel values like Gamma. A popular method called **Gridsearch** is often implemented with Scikit-learn as it create a grid of hyper-parameters and try all possible combinations to pick the best one. Of course, all this is done automatically for us. Finally, as one of the most powerful classification tool, SVM offers great accuracy while taking much less memory than other methods as only a subset of training points are used to take decisions, hence making the prediction phase very fast. But this comes with some downsides like a slow training time with **high computational cost** with larger datasets due to the iterative process. Also, the lack of probabilistic interpretation and their difficulties with overlapping classes make them only suitable for specific applications.

- Example 3:

> Below is an example of a **Support-Vector Machine (SVM)** classifier that was applied for a facial recognition problem where we used some already labeled faces of several thousand photos related to public figures to predict their name in our test set. For that, we used the Wild dataset that is directly accessible from Scikit-Learn. In our case, the feature extraction from the images where algorithm need to find the faces is already done, hence implementing the model is quite simpler. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

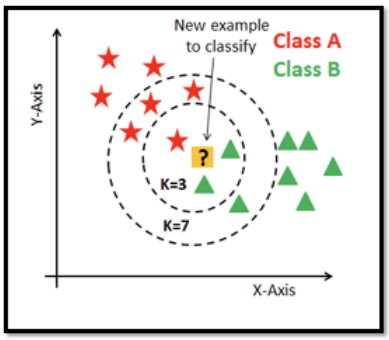
```
# Importing packages
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns; sns.set()



# Importing dataset
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)



# Dataset visualisation
fig, ax = plt.subplots(3, 5)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]])



# Dimensionality reduction
# PCA for extracting more meaningful features
# Go from 3,000 pixels (Images are 62×47) to 150 fundamental components
from sklearn.svm import SVC
from sklearn.decomposition import RandomizedPCA
from sklearn.pipeline import make_pipeline

pca = RandomizedPCA(n_components=150, whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')
model = make_pipeline(pca, svc)



# Splitting datasets (training and testing)
from sklearn.cross_validation import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data, faces.target,
                                                random_state=42)



# Implementing Grid search cross-validation to explore combinations of parameters for best model
# Adjust C (which controls the margin hardness)
# Adjust gamma (which controls the size of the radial basis function kernel)
from sklearn.grid_search import GridSearchCV
param_grid = {'svc__C': [1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
grid = GridSearchCV(model, param_grid)

%time grid.fit(Xtrain, ytrain)
print(grid.best_params_)



# Data prediction + visualisation
model = grid.best_estimator_
yfit = model.predict(Xtest)

fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
    axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
    axi.set(xticks=[], yticks=[])
    axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                   color='black' if yfit[i] == ytest[i] else 'red')
fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);



# Model performance
```

```
# Classification report (lists recovery statistics label by label)
# And confusion matrix
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
                            target_names=faces.target_names))

from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=faces.target_names,
            yticklabels=faces.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

---

4) K-NEAREST-NEIGHBORS (KNN)

- Model overview:

> To close this laboratory, we will introduce the K-nearest neighbors (KNN) algorithm, which can be seen as the most simple methodology of this section. Indeed, this supervised learning algorithm uses 'feature similarity' where predictions outcomes are made for new observations by searching through the entire training set for the **k most similar instances** ("neighbors") and classifying the output variable for those k instances. Hence, the KNN architecture has no model and only the entirety of the dataset is stored for prediction. When it comes to **regression**, the output may be the mean variable while in **classification**, the output can be calculated as the class with the **highest frequency** from the k-most similar instances.
>
> Computing the k value may be done in different ways according to the nature of the problem. In essence, a **distance measure** is always used to determine the k instances from the training dataset that are the most similar to a new observation. Many distance measures can be choosen like the **Euclidean** one for real-valued data or the **Hamming distance** when it comes to comparing two binary data strings (+ when one-hot encoding). It is often recommanded to experiment with different distance metrics in order to find the one that yield the most accurate model.
>
> 
>
>
> KNN is often refered as a **nonparametric** learning algorithm as it makes no assumptions about the functional form of a specific task. But this fact does not affect its performance as it is considered one of the most accurate machine learning classifier. In some cases, it is also used in other architecture for more complex tasks like speech recognition or handwriting detection. Now, here are the main steps for KNN when identifying the label for a test data section:
>
> > **-Step 1**: Choose the integer K of desired neighbors.
> >
> > **-Step 2**: For each point in the test data:
> >
> > **-Step 2.1**: Compute the distance (of choice) between the test data and each row of training data.
> >
> > **-Step 2.2**: Based on the calculated distance value, sort them in ascending order.
> >
> > **-Step 2.3**: Choose the top K rows from the sorted array.
> >
> > **-Step 3**: Assign the new data point to the most frequent category of the array.

- Example 4:

> Below is an example of a **K-Nearest-Neighbors (KNN)** algorithm used to classify a patient breast cancer presence according to multiple features corresponding to characteristics of a patient biological cells (ex: Cell size, Cell shape...). For this task, we will use the open source dataset from the UCI repository with real data of breast cancer

> patients. Again, you can visualize each step with the linked commentaries and below is the related code that you need to execute:

In [ ]:
```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd



# Importing dataset
dataset = pd.read_csv('Breast_Cancer.csv')
x2 = dataset.iloc[:, :-1].values
y2 = dataset.iloc[:, -1].values




# Splitting the dataset into the Training set and Test set (80/20 ratio)
from sklearn.model_selection import train_test_split
x2_train, x2_test, y2_train, y2_test = train_test_split(x2, y2, test_size = 0.25, random_state = 0)




# Feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# Fitting and Transforming
x2_train = sc.fit_transform(x2_train)
x2_test = sc.transform(x2_test)




# Training the KNN Model on the Training set
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)

# Fitting
classifier.fit(x2_train, y2_train)




# Model performance
# Prediction + Confusion Matrix + Accuracy Score
from sklearn.metrics import confusion_matrix, accuracy_score
y2_pred = classifier.predict(x2_test)
cm2 = confusion_matrix(y2_test, y2_pred)
print(cm2)
print("Accuracy Score of KNN Model is",accuracy_score(y2_test, y2_pred))
```

- Student section:

Hypothesis

Dataset

In [ ]:

Model coding

In [ ]:

In [ ]:

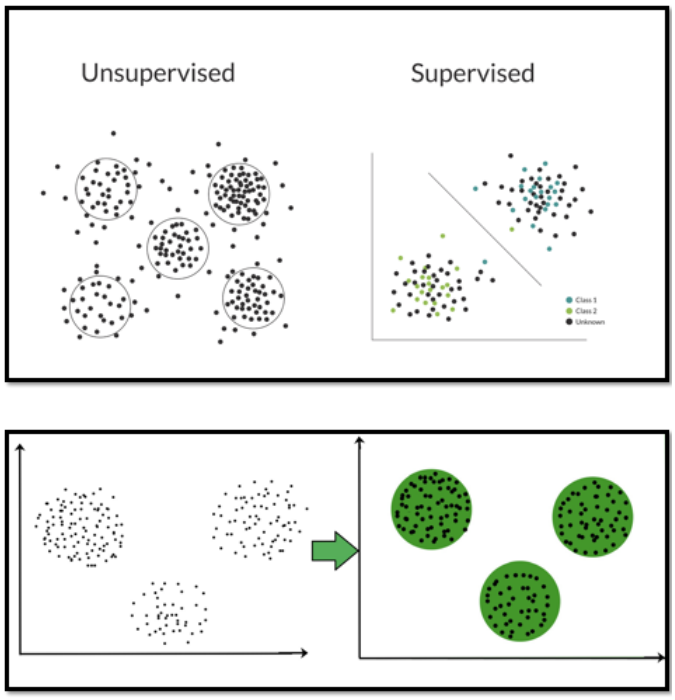
In [ ]:

In [ ]:

In [ ]:

Lab 5 --- Unsupervised learning (Clustering)

- Introduction on subject:

> Across this laboratory, we will shift our focus from supervised learning methods to **unsupervised** ones. Indeed, not all datasets are a collection of labeled examples, hence some algorithms must **find patterns** in the data in order to extract features and build models from them.
>
> The main categories of unsupervised algorithms that can discover interesting structure in the data are often said to be the **clustering methods** and **association learning**. The former is focused on discovering the inherent groupings in the data while the ladder is interested in finding important relations between different variables. In simple words, clustering refers to partitioning unlabeled observations into clusters (groups) with similar characteristics where a cluster itself is just an area of density in the feature space that has a center named "centroid" and some boundary layer to differentiate itself from other clusters. Now in our case, we will focus mainly on clustering algorithms as they are credited for being one of the crticial methods when it comes to designing applied solutions to problems.
>
> 
>
>
> Indeed, clustering methods can be used in many real world applications. For example, **outlier detection**, also called anomaly detection, require the fast detection of abnormal behaviours in order to secure systems or networks, which clustering methods are very good at. An other application is **image segmentation**, where it is necessary to cluster pixels together in order to reduce the number of different components in an image. These are very useful when it comes to object detection and tracking systems like those used in self-driving vehicules. Other popular applications can be **data labeling, search engines, recommender systems...**

- List of methods presented in this laboratory:

> 1. Centroid-based clustering
> 2. Hierarchical clustering
> 3. Distribution-based clustering

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

---

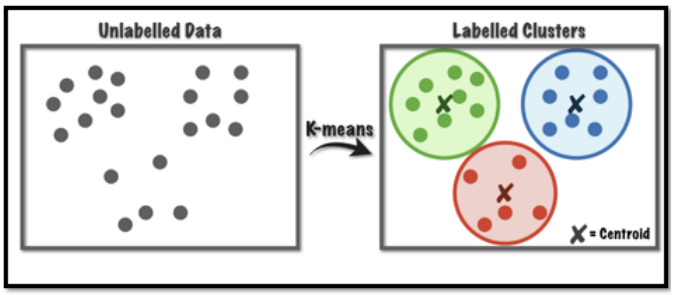
1) Centroid-based clustering

- Overview:

Starting with Centroid-based clustering (also called "Partitional clustering"), these methods are often said to be the simplest and most effective ones when it comes to organizing data. Straightforwardly, they work by initially assigning a parameter k of clusters to a dataset and iteratively assign data points to the closest cluster by measuring different Euclidean distance metrics in order to optimize an objective function.

The main algorithm in this category is named: **k-means**, which main objective can be stated as an optimization problem that will initially assume an initial value for each cluster mean and then will update those and relabel the data until it has converged. Often, around four to five iterations are needed to find an optimal solution. Hence, the general goal can be seen as minimizing the within-cluster sum of squares over an iterative process.

$$\underset{\boldsymbol{\mu}_j}{\operatorname{argmin}} \sum_{j=1}^{k} \sum_{\mathbf{x}_j \in \mathcal{D}'_j} \|\mathbf{x}_j - \boldsymbol{\mu}_j\|^2$$



Over years, different version of this algorithm was proposed in order to reduce the computationnal cost when it came to clustering large datasets. For example, the mini-batch k-means algorithm is a technique in order to avoid using all the dataset each iteration but a subsample of a fixed size. This method can be very cost effective for Big Data usages as it speed up the convergence rate but result often with a lower cluster quality. Now, here are clear steps defining the process used by K-mean:

- **Step 1** : Chosse the number K of clusters. (can be via the elbow method)
- **Step 2** : Initializing random K point for the centroid
- **Step 3** : Label each data point to the closest centroid in order to form K clusters
- **Step 4** : Computer and place the new centroid of each cluster.
- **Step 5** : Reaasign each data point to the new closest centroid.
- **Step 6**: If any reassignment took place, go to Step 4, otherwise end procedure.

- Example 1:

Below is an example of a **k-means algorithm** that is used to cluster in specific groups numerous customers of a shopping mall based on their level of income and their spending. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd



# Importing Dataset
dataset = pd.read_csv('Mall_Customers.csv')
x = dataset.iloc[:, [3,4]].values
print(x)
```

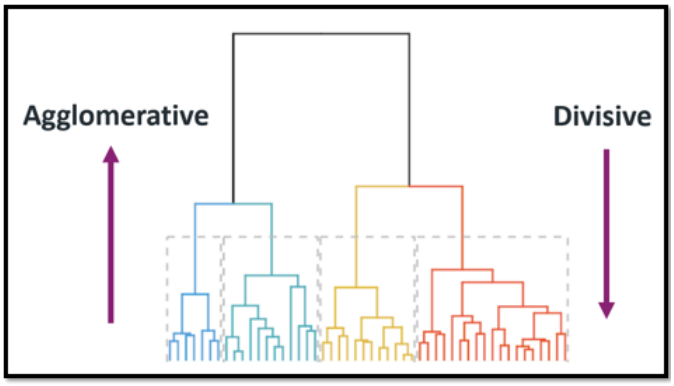
```
# Finding optimal number of cluster via the elbow method
# WSCC - Within Cluster Sum of Square (sum of the square distances between each observation point of the cluster)
# Elbow Method
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)
plt.title("The Elbow Method")
plt.xlabel("Number of Clusters")
plt.ylabel("WCSS")
plt.show()




# Training the Model with 5 cluster
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
y_kmeans = kmeans.fit_predict(x)




# Data visualisation
    # Showing the index value of cluster related to first customer, second customer...
print(y_kmeans)

    # Visualising the Clusters
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'red', label = "Cluster 1")
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'blue', label = "Cluster 2")
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green', label = "Cluster 3")
plt.scatter(x[y_kmeans == 3, 0], x[y_kmeans == 3, 1], s = 100, c = 'orange', label = "Cluster 4")
plt.scatter(x[y_kmeans == 4, 0], x[y_kmeans == 4, 1], s = 100, c = 'cyan', label = "Cluster 5")

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 300, c = "yellow", label = "Centroid")
plt.title("Clusters of Customer")
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.legend()
plt.show()



#The cluster 4 contains all the customers that have a low annual income and who spends very low in the mall.
#The cluster 5 contains all the customers that have a high annual income and who spends very low in the mall.
#The cluster 1 contains all the customers that have a low annual income and yet have a high spending score in the mall.
#The cluster 3 contains all the customers that have a high annual income and also who spend a lot in the mall.
#The cluster 2 contains all the customers that have an average annual income and spends normally in the mall.
```

---

2) Hierarchical clustering

- Overview:

> The second clustering method we will review is named: **Hierarchical clustering**. In contrast to the privous method (k-means), hierarchical clustering is known for creating a hierarchy of clusters by merging or splitting the data through a tree-based methodology called a **dendrogram**. Two types of Hierarchical clustering are used:
>
> The first ones are **Agglomerative hierarchical algorithms**, where each data point is represented initially as a single cluster and then will successively merge in pairs until the data has been successfully merged into a single cluster. This is also known as the bottom-up approach. The second ones are **Divisive hierarchical algorithms**, where all the data points are initially part of a single cluster and are then divided recursively into various smaller clusters until the desired level of separation is achieved. This is also known as the top-down approach.



Often, multiple distance metrics can be used for the separation methodology like the closest point between clusters, the furthest point between clusters, or the average distance between clusters... Now, here are the **main steps for a typical agglomerative hierarchical algorithm**:

> **- Step 1** : Make each data point a single point cluster -> That forms N clusters.
>
> **- Step 2** : Take the two closest data points and make them one cluster -> That forms (N - 1) clusters.
>
> **- Step 3** : Take the two closest clusters and make them one cluster -> That forms (N - 2) clusters.
>
> **- Step 4** : Repeat Step 3 until there is only one cluster. Than END.

- Example 2:

> Below is an example of a **Agglomerative hierarchical clustering algorithm** that is used to cluster in specific groups numerous customers of a shopping mall based on their level of income and their spending. You can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

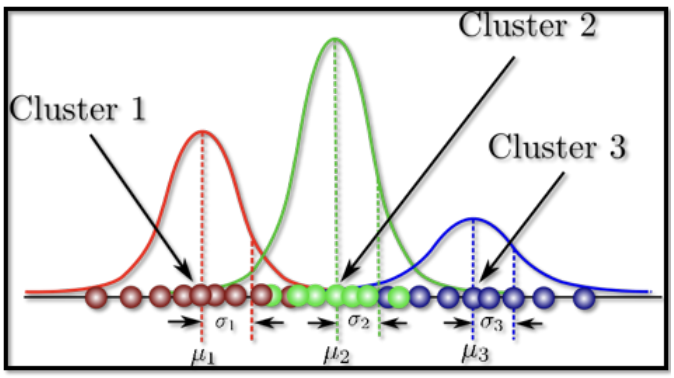
```
# Importing packages
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd



# Importing dataset
dataset = pd.read_csv('Mall_Customers.csv')
x = dataset.iloc[:, [3,4]].values
print(x)



# Using the Dendrogram to find the optimal number of clusters
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(x, method = 'ward'))

plt.title("Dendrogram")
plt.xlabel("Customers")
plt.ylabel("Euclidean Distance")
plt.show()



# Model training
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(n_clusters = 5, affinity = 'euclidean', linkage = 'ward')
y_hc = hc.fit_predict(x)

print(y_hc)



# Visualising the Clusters
plt.scatter(x[y_hc == 0, 0], x[y_hc == 0, 1], s = 100, c = 'red', label = "Cluster 1")
plt.scatter(x[y_hc == 1, 0], bx[y_hc == 1, 1], s = 100, c = 'blue', label = "Cluster 2")
plt.scatter(x[y_hc == 2, 0], x[y_hc == 2, 1], s = 100, c = 'green', label = "Cluster 3")
plt.scatter(x[y_hc == 3, 0], x[y_hc == 3, 1], s = 100, c = 'orange', label = "Cluster 4")
```

```
plt.scatter(x[y_hc == 4, 0], x[y_hc == 4, 1], s = 100, c = 'cyan', label = "Cluster 5")

plt.title("Clusters of Customer")
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.legend()
plt.show()

#The cluster 1 contains all the customers that have a high annual income and yet have a low spending score in the mall.
#The cluster 2 contains all the customers that have an average annual income and spends normally in the mall.
#The cluster 3 contains all the customers that have a high annual income and also who spend a lot in the mall.
#The cluster 4 contains all the customers that have a low annual income and who spends high in the mall.
#The cluster 5 contains all the customers that have a high annual income and who spends very low in the mall.




# Trying with 3 clusters
from sklearn.cluster import AgglomerativeClustering
hc1 = AgglomerativeClustering(n_clusters = 3, affinity = 'euclidean', linkage = 'ward')
y_hc1 = hc1.fit_predict(x)

print(y_hc1)

plt.scatter(x[y_hc1 == 0, 0], x[y_hc1 == 0, 1], s = 100, c = 'red', label = "Cluster 1")
plt.scatter(x[y_hc1 == 1, 0], x[y_hc1 == 1, 1], s = 100, c = 'blue', label = "Cluster 2")
plt.scatter(x[y_hc1 == 2, 0], x[y_hc1 == 2, 1], s = 100, c = 'green', label = "Cluster 3")

plt.title("Clusters of Customer")
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.legend()
plt.show()
```

---

3) Distribution-based clustering

- Overview:

> The last category of clustering in this laboratory are what is called model-based clustering, which unlike the previous heuristic-based algorithms methods where they extract clusters directly from the data, these algorithms **incorporate a notion of uncertainty** through probabilitic methodologies in order to perform the clustering assignment tasks.
>
> Indeed, when it comes to clustering well-separated data, the k-means algorithm can be very powerful. But because it has no way of quantifying the uncertainty of a cluster assignments and has no way of accounting for different clustering shapes (ex: elliptical), a model-based approach like the finite mixture models (FMM) that include a combination of **multiple probability distribution functions** will be much more useful.
>
> A popular model-based approach that we will focus on are the **Gaussian mixture models (GMM)**, where data are considered to be from an underlying probability distributions where each observation has a probability of belonging to each cluster. By measuring the uncertainty in cluster assignment with the comparison of distances of each point to all cluster centers, the GMM model is recognized to find a mixture of Gaussian probability distributions that best model any input dataset. Hence, it can simply be resumed as **fitting k Gaussian distributions to a dataset** with the two main attributes being the mean and the variance of these distributions.
>
> 
>

- Expectation–maximization approach:

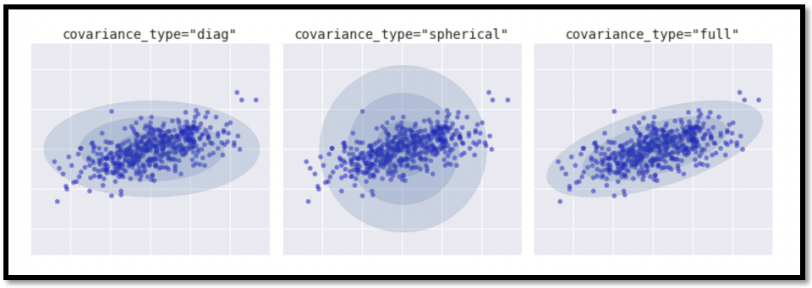
The GMM proceed to **find the maximum-likelihood** using the Expectation-Maximization (EM) algorithm, which recursively estimates a local maximum-likelihood for estimating the true parameters of the distributions. Indeed, it start by initially guessing the mean and the variance for the assumed k-distributions and then updates the weights versus the parameters of the mixtures. One alternates between these two until convergence is achieved. Now, even if it is not always guaranteed that the solution will converge, as it may stay into a local value of the maximum-likelihood, the derivative of the likelihood is always very close to zero, confirming often an optimal solution.

The main idea behind the mixture model can be seen as the **probability density function (PDF) for observations of the data that is composed of a weighted linear sum of a set of unknown distributions**. Hence, each PDFs is weighted and parametrized by an unknown vector of parameters, which are the mean and the variance of the normally distributed function. Now, the goal of the model can be understood as: **given the PDF, we need to estimate the weights and the parameters of the distribution**. Or, assuming a number of k mixtures, we need to find αp, and μp & σp for each mixture.

The algorithmic process can be seen as an **optimization problem to find the local maximizer (derivative = zero) of the maximum likelihood estimate**. Assuming an initial estimate (guess) of the parameter vector, the first step can be perceived as using this to estimate the posterior probability to compute clusters memberships. Then, the second step is where the algorithm updates the parameters and mixture weights using the covariance matrix containing the variance parameters. By alterning those steps, a desired convergence can be reached, where each cluster will be associated with a smooth Gaussian model.

> > **1) Choose starting guesses for the location and shape**
>
> **2) Repeat until converged:**
>
> **- E-step:** For each point, find weights encoding the probability of membership in each cluster. (Bayes rule)
>
> **- M-step:** For each cluster, update location, normalization, and shape based on all data points, using the weights.

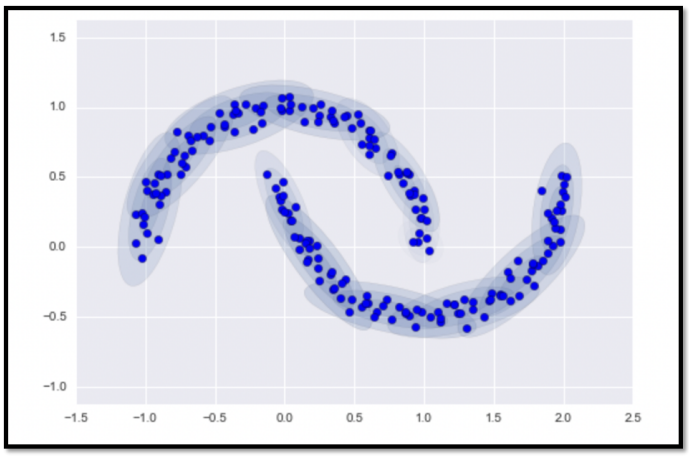
An important hyperparameter of the model is the **type of covariance**, which controls the degrees of freedom in the shape of each cluster. Depending on the nature of the problem, this need to be choosen according to the accuracy desired when clustering the data. For example, an important choice can be the fact that the cluster dimensions can be set independently or with all dimensions equal.

> > > 
> > >

- GMM as a density estimator:

Often seen as a clustering algorithm, the Gaussian mixture models can also be categorized as an **algorithm for density estimation** on more complex datasets. In fact, these can be used as a **generative probabilistic model** representing the underlying distribution of the data. Indeed, by modeling the overall distribution the observations, one can build a generative model to generate new random data distributed similarly to our input.

To understand generative models, one can see them from a perspective where a probability distribution for the dataset is used to evaluate the likelihood of the data under the model. Hence, by using k mixtures of Gaussians, we already seen that we can perform clustering tasks but we can also **model the overall distribution of the input data to generate new similar data** that were not in our dataset. For density estimation tasks, finding the optimal number of clusters can easily be done using cross-validation or the value that minimizes some information criterion like the AIC or BIC, which will avoid over-fitting the data. Generative models will not be something seen in this laboratory but later in the Deep learning section, we will dive into how some astonishing models represent the underlying distribution of the data to generate data sample like images that never existed.



- Example 3:

> Below is an example of a **Gaussian Mixture model (GMM)** that is used to cluster in specific groups some fictive data. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

```
# Import packages
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np



# Generate some data
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=400, centers=4,
                       cluster_std=0.60, random_state=0)
X = X[:, ::-1] # flip axes for better plotting



# Model generation + plotting
from sklearn.mixture import GMM
gmm = GMM(n_components=4).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');



# Visualizing the uncertainty of cluster attribution
# The size of each point is proportional to the certainty of prediction
probs = gmm.predict_proba(X)
size = 50 * probs.max(1) ** 2  # square emphasizes differences
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);




# Function to visualize the locations and shapes of the GMM clusters
# This will be done by drawing ellipses
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()

    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))
```

```
def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covars_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)



# Four-component GMM
# Unstretched dataset
gmm = GMM(n_components=4, random_state=42)
plot_gmm(gmm, X)



# Lets try fitting the stretched dataset
# Changing the hyperparameter "covariance_type" to full
# This will allow the model to fit longer stretched-out clusters
gmm = GMM(n_components=4, covariance_type='full', random_state=42)
plot_gmm(gmm, X_stretched)
```

- Student section:

Hypothesis

Dataset

In [ ]:

Model coding

In [ ]:

In [ ]:

In [ ]:

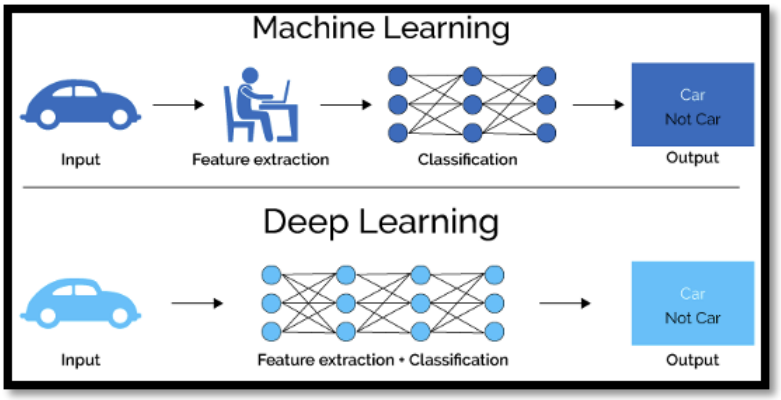
---------- VIDEOS TO WATCH ----------

> GMM / EM: https://www.youtube.com/watch?v=REypj2sy_5U&ab_channel=VictorLavrenko
>
> GMM / EM: https://www.youtube.com/watch?v=iQoXFmbXRJA&ab_channel=VictorLavrenko

---

Lab 6 --- Deep learning (Neural networks)

- Introduction on subject:

> After seeing the different methodologies used for supervised and unsupervised learning, it is time to introduce the famous **subset of machine learning** that is often credited for bringing the most excitement to the future of compuational intelligence. This is called: **Deep learning**.
>
> Starting from the basics, it again produce an input–output mappings representation, but on an other level. Indeed, the word "deep" refers to the fact that those complex functions are often organized into **many layers**, which make them an exception to the previous shallow learning methods where the parameters of a model are directly learn from the features of the training examples. In fact, because deep learning methods incorpore multiple layers between the input and output, parameters of a model are not directly learned from the features of the training examples but from the **outputs of the preceding layers**.
>
> 
>
>
> The most common architecture in deep learning are **Neural Networks**, which emerged from early work that tried to model networks of neurons in the brain with computational circuits. Of course, those network are only an abstraction of the brain processes and not representative of the real neural cells behaviours.
>
> But with the emergence of the Big Data era and **huge dataset** that could serve as training examples, deep neural networks were easily becoming very attractive for modeling functions with the ability to learn features from data. Indeed, their ability to **learn complex features by combining simple features** in a unique way is phenomenal. And by simply tweaking the parameters for adding more layers to a network, this can allow models to progressively learn more complex features by themselves. Hence, this is why Neural Networks are becoming the favorite tool for professionnals when it comes to modeling big data.
>
> Deep neural networks are mostly used these days on **supervised learning** tasks, where the task of modeling requires each training example to be labeled with a value for the target function. These networks can reach very high level of accuracy on specific tasks on conditions that there is very large dataset available for training. On the other side, deep neural networks are still not popular enough in the **unsupervised learning** category, where their ability to learn new representations and produce generative models is still evolving rapidly and can represent a serious field for new discoveries in efficient deep learning research.

- List of methods presented in this laboratory:

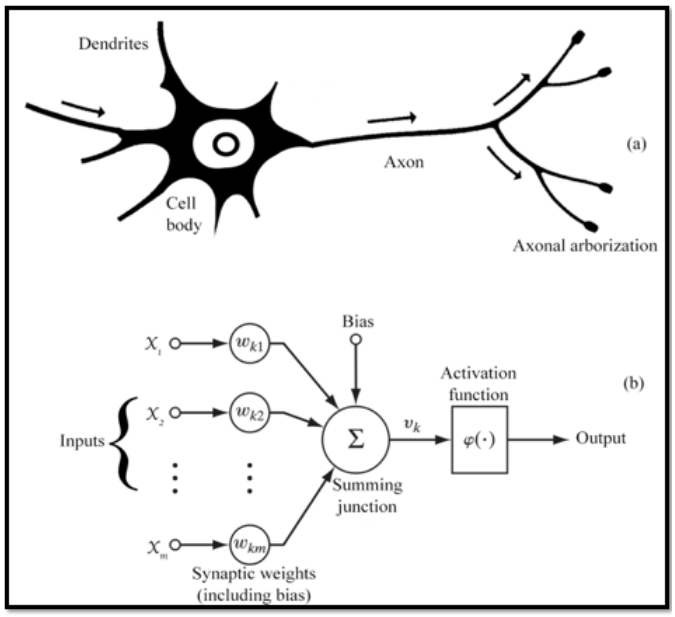
> 1. ANNs for supervised learning
>
>   - `Feedforward Neural Networks`
>   - `Recurrent Neural Networks`
>   - `Convolutionnal Neural networks`
> 2. ANNs for unsupervised learning
>
>   - `Autoencoders`
>   - `Generative adversarial networks`

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

1) FUNDAMENTALS OF ARTIFICIAL NEURAL NETWORKS (ANNs)

- Inspired from nature:

> The fascination of humans for creating intelligent machines really gained traction during the 20th century with scientists like Alan Turing or John von Neumann already stating the possibility of the invention of an "Artificial intelligence". Eventually, the best example we had in order to engineer those machines were us, humans. Hence, computer scientist started to model and understand the research made in Neuroscience in order to reverse engineer ourselves and replicate what was going on in our brain. Eventually, the concept of Artificial neural networks emerged from our current understanding of those processes.
>
> Basically, the brains main components are neurons, which form networks between them to carry information and make computation. The neuron can be modeled as a central cell (soma) where it receives electrochemical inputs from other central cells (somas) via its dendrites and produce outputs via axons which connect to others cells to transmit information. But the key principle to understand is that an activated neuron re-transmits the signal to other neurons via his dendrites ONLY if the electrical input is sufficiently powerful to activate the neuron. This is called an action potential (AP). In other words, a neuron firing is a binary operation (either it fires or not) that is drived by the fact that if the total signal received at the soma exceeds a certain threshold, the neuron will spike.
>
> In artificial neural networks, the idea was partially used to apply it to a graph structures where multiples nodes performs a simple computation, and where each connection between these nodes carries a signal that is the output of those computations. A weight is attributed to to those connections that precise if the signal is amplified or diminished. Large positive weights amplify the signal while negative ones diminish it strength.
>
> 
>

- Layers in ANNs:

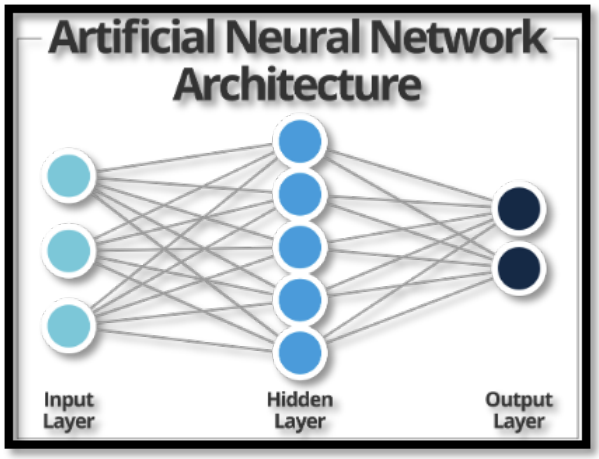
> The basis of the neural networks architecture are layers, which are chain structure that allow each layer in the network to be a function of the layer that preceded it. Indeed, because each node ("neuron") is connected to all the nodes of the next layer via their connection weights, this allow them to take continuous inputs and produce continuous outputs, which are the basis of the parameters the network will seek to adjust in order to learn fitting the data. We can separate networks into 3 main layers:
>
> > **1) Input layer**
> >
> > The input layers is composed of a single row of the independant values x1, x2, x3 [...] where training example are encoded as values that can be 0 or 1 if the attributes are Boolean or numerical values that are often scaled to fit a fixed range. For example, feeding an image into a network will require to transform it into a matrix of pixel representing the values associated to the location of the pixel and its associated color (RGB). Each x is connected to a neuron of the next layer via a weight vector W. Hence for each input x we also have an associated weight w.
> >
> > **2) Hidden layer(s)**

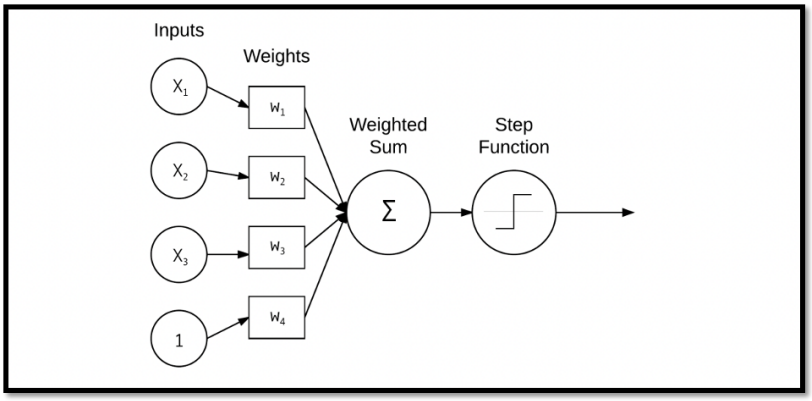
As the network grow in complexity, multiple hidden layers are required for better feature extraction of the data. Essentially, the role of those layers are to transforms the representation of the preceding layer in order to create a new one. Concretely, a hidden layer will take the **weighted sum of the input x and weights w**, then will often **add on a bias value as a discretionnary parameter**, and will then passed all those inputs through the activation function (nonlinear) to produce it output, which can be seen as the **degree of firing related to the neuron**.

In some architectures, only forward connections are allowed, which means connection from nodes in layer i to nodes in layer i+1. These are called feedforward networks. In other cases, certain network architecture possess backward or inter-layer connections, which is the case in recurrent neural networks, where feedforward networks are mixed with feedback connections. More detail on them later.

3) Ouput layer

The output layer is the last layer of neurons and can be interpreted in many ways depending on the desired task. But in general, it can be seen as the layer that provides the parameters for making a distribution over (Y), our dependant value. In most cases, the output layer will typically use a different activation function from the hidden layers as the choice is guided by the type of predictions the model need to make. A common one that is used in regression tasks is simply a linear output layer, where no real activation is used as the weighted sum of the input do not changes. An other one that is used in multiclass classification tasks is called a softmax output layer, where the layer take as input the vector of real values and output a vector of values between 0 and 1 that can be seen as the probability of class membership. Indeed, by producing a vector of nonnegative numbers that sum to 1, this layer can now act as a classifier that give us the probabilities corresponding for each class label. Finally, an other type of ouput layer used in binary or multilabel classification tasks is a layer using a Sigmoid (or Logistic) function, the one we saw earlier.





- Activation functions:

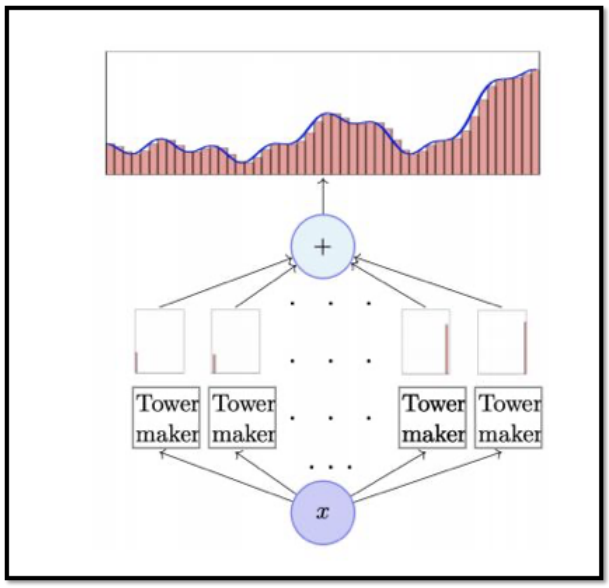
The use of activation functions in neural networks is the key ingredient that allow those models to be able to separate nonlinear classes that are often seen from the real world. Indeed, by using **nonlinear** activation function, the famous **Universal approximation theorem** tell us that we can effectively represent arbitrary functions at the condition that we create a network with enough computationnal hidden units.

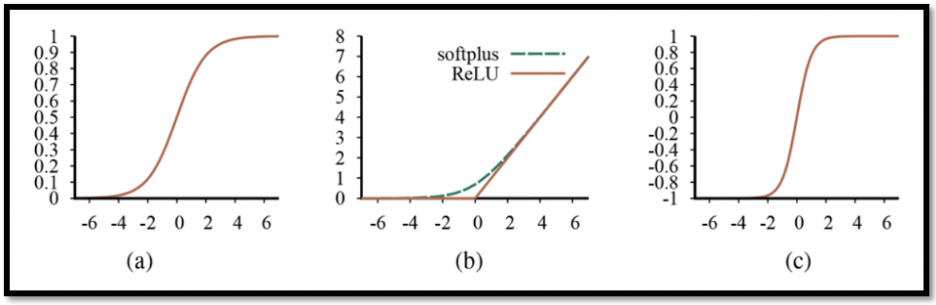
> **Theorem**: *"If a feedforward neural network has a linear layer and at least one layer of activation functions with "squeezing" properties (such as sigmoid, etc.), given a network with a sufficient number of hidden units, it can be*

> *approximated with arbitrary precision Any borel measurable function from one finite dimensional space to another finite dimensional space."*

In other words, as the network grow exponentially, it can represent exponentially many variations of different amplitude at different locations in the input space, thereby **approximating the desired function**. Now that we know it is possible to represent the function we are seeking to model, the main critical task is to find the training algorithm that will learn the right parameters, hence avoiding the trap of overfitting the data.

The activation function task of each node can be described as **computing the weighted sum of the inputs from the predecessor nodes and then applying a specific type of nonlinear function to produce an output**. Choosing the right type of function is critical for an optimal learning process and is often based on the type of neural network architecture we are using. More detail on this later but here is an overview of the three main type of functions:





A) Sigmoid function

> The sigmoid function (logistic) is the same function used in the logistic regression classification algorithm that we saw earlier. It takes any real value as input and **outputs values in the range 0 to 1**. The larger the input, the closer the output value will be to 1. And the smaller the input, the closer the output will be to 0.

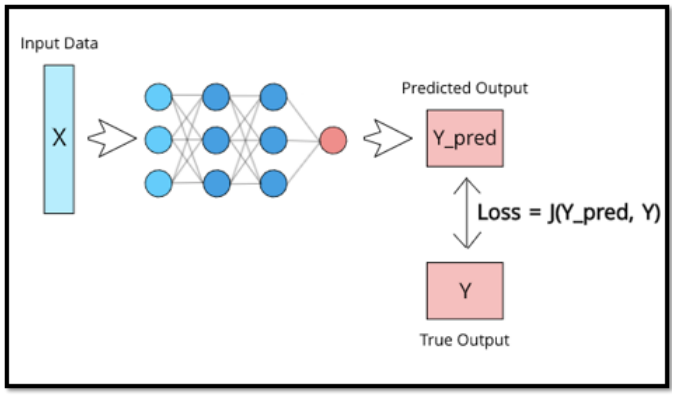
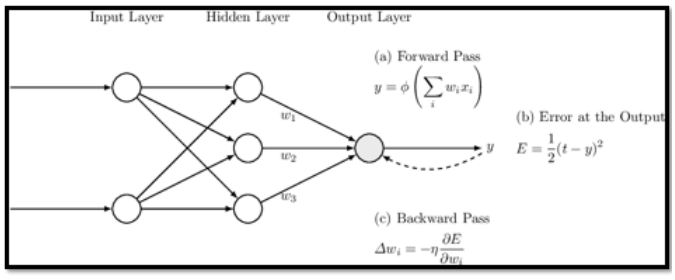
B) ReLU function

> The ReLu function (rectified linear unit) can be defined as: **if the input value (X) is negative, then the value 0 is returned. Otherwise, the value is returned**. It is the simplest one and is often effective at overcoming the vanishing gradients problem, something that we will se later. However, it may suffer from the dead neuron problem without being able to recover, which means it may come to a state where a neuron will always outputs the same value for any input given.

C) Tanh function

> The Tanh function (hyperbolic tangent) takes any real value as input and **outputs values in the range of -1 to 1**. The larger the input, the closer the output value will be to 1. The smaller the input, the closer the output will be to -1. In other words, it is very similar to the sigmoid function, but only with a largeR range.

- Learning process:

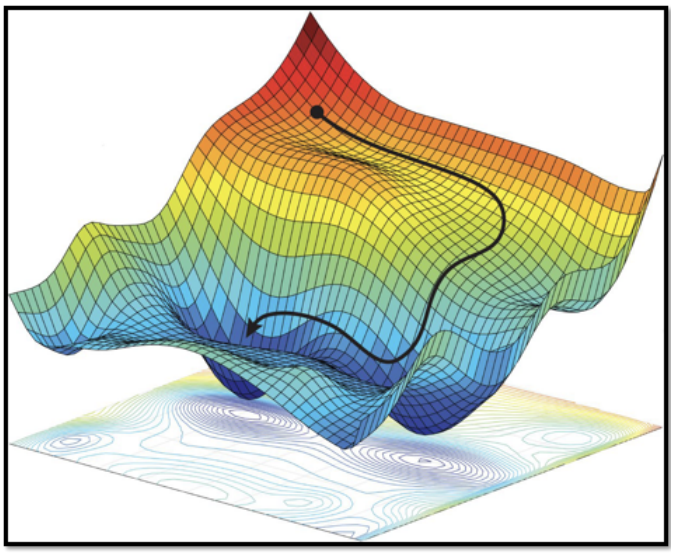
> The learning process of a neural network used on a supervised task can be described as the task of **modifying the network's parameters in order to minimize the errors** made on certain predictions from the training dataset. Let us decompose this process into 4 main steps:
>
> > **1) Forward propagation**
> >
> > First, after creating our network and initialising the weights of the connections, we must apply a forward propagation with the training data (X), which means that the information enter the input layer and will propagate from left to right through the hidden units at each layer and will finally produces our prediction(s) yˆ. This phase can also be described as applying the **scoring function** to our input data or applying the forward pass.
> >
> > **2) Error computing**
> >
> > The second phase consist on using a loss function to compute the errors made in our predictions by comparing the predicted result to the actual result (ground truth). It is important to understand the concept of loss function, also called the cost function or error function, which is the **function we want to minimize** as it quantifies the difference between our predicted data and the actual data, hence being a key indicator for the accuracy of our model.
> >
> > > 
> > >
> >
> > **3) Backward propagation**
> >
> > Now the third step, which is the backward pass through the network, can be seen as computing the gradient of the loss function with respect to the parameters, which means backpropagating the error in order to **understand which of the weights are responsible for generating this error**. In order to apply the backpropagation algorithm, our activation function must be **differentiable** so that we can compute the partial derivative of the error with respect to a given weight wi, j, loss (E), node output oj, and network output netj. In other words, in order to get the gradient for each weight, we will use something called **Automatic differentiation**, which can be seen as a decomposition of a composite function derivatives provided by the chain rule. Hence, the chain rule bring us the tool for finding the influence of early layer parameters on the final loss. For this task, **Jacobian matrices** are important as we can apply the chain rule to a vector-valued function just by multiplying Jacobians. To understand those, let's call (Y) the output vector of (f). Simply, **the Jacobian matrix of f contains the partial derivatives of each element of y, with respect to each element of the input x**. In brief, even if all those theoric concepts are abstracted by easy to use programming packages, it is important to understand the building block of backpropagation.
> >
> > > 
> > >
> >
> > **4) Gradient based optimization**
> >
> > The fourth step can be seen as gradient based optimization, where an algorithm such as **stochastic gradient descent**, which is an extension of the gradient descent algorithm we saw earlier, is used to perform learning by computing the right gradient direction change and then adjusting the parameters of the model towards a reduction of the loss function. In fact, by adjusting up and down each weight slightly, we can figure out in which direction the error

changes, hence informing us if the specific weight should be adjusted higher or lower. Repeating this process guide us in a **downhill direction on the loss function curvature**, eventually reaching a minimum. Of course, this is done by a very large number of iterations where each time, we update the weights according to how much they are responsible for generating the error (given by the calculated gradient).

Over time, this process of learning the weights of the network is usually one that exhibits diminishing returns as it eventually becomes no more efficient to decrease the test error by adjusting the weights. Often, this can mean we have reached a local or sometimes a global minimum of the loss function. But in other situations, it may mean that the number of new iteration required to reduce further the error became very high or that additionnal iterations will result in overfitting the data.

Then, factors like the learning rate (deciding by how much we update the weights) or the frequency to which we update those weights can be adjusted depending on the desired learning goal. Eventually, when the whole training set is passed through the ANN, it is said that one **epoch** was made. Normally, large numbers of epoch are required to accurately train a model.

Moreover, it is important to understand that the stochastic gradient descent (SGD) is simply a modification of the standard gradient descent algorithm as SGD was designed to computes the gradient and updates the weight matrix W on **small batches of training data** and not on all the dataset. This allow the process to be much faster as the convergence towards the loss function minimum is made more efficiently.



---

2) STEPS FOR BUILDING A NEURAL NETWORK

---

1) Defining a Cost function & the Network architecture

The cost function always need to be representative of problem we are trying to solve. For example, an image classification task, the go-to is a cross-entropy loss (categorical or binary).

Here, we are talking mainly on the depth of the network, the width of each layer, and the activation function used. Deeper networks can be tempting as they provide good performance when it comes to generalization and accuracy but they are costly and hard to optimize. Hence, the ideal architecture need to be adapted to your problem and is best found via experimentation with the validation set error. The main factors you need to consider in your problem is the size of our dataset, the number of classes and their similitude.

2) Defining an Optimization algorithm & Parameter initialization strategy & Hyperparameters

This process depends on your goal, but the go-to is the traditionnal SGD or its modified version called Adam optimizer. Now we need to randomly initialise our weight matrices and bias vectors. Some use uniform and normal distributions to generate values, other prefer to use a constant initialization strategy where all weights in the neural network are initialized with a constant value (ex: 1).

Using a specific strategy, we often choose specific combinations of hyperparameter values to train our network. The two important hyperparameters are the learning rate α and the regularization term λ. The former decides by how much we update the weights at each iteration and ladder controls the amount of regularization strength we are applying. Like we saw earlier, the three common type of regularization are L1, L2, or the mix of both called Elastic Net.

3) Training, reviewing & saving the model

Like we saw, we input the observations, perform the forward-propagation pass, measure the error between the predictions and the ground-truth, back-propagate the error through the network, update the weights, and repeat. Of course, if techniques like batch learning are used, we update the weigths only after a specific batch of observations. Therefore when the whole training set is passed through the ANN, we made one epoch. We then redo more to achieve optimal results. The **total number of epochs the network should be trained on is also considered a model parameter**. Using more sophisticated training methods like training momentum or the Nesterov acceleration should also be specified in the process.

We train model until a specific performance threshold is achieved that can be measured by our metric goal. Once we are done, we discard the input data and keep only the weight matrix W and the bias vector b as our raw model.

PART 1 --- ANNs FOR SUPERVISED LEARNING

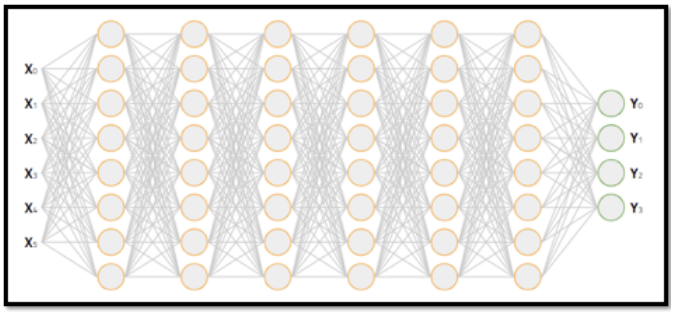
---

1) Deep Feedforward Networks

- Overview:

Deep feedfoward neural networks are the **basic category** when it comes to deep learning. Indeed, they consist of units arranged into a set of layers (input, hidden, output) that form a network. The number of layers is known as the depth, and the number of units in a layer is known as the width. All you need to remember is that in those type of network, there are no feedback connections in which outputs of the model are fed back into itself. The most common one is named: **Multilayer Perceptron (MLP)**.

MLPs are often adapted to simpler tasks like classification & regression prediction problems as their basic architecture is not adapted to map spacial or temporal complex relationships like other type of networks we will see further. Indeed, being able to process successfully patterns with sequential and multidimensional data requires some changes in the architecture of feedforward networks. The name perceptron comes from a type of algorithm made in the ealy days of AI to perform binary classification tasks. As a single perceptron is unable to modify its structure to map complex functions, they are often merged together in layers in order to learns smaller and more specific features.



- Example 1:

Below is an example of a simple **Artificial neural network (ANN) model** that is used to make a prediction on whether a specific customer of a bank is prone to quit the institution for newer financial competitors. To do this, we are using a dataset with observations that use multiple factors like the Credit Score, the Geography, or the Gender of an

> individual. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

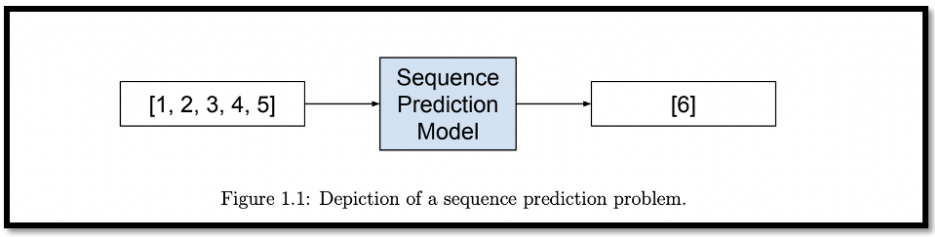
```
# Importing the Libraries
    # Numpy allows us to work with array.
    # Pandas allows us to not only import the datasets but also create the matrix of features(independent) and
    # dependent variable.
    # TensorFlow has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researcher
    # push the state-of-the-art in ML and developers easily build and deploy ML powered applications.
    # IMPORTANT (if not working): conda install tensorflow or import os or os.environ['KMP_DUPLICATE_LIB_OK']='True'
import numpy as np
import pandas as pd
import tensorflow as tf



# Data Preprocessing
    # Importing the Dataset + defining variables
dataset = pd.read_csv('Churn_Modelling.csv')
x = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values


    # Encoding Categorical Data
        # Label encoding the "Gender" column
            # Female = 0   /   Male = 1
        # One Hot Encoding the "Geoghraphy" column
            # France = 1 0 0
            # Spain = 0 0 1
            # Germany = 0 1 0
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
x[:, 2] = le.fit_transform(x[:, 2])

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthrough')
x = np.array(ct.fit_transform(x))


    # Splitting the dataset into the Training and Test Set
        # Dividing training and test set.
        # The best ratio is 80 - 20 for trainging and testing respectively.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 1)


    # Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.fit_transform(x_test)



# Model building
    # Initailizing the ANN
    # An instance of the sequential class which intializes as a sequence of layers.
ann = tf.keras.models.Sequential()

    # Adding the input layer and the First hidden layer
    # ReLU activation function selected (often choosen with MLPs)
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

    # Adding the Second hidden layer
    # ReLU activation function selected (often choosen with MLPs)
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

    # Adding the Output layer
    # Sigmoid activation function selected (Classification task)
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))



# Training the model
    # Compiling the ANN
        # Declaring the optimizer to use in order to minimize the loss function
            # Most of them are based on gradient descent
            # Adam optimizer is the one used by default
        # Declaring the loss function to minimize (binary 0/1 classifier = binary_crossentropy )
        # Declaring the metrics to report statistics on (classification = accuracy)

ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
    # Training the ANN on the Training Set
ann.fit(x_train, y_train, batch_size = 32, epochs = 100)



# Making Prediction
    # Predicting the result of the desired observation:
    # Use the ANN model to predict if the customer with the following informations will leave the bank:
        # Geography: France
        # Credit Score: 600
        # Gender: Male
        # Age: 40 years old
        # Tenure: 3 years
        # Balance: \$ 60000
        # Number of Products: 2
        # Does this customer have a credit card? Yes
        # Is this customer an Active Member: Yes
        # Estimated Salary: \$ 50000
print(ann.predict(sc.transform([[1, 0, 0, 600, 1, 40, 3, 60000, 2, 1, 1, 50000]])) > 0.5)



# Model evaluation
    # error matrix (Confusion Matrix)
    # Accuracy Score
from sklearn.metrics import confusion_matrix, accuracy_score
y1_pred = classifier.predict(x1_test)
cm1 = confusion_matrix(y1_test, y1_pred)
print(cm1)
print(accuracy_score(y1_test, y1_pred))
```

---

2) Recurrent Neural Networks (RNN)

- Overview:

> The Recurrent neural architecture is distinguishable from the traditionnal networks seen before from its ability of **sharing parameters** across different parts of a model, which create a type of sequential memory to the model. While feedforward networks have different weights across each unit, RNNs share the same weight parameter within each layer of the network. This trait make them well adapted for **Time series-based problems**.
>
> RNNs are often said to remembers the past and make decisions later according to it. This characteristic allow them to have outstanding performance when it come to processing sequential data and face **sequence prediction problems** that involves using historical sequence information to predict future values in the sequence. Example of sequences can be letters in a sentence, stock prices or any audio data.
>
> > 
> >
> > Figure 1.1: Depiction of a sequence prediction problem.
>
> The major thing about RNNs is that the output(s) are influenced by the weights AND by a **"hidden" state vector** representing the context extracted from prior sequences. In fact, this hidden state captures the relationship that neighbors might have with each other in a serial input and it keeps changing in every time step. In addition to generating the output as a function of the input and hidden state, the hidden state is self-updated in order to continue processing the next input.

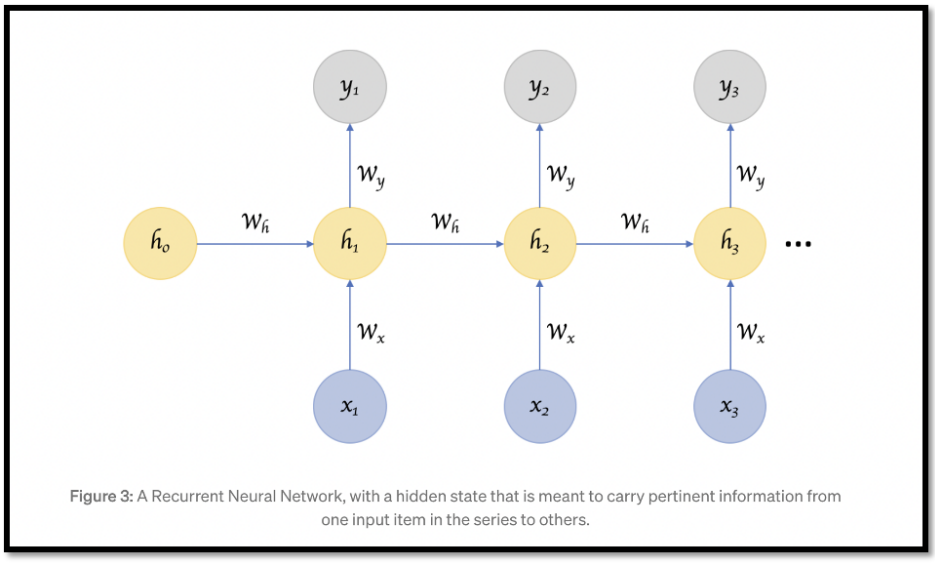


Figure 3: A Recurrent Neural Network, with a hidden state that is meant to carry pertinent information from one input item in the series to others.

The learning process is quite similar to other "vanilla" networks, but it is often used with the backpropagation Through Time (BPTT), which can be seen as the traditionnal backpropagation algorithm applied sequencial data that unroll all inputs into timesteps in order to compute the errors effectively. Other than producing one output value for each input value, there is different type of sequence prediction problems that influence the architecture needed for the RNN:

For image captionning task where we output a description sentence to an image, vector to sequence architecture (One-to-Many) are used because it is required to produce multiple output values for one input value. For producing sentiment analysis from texts, sequence to vector architecture (Many-to-One) are used because it is required to output value as a fixed length vector after receiving multiple sequences of input values. For language translation, sequence to sequence architecture (Many-to-Many) are used because it is required to produce multiple output values after receiving multiple input values. And when there is different lenghts in the inputs and ouputs sequences, a type of Encoder/decoder architecture is used, where an encoder converts the input sequence to a vector and the decoder will convert the vector back into the output sequence.

- Long Short-Term Memory (LSTM)

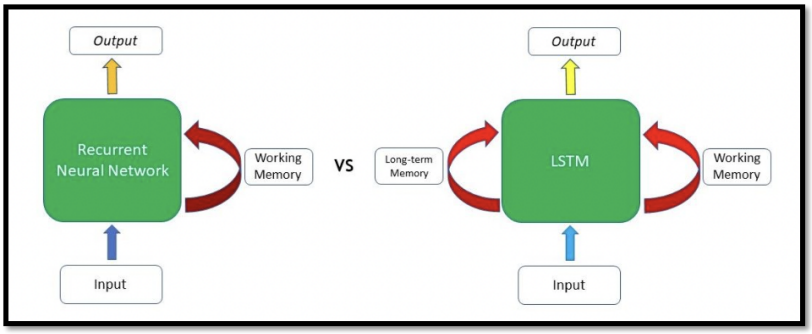
Different type of RNNs were engineered to solve different kind of tasks. The most popular are Recursive Neural Networks, Bidirectional RNNs, LSTMs... The one that we will briefly see through this course is the Long Short-Term Memory (LSTM) model, which introduce new parameters that act as gates in order to decide what to remember and what to forget.

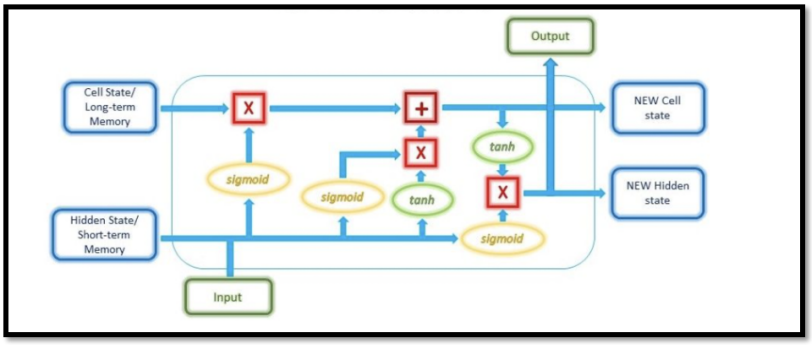
The LSTMs architecture emerged from the problematic training of large deep RNNs, where gradients were eventally vanishing or exploding over time. Like RNNs, they have recurrent connections that allow information from previous time step to stay. But what makes them special is that they are integrated with a *Long-term memory* capacity where information flows through a mechanism known as cell states. In addition of the hidden state vector, each LSTM memory cell keeps a unique cell state vector that allows the configuration of different type of weights. Similar to a computer chip, LSTM cells make decisions about what to store, when to read, write or erase via gates mechanisms. To make it simple, there is 6 main "things" that govern the information flow of a LSTM block: 3 type of data and 3 types of gates:

- Current input data to the cell
- Hidden state data: Short-term memory from the previous cell (similar to hidden states in RNNs)
- Current state data: Long-term memory.

- Input gate: With info from the current input and the short-term memory, it decide NEW info to long-term memory.

- Forget Gate: Which information from the long-term memory should be kept or discarded.
- Output Gate: Decides what to output based on current input and the 2 type of memory weights of the cell.





Hence, after this filtering computation process, two states are transferred to the next cell, which is the cell state and the hidden state. This process allow LSTM models to keep a type of long term memory, making them very attractive for task needing contextual information from earlier time steps. When it comes to the training process, the Backpropagation Through Time (BPTT) used on Recurrent Neural Networks is slightly modified for optimal performance by limiting how many time steps are required before any update to the weights. The algorithm is then named: Truncated Backpropagation Through Time (TBPTT).

- The Attention mechanism

When it comes to neural networks & specifically NLP tasks, understanding the concept of Attention is instrumental. Indeed, this idea was implemented to allow the encoder-decoder architecture to look beyond the fixed-length internal representation that it was constraint on. To make it simple, attention is used to **give higher importance or weights to specific words in the input sequence** for each word in the output sequence, hence allowing the model to focus only on relevant information. This is done by introducing an additional input to the target RNN that consist of a **context vector** that host the important information for generating the next target word. This idea was so powerfull when it was introduced that many other type of networks are now implementing it.

Of course, I could not introduce attention without the famous model called **Transformer**, which is recognized to steal all the popularity of other RNNs like the LSTM models. Indeed, by allowing parallel computation in order to reduce training time & the traditionnal drop in performances of long dependencies, sentences are now **processed as a whole** thing rather than word by word in traditionnal RNN/LSTM. If more information is desired on NN applied to NLP, I recommand reading on Self Attention, Positional embeddings and Tokenization of the input data.

- Example 2:

Below is an example of a **Long Short-Term Memory (LSTM)** model used on a sequence prediction problem where we desire to predict the future stock price of the company AMAZON. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:
```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from tensorflow import random



# Import training dataset
dataset_training = pd.read_csv('../AMZN_train.csv')
training_data = dataset_training[['Open']].values
training_data
```

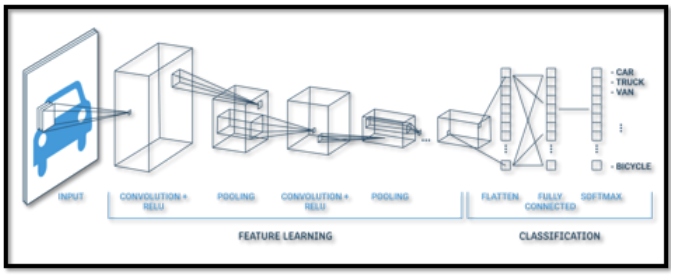
```
# Scaling the data for the input layer
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
training_data_scaled = sc.fit_transform(training_data)



# Create Data Time Stamps & Rehape the Data
X_train = []
y_train = []
for i in range(60, 1258):
    X_train.append(training_data_scaled[i-60:i, 0])
    y_train.append(training_data_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))



# Create & Compile an RNN Architecure
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

seed = 1
np.random.seed(seed)
random.set_seed(seed)

# initialize model
model = Sequential()

model.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))

# Adding a second LSTM layer
model.add(LSTM(units = 50, return_sequences = True))

# Adding a third LSTM layer
model.add(LSTM(units = 50, return_sequences = True))

# Adding a fourth LSTM layer
model.add(LSTM(units = 50))

# Adding the output layer
model.add(Dense(units = 1))

# Compiling the RNN
model.compile(optimizer = 'adam', loss = 'mean_squared_error')

# Fitting the RNN to the Training set
model.fit(X_train, y_train, epochs = 100, batch_size = 32)







# Prepare the Test Data
dataset_testing = pd.read_csv('../AMZN_test.csv')
actual_stock_price = dataset_testing[['Open']].values

# Concatenate Test & Train Datasets
total_data = pd.concat((dataset_training['Open'], dataset_testing['Open']), axis = 0)
inputs = total_data[len(total_data) - len(dataset_testing) - 60:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)
X_test = []
for i in range(60, 81):
    X_test.append(inputs[i-60:i, 0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
predicted_stock_price = model.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)



# Visualising the results
plt.plot(actual_stock_price, color = 'green', label = 'Real Amazon Stock Price',ls='--')
plt.plot(predicted_stock_price, color = 'red', label = 'Predicted Amazon Stock Price',ls='-')
plt.title('Predicted Stock Price')
plt.xlabel('Time in days')
plt.ylabel('Real Stock Price')
plt.legend()
plt.show()
```

3) CONVOLUTIONAL NEURAL NETWORKS (CNNs)

- Overview:

Among the most popular neural network architecture of the recent years, the CNN architecture is clearly amid the favorites. Indeed, by removing most of the traditionnal fullyconnected (FC) layers where neuron are connected to every output neuron in the next layer, CNNs are mostly known for using a **filtering methodology** in order to extract desired features from an input matrix and discern with very good precision spatial relationships. In other words, because each layer in a CNN applies a different set of filters in order to combine the result and feed it to the next layer in the network, this allow them to really outperform other models when it comes to **processing grid of values** such as images.

Using a technique called **Deep stacking**, where set of layers with different functions are used several time in a network, these neural networks can make very accurate predictions on the nature of high level features in a image simply by detecting the presence of low-level characteritics (edges, shapes...) and assigning them a higher level category. In order to fully grasp their architecture, we need to look down at the composition of these neural networks by briefly **reviewing the function of their main layers**:



- Convolutional layer:

The convolution layer can be seen as the layer who is responsible for applying specific set of filters K to detect desired features. By moving around a filter around the input grid, we output a filtered image with values representing how well that feature is represented at a specific position. Actually, the term "convolution" just mean the repeated application of these filters in order to produce a new "image".

- Activation layer:

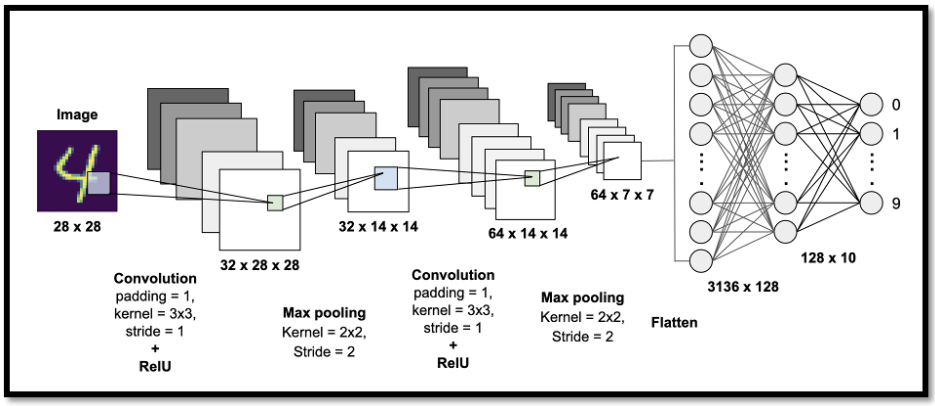
Often found just after a convolutionnal layer, the activation layer role is just to assess the optimal degree related to the neuron spike. By taking as inputs the filtered image and applying an activation function (often RELU), it contribute in inhibiting the neurons without importance by converting negative value to 0, hence adding efficiency to the network. No parameters or weights are learned inside these layers.

- Pooling layer:

The pooling layer is responsible for shrinking the matrix (or image) in order to analyse the presence of features at different scales. Hence, the role of the most common type of pooling layer named Max POOL layer can be resumed as: reducing spatial size.

- Fully-connected layer:

The FC layers are typically the final layers of a CNNs and often, a single one may be used. The nodes in those layers are fully-connected to all activations in the previous layer, which can now be used to apply a softmax classifier to output specific classification labels.



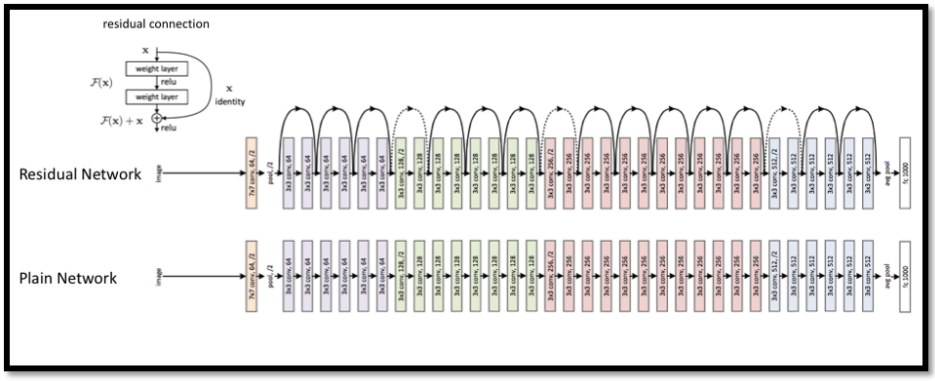
Additionally, other less common type of layers are used in CNNs. For example, normalization layers make use of **batch-normalization techniques** in order to reduce and stabilize the training of the neural networks through data scaling. Also, layers called **dropout layers** are often used as a regularization method in order to add redundancy in a network for better generalization capabilities. Finally, if your decide to handle deeper networks, applying **zero-padding** is often recommanded (modify border of input matrix before conv), where the output size relative to the input is kept constant through some layers in order to avoid the fast reduction of the spatial dimension volume. Without it, the complete training of the network will often not be possible.

- CNNs vs Resnets:

Despite their success in solving tasks belived to be impossible before, one issue emerged when deeper convolutional neural networks were build in order to add additionnal feature extraction capacity. Indeed, problems like vanishing gradients, higher errors rates or degradation in prediction accuracy were becoming a major challenge that needed to be resolved.

This is where the recent introduction of Residual neural network (Resnets) came and solved this problem by letting information skip layers and flow more easily through network layers. In a very generalized way, these networks introduce a concept named residual learning blocks were identity mapping, the concept of refeeding the input of a layer as an output to a next layer, allow the learning to continue even as the depth of a networks grows, which translate in better learning capabilities and faster convergence.

This type of variant will not be seen in detail for this course but it is important to consider the arrival of residual neural networks in the field of Deep Learning as they are progressively choosen for tasks requiring deeper networks training. Because the field of Machine Learning is constantly evolving, it is important to follow the new developements and reseach as the most performing models are constantly changing. For example, the simple Multilayer perceptron (MLP) model presented earlier was recently (June 2021) proven to be able to match the best-performing architectures for image classification like CNNs if the model was slightly modified to process images across patches rather than individual pixels in order to improve feature learning capabalities. (name: MLP-Mixer)



- Example 3:

Below is an example of a **Convolutional Neural Network** trained on classifying images based on their respective binary class: dog or cat. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

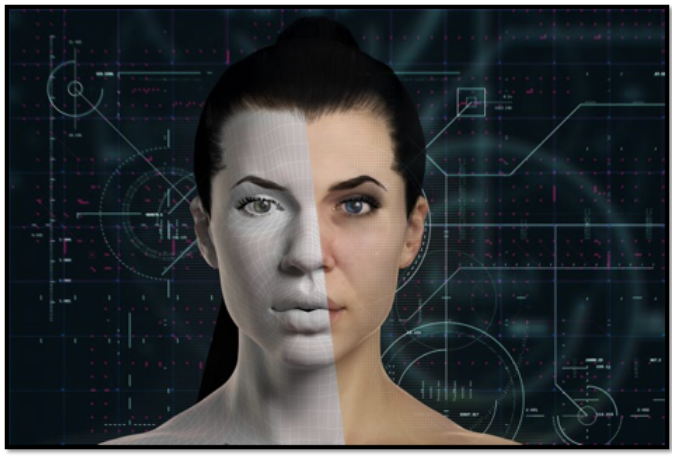
In [ ]:
```
# Importing Libraries
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
```

```
# Data Preprocessing
    # Preprocessing the Training Set
        # Transformation is needed, because to avoid over fitting
        # Transformation are some geometrical transformation or some zoom or some rotation on image.
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
training_set = train_datagen.flow_from_directory('D:\\Programming Language\\Machine Learning\\dataset\\training_set',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')



    # Preprocessing the Test Set
        # Transformation is needed, because to avoid over fitting
        # Transformation are some geometrical transformation or some zoom or some rotation on image.
test_datagen = ImageDataGenerator(rescale = 1./255)
test_set = test_datagen.flow_from_directory('D:\\Programming Language\\Machine Learning\\dataset\\test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')



# Model building
    # Initialising the CNN
cnn = tf.keras.models.Sequential()

    # Convolution
cnn.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 3, activation = 'relu', input_shape = [64, 64, 3]))
    # Pooling
cnn.add(tf.keras.layers.MaxPool2D(pool_size = 2, strides = 2))

    # Second stack
    # Convolution
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'))
    # Pooling
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))

    # Flattening
cnn.add(tf.keras.layers.Flatten())
    # Full Connection
cnn.add(tf.keras.layers.Dense(units= 128, activation = 'relu'))
    # Output Layer
cnn.add(tf.keras.layers.Dense(units = 1, activation = 'sigmoid'))




# Model training
    # Compiling CNN
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
    # Training the CNN on the Training Set
    # Evaluating the CNN on the Test Set
cnn.fit(x = training_set, validation_data = test_set, epochs = 25)




# Prediction
    # Import Libraries and Packages (Numpy for array)
import numpy as np
from keras.preprocessing import image

test_image = image.load_img('D:/Programming Language/Machine Learning/dataset/single_prediction/cat_or_dog_1.jpg',
                            target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = cnn.predict(test_image)
training_set.class_indices
if result [0][0] == 1:
    prediction = 'Dog'
else:
    prediction = "Cat"

print(prediction)
```

```
# Example of ResNet50 prediction: https://github.com/PacktWorkshops/The-Deep-Learning-with-Keras-Workshop/blob/master/Cha
```

PART 2 --- ANNs FOR UNSUPERVISED LEARNING

- Introduction:

> Now that we have seen the main artificial neural networks used in supervised learning tasks, I propose that we look in brief details some of the ANNs used in unsupervised learning tasks, which is actually an **active field of research** due to the dataset labeling problem that limit the traditionnal neural networks capabalities.
>
> One of the main characteristics of deep unsupervised ANNs is that they are able to perform strong **representation learning** by finding distinct pattern in large dataset via a set of parameters that is significantly smaller than the number of examples available. Indeed, those learning algorithms are using many methods to discover the main underlying factors governing data. For example, the smoothness of a function, which can be seen as its degree of continuous derivatives at every point, can be a key factor for good prediction accuracy. Other properties like the number of explanatory factors, sparsity of a function, or simply the dependencies between factors can all be key to making good unsupervised algorithms capable of astonishing representation learning capacities.
>
> An other task that deep unsupervised learning methods are really good at is to learn a **generative model**. Indeed, by learning the structure of the underlying probability distribution of a dataset, those large-scale probabilistic models are able to generate new samples that can easily be confused with real data. Unlike the discriminative models that we have seen earlier in deep supervised learning, these models make use of probability estimates and likelihood in order to differentiate classes or generating new data points. Actually, those models are finding new applications everyday in the field of imaging, communication channels, arts, or applied scientific fields.
>
> > > 

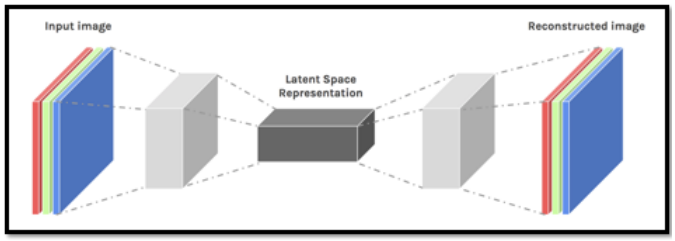
---

1) AUTOENCODERS

- Overview:

> Autoencoders are a class of neural network known to be very good at learning the main useful properties of a dataset, making them very adapted for dimensionality reduction and feature discovery tasks. In other words, they are able to reduce the number of features and keeping only the main parts of a data structure through a reduced representation space.
>
> They are composed of two main parts: an encoder and a decoder. The encoder take as inputs the raw data and is often constrain to learn an undercomplete representation of the data by producing outputs whose code dimension is less than the input dimension, hence producing a type of projection in lower dimensions. In order to train the encoder, a decoder is needed that take as inputs the code vector given by the encoder and try to reconstruct the initial data as its output. Therefore, we are looking for the pair that keeps the maximum of information when encoding and, so, has the minimum of reconstruction error when decoding.
>
> Through an iterative optimisation process, the encoder & decoder networks are guided to learn the most efficient scheme through the minimization of a loss function penalizing the network for being dissimilar from the input data X. After training, the decoder is

discarded and the output from the converging phase of the encoder known as the "bottleneck" is used directly as the reduced dimensionality of the input.



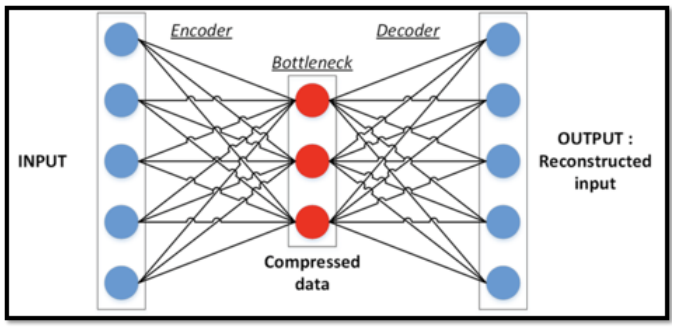
- More details:

The training phase is similar to other traditionnal neural networks where the encoded-decoded output from the raw data will lead to a backpropagation of the error through the architecture in order to update the weights via the gradient descent mechanism. The loss function used depends on the nature of the input values: If they are between 0 and 1, a crossentropy function is used. Otherwise, the typical mean squared error methodology is applied.

Before implementing a model, it is important to decide the Code size desired (nodes in the middle layer) as the smaller it is, the more your degree of compression will be. Other hyperparameters like the number of layers and the number of nodes per layer are key for getting the most of the model, but the general rule is that the number of nodes per layer should decreases with each subsequent layer of the encoder and increases back in the decoder. The number of different autoencoders architecture is increasing each year, but here are the main categories to remember:

1) Vanilla autoencoders:

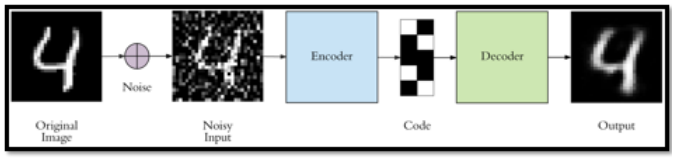
Simplest encoder with a single or multiple hidden layers trained to output the principal subspace of the training data via a loss function penalizing the decoder for losing information in the reconstruction process. Hence, the decoder is kept shallow and the code size small (named "undercomplete"), forcing it to learn useful features. If linear, the result is often similar to a PCA algorithm. If non-linear as it is the case with deep autoencoders, the network possess higher ability to generalize and extract features.



2) Regularized autoencoders:

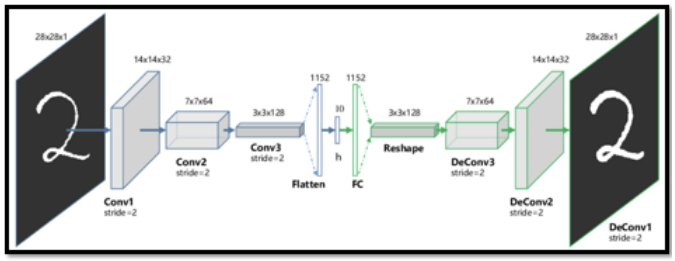
Category of autoencoders using a different loss function that prioritize other properties of the reconstructed data. These are often said to be "overcomplete" with a larger capacity to learn the data distribution. The most common one are the Sparse autoencoder and the Denoising Autoencoders:

> **- Sparse autoencoder:** Type of regularized autoencoder whose loss function was bonified with a sparsity penalty over the traditionnal reconstruction error. Because they are very good at discovering interesting structure in the data, these are often used to reduce the size dimension of a dataset in order to feed it to a classifier network. Because the design of the loss function make the network uses only a fraction of its nodes simultaneously, this method works even if the code size is large.
>
> **- Denoising Autoencoders:** Type of regularized autoencoder that is designed to receive corrupted data points as input and is trained to predict the original data as its output. By adding random Gaussian noise to its inputs and forcing it to recover the original noise-free data, this type of autoencoder is able to learn a great representation of the data even if the code layer is not kept small. Hence, those type of networks are very adapted for tasks requiring denoising capabalities.



3) Convolutional autoencoder:

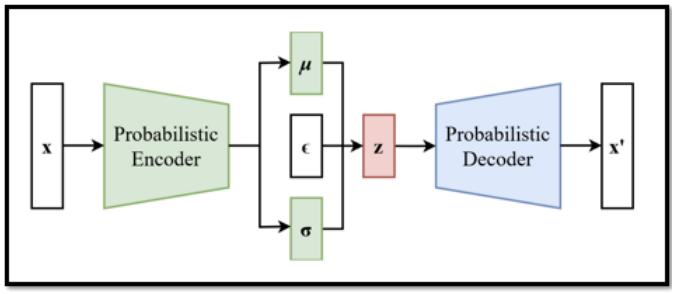
When applied to image related tasks, the traditionnal one dimensional vectors are replaced by three-dimensional vectors in order to obtain a latent representation space of images. Hence, because we are mixing convolutional neural networks as encoders and decoders, it is said that we are using a convolutional autoencoder.



4) Variational autoencoder:

Variational autoencoders (VAE) are a special case of autoencoder as they learns the parameters of the probability distribution modeling a specific dataset. Because it is possible to generate new input data samples simply by sampling points from a distribution, they are often called a generative model. But since their ability to generate samples of good quality is very limited, GANs (see next section) are prefered for this task.

By mapping the inputs to a distribution rather than a fixed vector, you get two separate vectors as a bottleneck, one representing the mean of your distribution and the other one representing the standard deviation. Thus, VAE can be seen as a probabillstic spin on regular autoencoders. The training phase may be seen as sampling a vector from the distribution in order to feed it to the decoder, which a loss function made of the reconstruction error is combined with the KL divergence, which can be seen as a way to quantify the difference between two probability distributions: the learned latent one vs the prior distribution.



- Main applications:

- Dimensionality reduction
- Information retrieval
- Image Denoising
- Image Compression
- Image segmentation

- Example 7:

Below is an example of a **Convolutional Autoencoder** trained to perform an image denoising task. The autoencoder is trained to map noisy digits images from an open-source dataset (MNIST) to clean digits images. The noisy images

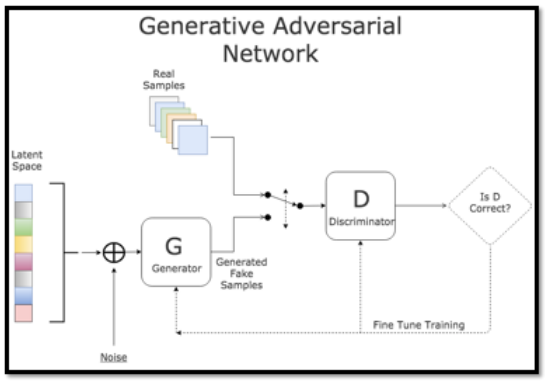
> are generated by applying a gaussian noise matrix on them. Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

```
# Import packages
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt


# Data importation of MNIST
(x_train, _), (x_test, _) = mnist.load_data()


# Data pre-processing
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))



# Noise addition
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)




# Noisy data visualisation
n = 10
plt.figure(figsize=(20, 2))
for i in range(1, n + 1):
    ax = plt.subplot(1, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()



# Model building
# Deeper layers = better noise removal capabalities

input_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# At this point the representation is (7, 7, 32)

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')




# Model training
# 100 epochs
autoencoder.fit(x_train_noisy, x_train,
                epochs=100,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb', histogram_freq=0, write_graph=False)])



# link: https://blog.keras.io/building-autoencoders-in-keras.html
# Other option: https://github.com/sourcecode369/deep-learning/blob/master/autoencoder/Convolutional_Autoencoder_Solution
```

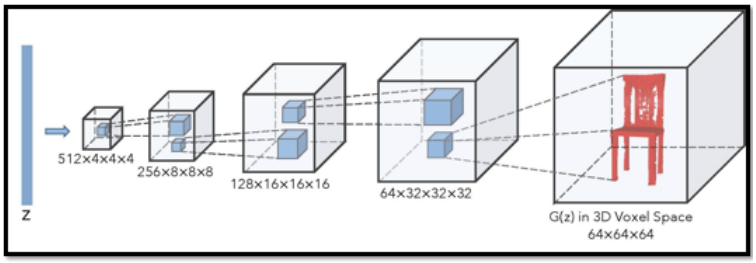
2) GENERATIVE ADVERSARIAL NETWORKS (GANs)

- Overview:

> We introduced above the Variational autoencoder (VAE), a type of autoencoder with generative modeling capabalities. In this section, we will introduce another type of generative model build on a different architecture that is often seen more difficult to train but much more powerful when it comes to performance. These are named **Generative Adversarial Networks (GANs)**.
>
> They were introduced in 2014 by Ian Goodfellow and really got the attention of the world in the recent years when they were shown to be able to generate realistic "Deep fakes" and realistic pictures of individuals that never existed. In essence, the model is based on the idea of fooling a feedforward classifier by creating artificial but plausible examples from the input problem domain from which the model is trained on.
>
> Like VAEs, GANs are based on the idea of **differentiable generator networks** where samples are generated via the mechanism where a model transforms samples of some latent variables z to samples x using a differentiable function that is represented by a neural network. But unlike VAEs where the generator network is paired with an inference net, GANs generator networks are **paired with a discriminator network** that act as an adversary. Indeed, the generator model is trained to generate new examples, and the discriminator model is classifying those examples as either real or fake. During training, both networks are trained together until the discriminator model is fooled around half the time, meaning that the generator model is able to produce realistic examples.
>
> Over the years, an iteration of the initial model named **Deep Convolutional Generative Adversarial Networks (DCGAN)** became the version of choice as it was more stable and because the model was typically used with images, using CNNs as the generator and the discriminator was an obvious choice for better performance. This is the one we will focus on in this lab.
>
> > 
> >

- Details:

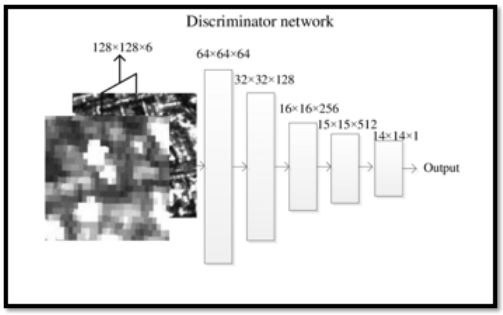
> GANs can be seen as a pair of network where the generator is responsible for mapping values to **produce samples** from the distribution and the discriminator is responsible for **classifying those as real or fake**. The game can then be resumed as the **Generator trying to fool the discriminator**, while the **disciminator itself is trying to not get fooled**. In other words, they both try to optimize the opposite cost functions. Hence, even if GANs are a type of unsupervised learning algorithms because they learn the underlying structure of the true distribution without labels, it is possible to view the training phase of the two models as a supervised learning problem.
>
> **- The Generator:** >
>
> > The neural network of the generator is composed of multiple hidden layer that will **capture the representation of the underlying data structure** in order to accomplish its goal of generating new plausible examples from the problem domain. The underlying distribution can be seen as the latent space with a compressed representation of the main features related to the data. In other words, this **vector space is comprised of latent variables** that act as a projection of the high-level concepts observed in the raw data.
> >
> > The input taken from the generator is fixed-length random vector drawn randomly from a Gaussian distribution and the output is a sample in the domain (ex: image). When the statistical latent space of the data is learned, the generator used two types of layers to create a realistic sample: an **upsample layer** and a **transpose convolutional layer**. The former is a simple layer with no weights that is used to double the dimensions of the input by repeating the rows and columns while the ladder is used to perform an inverse convolution operation by learning how to fill in details. The only requirement is to specify the number & size of the filters and specifying the strides, which refers to the manner a filter scan across an input.



- The Discriminator: >

The neural network of the discriminator is simply trained to accurately takes an example from the problem domain as input and predicts the associated **binary class label (real or generated)**. The real example are from the training dataset while the generated ones are from the generator model. After the training process, the discriminator model is removed as we are interested in using the generator only.

Hence, the discriminator simply output a probability based on a strategy using **density ratio estimation (DRE) methods**. In fact, by comparing the ratio between the density of the real dataset and the density of the generator model, many divergences (ex: Jenson Shannon & KL) between the data and the model probability distribution can be found and used to train the maximum likelihood estimation.

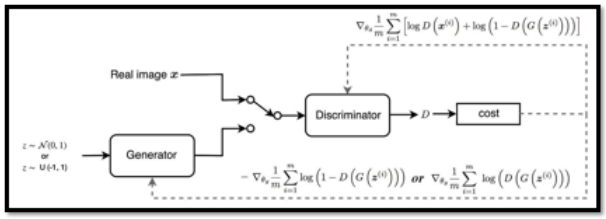


- Training:

The training algorithm involves parallel training of both the discriminator and the generator through minibatch SGD, a variant where two minibatches are sampled. One is from the real dataset and the other is from the model latent space. After the batch of real data is sampled, it is provided to the discriminator, where it is tasked with making predictions for the nature of those samples. After that, the weights of the discriminator are updated proportionally to the predictions accuracy. Now, a batch of random points is sampled from the latent space and provided as input to the generator, which will create artifical samples that will be send to the discriminator for classification. Hence, after one training iteration, the discriminator is updated with two batches of samples while the generator is only updated with a single batch.

The learning process is often said to follow a minimax objective function where to two players are affronting each other. The discriminator loss function is designed to maximize the log of predicted probability of real images and the log of the inverted probability of fake images, averaged over each minibatch. Thus, Gradient Ascent optimization is used and the same standard cross-entropy cost used on a binary classifier can be implemented.

Switching on to the generator, its loss function is often different depending on the optimization and learning rate required. It can be represented as the minimization of the log related to the inverted probability of the discriminator's prediction of artificial images (Gradient Descend) or as a maximization of the log of the discriminator's predicted probability for fake images (Gradient Ascent). In simple words, the former is minimizing the chances that the discriminator is right while the ladder is maximizing the chances that the discriminator is wrong.



- Challenges:

> Despite showing impressive results, GANs are often credited for having learning stabilization problems as both the generator and the discriminator model are trained simultaneously, which often do not guarantee an equilibrium to the gradient optimization processes. Indeed, this situation can be seen as a dynamical system where the change in parameters for one model modify the nature of the problem at each time step. Hence, training competing neural networks at the same time can often lead to a lack of convergence.
>
> This is why the model architecture need to be **carefully selected and the hyperparameters well optimized** for the desired task. In addition to using more modern architecture like the DCGAN, changes in the pooling layers, in the activation functions, and the use of techniques like batch normalization is often recommanded for optimal performance. Otherwise, new type of GANs are emerging every year and are made more adapted to specific tasks. Example: Conditional GANs.

- Applications:

> - Create photorealistic images of people who never existed (https://thispersondoesnotexist.com/)
> - Art generation
> - Super-resolution enhancing (Upscaling) (DLSS from Nvidia)
> - Image-to-Image Translation
> - Add colours to images

- Example 8:

> Below is an example of a **Generative Adversarial Network** (GAN) trained to perform a task related to generating new artificial samples of handdigit pictures. The model is trained on the open-source dataset (MNIST). Again, you can visualize each step with the linked commentaries. Below is the related code that you need to execute:

In [ ]:

```
# Import packages
import pandas as pd
import numpy as np
import tensorflow
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook
%matplotlib inline



# Import Dataset
# Data visualisation
from tensorflow.keras.datasets import fashion_mnist, mnist

(trainX, trainY), (testX, testY) = mnist.load_data()

print('Training data shapes: X=%s, y=%s' % (trainX.shape, trainY.shape))
print('Testing data shapes: X=%s, y=%s' % (testX.shape, testY.shape))

for k in range(9):
    plt.figure(figsize=(7, 7))
    for j in range(9):
        i = np.random.randint(0, 10000)
        plt.subplot(990 + 1 + j)
        plt.imshow(trainX[i], cmap='gray_r')
        plt.axis('off')
        #plt.title(trainY[i])
    plt.show()



# Data pre-processing
trainX = [image/255.0 for image in trainX]
testX = [image/255.0 for image in testX]

trainX = np.reshape(trainX, (60000, 28, 28, 1))
testX = np.reshape(testX, (10000, 28, 28, 1))

print (trainX.shape, testX.shape, trainY.shape, testY.shape)




# Generator network
random_input = tensorflow.keras.layers.Input(shape = 50)
```

```
x = tensorflow.keras.layers.Dense(1200, activation='relu')(random_input)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(1000, activation='relu')(x)
x = tensorflow.keras.layers.BatchNormalization(momentum=0.8)(x)
x = tensorflow.keras.layers.Dense(28*28)(x)
x = tensorflow.keras.layers.Reshape((28, 28, 1))(x)

generated_image = tensorflow.keras.layers.Activation('sigmoid')(x)
generator_network = tensorflow.keras.models.Model(inputs=random_input, outputs=generated_image)
generator_network.summary()
```

```
# Discriminator network
image_input = tensorflow.keras.layers.Input(shape=(28, 28, 1))

x = tensorflow.keras.layers.Flatten()(image_input)
x = tensorflow.keras.layers.Dense(256, activation='relu')(x)
x = tensorflow.keras.layers.Dropout(0.3)(x)
x = tensorflow.keras.layers.Dense(128, activation='relu')(x)
x = tensorflow.keras.layers.Dropout(0.5)(x)
x = tensorflow.keras.layers.Dense(1)(x)

real_vs_fake_output = tensorflow.keras.layers.Activation('sigmoid')(x)
discriminator_network = tensorflow.keras.models.Model(inputs=image_input, outputs=real_vs_fake_output)
discriminator_network.summary()

adam_optimizer = tensorflow.keras.optimizers.Adam(learning_rate=0.00005, beta_1=0.5)
discriminator_network.compile(loss='binary_crossentropy', optimizer=adam_optimizer, metrics=['accuracy'])
```

```
# GAN model (combined model)
discriminator_network.trainable=False

g_output = generator_network(random_input)
d_output = discriminator_network(g_output)

gan_model = tensorflow.keras.models.Model(random_input, d_output)
gan_model.summary()

gan_model.compile(loss='binary_crossentropy', optimizer=adam_optimizer)
```

```
# Defining the training procedure
# Indices of Zero Images
indices = [i for i in range(len(trainX))]

def get_random_noise(batch_size, noise_size):
    random_values = np.random.randn(batch_size*noise_size)
    random_noise_batch = np.reshape(random_values, (batch_size, noise_size))
    return random_noise_batch

def get_fake_samples(generator_network, batch_size, noise_size):
    random_noise_batch = get_random_noise(batch_size, noise_size)
    fake_samples = generator_network.predict_on_batch(random_noise_batch)
    return fake_samples

def get_real_samples(batch_size):
    random_indices = np.random.choice(indices, size=batch_size)
    real_images = trainX[np.array(random_indices),:]
    return real_images

def show_generator_results(generator_network):
    for k in range(9):
        plt.figure(figsize=(7, 7))
        fake_samples = get_fake_samples(generator_network, 9, noise_size)
        for j in range(9):
            plt.subplot(990 + 1 + j)
            plt.imshow(fake_samples[j,:,:,-1], cmap='gray_r')
            plt.axis('off')
            #plt.title(trainY[i])
        plt.show()
    return
```

```
# Model training
```

```
epochs = 200
batch_size = 100
steps = 500
noise_size = 50

losses_d = []
losses_g = []

for i in range(0, epochs):
    if (i%10 == 0):
        show_generator_results(generator_network)
    for j in range(steps):
        fake_samples = get_fake_samples(generator_network, batch_size//2, noise_size)
        real_samples = get_real_samples(batch_size=batch_size//2)

        fake_y = np.zeros((batch_size//2, 1))
        real_y = np.ones((batch_size//2, 1))

        input_batch = np.vstack((fake_samples, real_samples))
        output_labels = np.vstack((fake_y, real_y))

        # Updating Discriminator weights
        discriminator_network.trainable=True
        loss_d = discriminator_network.train_on_batch(input_batch, output_labels)

        gan_input = get_random_noise(batch_size, noise_size)

        # Make the Discriminator belive that these are real samples and calculate loss to train the generator
        gan_output = np.ones((batch_size))

        # Updating Generator weights
        discriminator_network.trainable=False
        loss_g = gan_model.train_on_batch(gan_input, gan_output)

        losses_d.append(loss_d[0])
        losses_g.append(loss_g)

        if j%50 == 0:
            print ("Epoch:%.0f, Step:%.0f, D-Loss:%.3f, D-Acc:%.3f, G-Loss:%.3f"%(i,j,loss_d[0],loss_d[1]*100,loss_g))




# Model testing
# Generating Unlimited samples
for i in range(5):
    show_generator_results(generator_network)


# Model performance review
steps = [i for i in range(len(losses_d))]
plt.figure(figsize=(10, 6))
plt.plot(losses_d[:5000])
plt.plot(losses_g[:5000])
plt.xlabel('Steps')
plt.ylabel('Loss Value')
plt.title("GAN: Loss Trends")
plt.legend(['Discriminator Loss', 'Generator Loss'])
plt.show()



# link: https://github.com/kartikgill/TF2-Keras-GAN-Notebooks/blob/main/gan/Generative-Adversarial-Network.ipynb
```

PART 3 --- PRACTICAL CASE STUDIES

---

1) Impressive models info & reading

- Top rankers in image recognition:

> The ranking is determined by the error margin that different models obtained on the famous ImageNet dataset that contains 14,197,122 annotated images. The current best models are achieving around 90% accuracy. Of course, their parameters are in the range of 300M to 14700M parameters, which is very deep for a network. I recommand you to go read some papers about the architecture of those models, it is very interesting.
>
> - Currently number 1: Transformer models
> - Other top rankers: EfficientNet
> - Papers list: https://paperswithcode.com/sota/image-classification-on-imagenet

- Other impressive models:

> 1) GPT-3
>
> > -175B parameters
> >
> > -96 transformer decoder layers (1.8B parameter / layer)
> >
> > -Convert word to vector
> >
> > -Compute prediction (through layer of transformer decoders)
> >
> > -Convert resulting vector to word
> >
> > -2048 tokens wide ("context window")
>
> 2) OpenAI Codex: Translates natural language to code (https://openai.com/blog/openai-codex/)
>
> > -Transformer neural network
> >
> > -Large language model that produce code from natural language docstrings
> >
> > -Based on GPT-3
> >
> > -Generating standalone Python functions from docstrings
> >
> > -Models are trained on a large fraction of GitHub (final dataset totaled 159 GB)
> >
> > -Testing: HumanEval dataset (set of 164 handwritten programming problems with unit tests)
> >
> > -Training and validation process through automated unit tests
> >
> > -How it solve problems: Many samples are generated from the model and the one who passes all unit tests is selected
> >
> > -Within 100 samples, Codex-S can produce at least one correct function for 77.5% of the problems.

Section not finished

- Student section:

Hypothesis

Dataset

In [ ]:

Model coding

In [ ]:

In [ ]:

In [ ]:

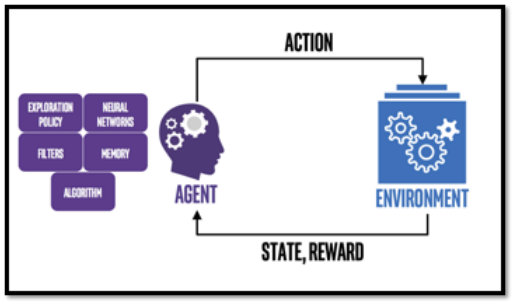
---------- VIDEOS TO WATCH ----------

> - CNN: https://www.youtube.com/watch?v=FmpDIaiMIeA&list=PLIrbp3VWo-GIDEXNG9zZrW5reQPmPnXwk&index=21&ab_channel=BrandonRohrer
> - Backprop: https://www.youtube.com/watch?v=i94OvYb6noo&ab_channel=AndrejKarpathy

---

Lab 7 -- Deep Reinforcement learning

- Introduction on subject:

> The last laboratory of this course can be seen as an introduction to the field of deep reinforcement learning, a subject that **merges multiple disciplines** like machine learning modeling and **optimal control theory** together.
>
> In a very brief way, the RL field is often concerned about **modeling the behaviours of an agent**, hence mapping situations to actions in order to maximize some specific reward signal. Indeed, by mixing trial-and-error search with delayed reward maximization, an agent can potentially capture the most important aspects of a problem it is facing and solve it over time. For that, agents in a reinforcement learning problem have goals, can interact with their environnement, and choose which actions to take in order to maximize the total reward it receives in the long run.
>
> > 
> >
>
> Unlike traditionnal supervised learning problem where a knowledgable external "supervisor" is present to guide the model, RL can be understood as an interactive problem where an agent must be able to **learn from its own experience**. In fact, the same tools used for approximating complex non-linear functions (neural networks) and their training methodologies (SGD & backpropagation) are used to learn data representation in Deep reinforcement learning, but those are nonetheless applied in a complete different way. A related recurring theme in the field is the famous trade-off between exploration and exploitation, where an agent is trying to decide whether he should prefer known actions with high reward or explore uncharted territory to potentially find newer actions with higher reward than before.
>
> This type of learning is often best associated with the actual human process of learning and decision making in a complex environnement, hence making it the most exciting according to me. Applications can range from creating powerfull autonomous agent for video games, strategy-based games, robotics control, self driving vehicles, and more.

- List of methods presented in this laboratory:

> - Monte-Carlo methods
> - Temporal-Difference methods
> - Policy-Gradient methods
> - Actor-Critic methods

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

---

1) FUNDAMENTALS OF REINFORCEMENT LEARNING

- Basic terminologies:

> To make it simple, when you hear solving a reinforcement learning task, which is often the goal of RL, it refers to finding an optimal policy ("rules guiding the behaviour") for the agent that will achieves the maximum amount of reward over the long run. The interaction between the agent and the environment can be modeled as follow: at each sequence of discrete time steps, the agent receives a representation of the environment he is currently in, which is called the state S. In order to progress and obtain rewards, it needs to select a specific action to take from a whole set of different ones in order to evolve in a different state one time step later. Then, as a consequence of the action taken, the agent will receives a numerical reward and find itself in a new state. To make sure everything is clear, let us define certain concepts:

- Agent: Entity with a goal trying to learn the best way to achieve it

- Environnement: The "things" our agent interacts with

- State: Obervations of the agent at a specific time in a specific environnement.

- Action: Input applied by the agent according to its policy that will lead him in a different state

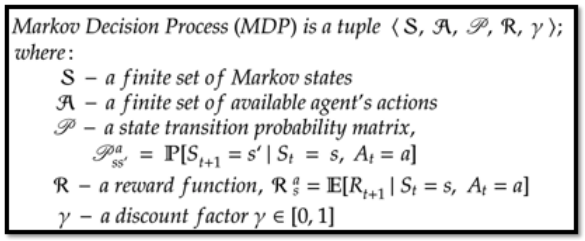
- Policy: Laws governing the behaviours of the agent. Or given a state, here are the chances of selecting X action.

- Reward: Feedback from the environnement to the agent at a specific state.

- Trajectory: Sequence of states, actions, and rewards from an agent.

- Markov decision process (MDP)

Before explaining a building block of RL modeling which are Markov decision processes, one need to understand their key proprety: **the Markov proprety**. Simply, an environment with this proprety can be understood as providing to the agent a state signal that succeeds in retaining all past relevant information necessary in order to predict the next state & expected reward as a function of the current state and action. In other words, current decisions and values are only assumed to be a function of the current state, which need to contain just enough information for the agent. In a more mathematical framework, we see a Markov chain as a rule where the next state St+1 is only dependent on the current state St, with the probability of a state jumping to the next state described by a state transition matrix P.



Now, a reinforcement learning problem that is said to satisfies the Markov property is called a Markov decision process (MDP), which is best understood as a 5-tuple (S, A, P, R, γ) framework composed of a set of states S, a set of actions A, a transition kernel P, a reward function R, and a discount factor γ. The use of MDP is fundamental when modeling decisions that can have both probabilistic and deterministic rewards values as with it, it is possible to know the optimal value functions if the MDP is solved. Therefore, an MDP can be defined as an environment in which all possible states are "Markov".

In case of a finite deterministic MDP, given any state and action, we can compute the transition probabilities and the expected rewards of our agent and use an optimal planning methodology. Otherwise, the reward function R and the transition function P(s'|s, a) are often unknown and impose trials and errors processes to the agent in order to learn them by interacting with the environment and the respective rewards obtained.

Now that we have defined what is an MDP modeling framework, we can really understand what is a policy: in simple words, a policy π can be seen as a distribution over actions given some states, which indeed can be resumed to the single word: behaviour (of the agent). Because MDP policies are Markovian and depend only on the actual state, they are said to be time-independent, or stationary. Depending on the problem, our policy can be either deterministic or stochastic which gives us the proability of taking an action given the current state under the certain policy.

- Finding the Optimal Policy:

The task of finding the optimal policy in an MDP problem can be seen as finding and computing the optimal state-value and action-value functions for our agent, which are functions derived through a maximization process of our initial value functions. To make it clear, I will start by defining what are those value functions and how they evolve to become the optimal Value function we need, but this will simply be a very brief overview of the main aspects when it comes to solving RL problems and you should definetely go seek more information about the different equations related to this process (link below):

1) State-value function

- v(s) = E [Gt| St = s]
- Expected return starting from state s, and then following policy π
- Decomposition into two parts: immediate reward & discounted rewards (Bellman Expectation equation)

- Optimal state-value function: maximum value function over all policies
- To find it, we use the Bellman Optimality Equation
- Estimates the value of a state by computing the expected rewards that the state can generate.
- $v*(s) = max\pi\ v\pi(s)$

2) Action-value function

- $q\pi(s, a) = E\pi\ [Gt \mid St = s, At = a]$
- Epected return starting from state s, taking action a, and then following policy π.
- Decomposition into two parts: immediate reward & discounted rewards (Bellman Expectation equation)
- Optimal action-value function: maximum action-value function over all policies
- To find it, we use the Bellman Optimality Equation
- Optimal state value alone doesn't tell the agent which action to take in each state
- $q*(s, a) = max\pi\ q\pi(s, a)$
- Also called the Q value function

Solving those Optimal value functions through the Bellman Optimality Equations, is a non trivial task as it is non-linear with often no closed form solution, which requires iterative solution methods (ex: Value Iteration, Policy Iteration, Q-learning or Sarsa). Since many problem possess a huge number of possible states, it would take too many ressources and time to solve the Bellman equation for V (or for finding Q). Hence, in reinforcement learning, we typically target optimal approximative solutions of the Bellman optimality equation.

This can be explained by the fact that if we knew the entire environment, hence our reward function and transition probability function, we could simply solve for the optimal action-value and state-value functions via Dynamic Programming techniques. But the lack of information about the environnement provide no closed form solution in getting optimal action-value and state-value functions, therefore the need of iterative approximations.

- link: http://www.incompleteideas.net/book/ebook/node27.html
- link: https://www.deeplearningwizard.com/deep_learning/deep_reinforcement_learning_pytorch/bellman_mdp/
- link: https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

---

- Exploration vs Exploitation tradeoff:

When it comes to modeling a problem, deciding a balance between the exploration of new states and the exploitation of known states is always challenging. By exploitation, we mean maximizing the existing knowledge of the agent to obtain rewards, hence following a greedy policy. At the other hand, exploration means increasing the existing knowledge of the agent by choosing new actions and interacting with the environment. To find a good balance, tuning the hyperparameter Epsilon Greedy, which represents the probability for an agent to choose a random action and not following the actual policy, is critical.

- Finite MDPs methods: Dynamical Programming

- Simplify complex problems by breaking them up in sub-problems and solving them recursively
- Algorithms for computing optimal policies given a perfect model of the environment as a MDP
- Assumption of a perfect model (known transition probabilities + rewards) make them non-useful for most of the cases
- Often qualified of computational expensing as it operate over the entire state set of the MDP
- Policy evaluation: Iterative process of computing the value functions for a given policy.
- Policy improvement: Computation of an improved policy given the value function for that policy.
- 2 algorithms for convergence towards optimal policie & value function of a finite MDP with complete knowledge
- *Policy Iteration*: (yield iteratively a better policy each time)
- *Value Iteration*: (combines at different step policy evaluation & policy improvement)

- Non-finite MDPs methods:

> When solving a RL problem, we can estimate the values by prediction of find an optimal policy by control. Now typically, when we do not know the environment completely, we use iterative approaches as our methodologies. Some are said to be State-value based, which mean that we search for an optimal state-value function. Others are Action-value based, which search the optimal action-value function. Finally, some decide to mix the both and search two functions in order to approximate a solution. To make it clear, here is a overview plus some keywords of the main methodologies used in almost all reinforcement learning problems.

1) Model-based methods:

> - Modeling the environment to find the best policy through interactions between agent and the environment itself
> - Based on the learnt model, planning is applied and the policy is iteratively learnt from experience
> - Model: given a state and action, the model often predict the next state and next reward
> - Or simply an ensemble of acquired environmental knowledge (transition (T) and the reward (R) model)
> - Often use tree search algorithm (ex: MCTS) for planning and looking ahead
> - Methods that work with a given model (ex: game rules)
> - Methods that learn the model (use collected data to learn a predictive model of the future latent vector)
> - Then, policy improved via learned model

2) Model-free methods:

> - Agent learns policies directly from real experience with the environment (No planning)
> - Use trial and error to optimize for the best policy to get the most rewards over time
> - Learning by bootstrapping (making predictions and feedback errors)
> - Sampled reward values used for improving action preferences
> - On-policy vs off-policy: Different updating manner (off: sampled data for low variances) (on: less bias)
>
> *- Policy-based optimization*
>
> - Updates policy iteratively until the accumulative return is maximized
> - Often simpler policy parameterization & better convergence
> - Gradient-based: REINFORCE
> - Gradient-free methods: Cross-entropy
>
> *- Value-based optimization (Q learning)*
>
> - Temporal Difference Learning (combining Dynamical Programming and Monte Carlo concepts)
> - Learn by bootstrapping from the current estimate of the value function
> - DQN, Sarsa...
>
> *- Actor-critic*
>
> - Merging Value-based & Policy-based
> - Leverage the value function for updating the policy
> - Use a critic as described in the value-based optimization to estimate the action-value function.
> - Ex: deep deterministic policy gradient (DDPG)

---

2) MONTE-CARLO METHODS

> *- Introduction*
>
> - Solving RL tasks by averaging sample returns
> - Average are good approximation as the state's value act as the expected return

- Intermix policy evaluation and policy improvement
- Hence learn optimal behavior with environnement interactions
- 2 methods:
- Every-visit MC method: averages the returns following all visits to a state or state/action pair
- First-visit MC method: average of the returns following first visits
- Both will converge throught the expected value

- Monte-Carlo Tree Search (MCTS)

- Compute optimal action for current state
- Given model, build a search tree rooted at the current state st
- Next samples actions and next states
- Then construct and update tree with simulation starting from the root state
- When finished, select current (real) action with maximum value in search tree
- Clear explanation: https://www.youtube.com/watch?v=UXW2yZndl7U&ab_channel=JohnLevine

- Evaluation

- Learning state-value function for a given policy
- Learning estimate values of state–action pairs (q*)

- Control

- Policy improvement: Approximate optimal policies

---

3) TEMPORAL-DIFFERENCE LEARNING

- On-policy TD Control: SARSA
- Off-policy TD Control: Q-learning
- Deep Q-networks (DQN)

---

4) POLICY GRADIENT METHODS

- REINFORCE:

---

5) ACTOR-CRITIC METHODS

- Deep Deterministic Policy Gradient (DDPG)
- Soft Actor-Critic (SAC)

PART 2 -- RL CHALLENGES AND CASE STUDIES

1) Challenges in deep reinforcement learning

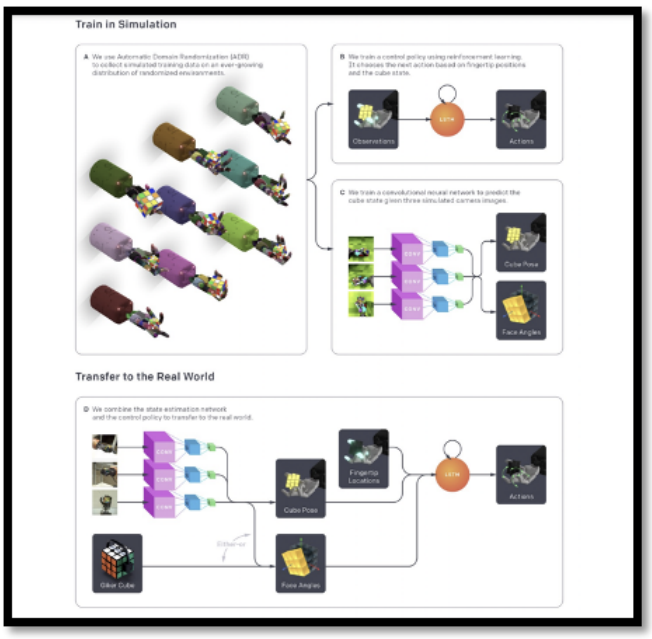
- Learning efficiency
- Meta-learning & Representation learning
- Reality gap from simulation training (Sim2Real)
- Scalability of RL
- Lack of Exploration incentives

2) Research & innovations

- Imitation learning
- Hierarchical reinforcement learning
- Domain randomization & Automatic Domain Randomization
- Transfer learning from simulation
- Reward Engineering
- Self-supervised learning
- Multi-agent RL
- Dataset generation methods

- Interesting papers and algorithms:

- AlphaGo & AlphaZero (Deepmind)
- Solving Rubik's Cube with a Robot Hand (OPEN AI)
- RL Dogfighting agents (air to air combat)



- Student section:

Dataset

In [ ]:

Model coding

In [ ]:

In [ ]:

In [ ]:

----------VIDEOS TO WATCH ----------

> 1) https://www.youtube.com/watch?v=nIglv4IfJ6s&list=LL&index=17&ab_channel=CrashCourse
>
> 2) https://www.youtube.com/watch?v=0MNVhXEX9to&list=PLIrbp3VWo-GIDEXNG9zZrW5reQPmPnXwk&index=37&ab_channel=SteveBrunton

>

>

- *REFERENCES*

- Lab 1: Introduction

> Definitions: 1: (https://flatironschool.com/blog/deep-learning-vs-machine-learning)
>
> Predictive modeling: 1: (https://www.logianalytics.com/predictive-analytics/what-is-predictive-analytics/)
>
> Features: 1: (*https://quantdare.com/what-is-the-difference-between-feature-extraction-and-feature-selection/*) 2: P.79 Burkov
>
> Algorithms: 1: https://www.edureka.com
>
> Tradeoff: Burkov 185
>
> Reinforcement learning: 1: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292
>
> Review and examples 1: https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-models-machine-learning/) 2: https://www.mathworks.com/help/stats/machine-learning-in-matlab.html)
>
> model steps
>
> training: https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/

- Lab 2:

> Intro image: https://www.google.com/search?q=Dimensionality+reduction+projection&hl=fr&source=lnms&tbm=isch&sa=X&ved=2ahUKEwixvvjQrqzyAhVkleAKHUbLDGUQ_AUoAXoECAEC
>
> Intro image: https://www.google.com/search?q=Dimensionality+reduction&hl=fr&source=lnms&tbm=isch&sa=X&ved=2ahUKEwje1OT-razyAhUEZd8KHcxAAG0Q_AUoAXoECAEQAw&cshid=1628801529134469&biw=1520&bih=824#imgrc=KJqXqOs6ayY-LM
>
> Example PCA: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Principal%20Component%20Analysis%20(PCA).ipynb
>
> Image ICA: https://linuxtut.com/en/c05908db9aebd1afa99f/
>
> Example ICA: https://github.com/scliu476/ML-Exercise-10-Independent-Component-Analysis/blob/master/Independent%20Component%20Analysis%20Lab%20%5BSOLUTION%5D.ipynb
>
> Good sources:
>
> https://jonathan-hui.medium.com/machine-learning-singular-value-decomposition-svd-principal-component-analysis-pca-1d45e885e491
>
> k-Fold Cross-Validation
>
> https://machinelearningmastery.com/k-fold-cross-validation/
>
> Examples: https://github.com/aapatel09/handson-unsupervised-learning/blob/master/04_anomaly_detection.ipynb
>
> Photo + source LDA: https://sebastianraschka.com/Articles/2014_python_lda.html
>
> Example LDA: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Linear%20Discriminant%20Analysis%20(LDA).ipynb

---

- Lab 3:

> IMAGE INTRO: https://www.analyticsvidhya.com/blog/2015/08/comprehensive-guide-regression/
>
> EXAMPLE 1: https://www.askpython.com/python/examples/linear-regression-in-python
>
> EXEMPLE 2: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/All%20Regression%20Together.ipynb

> EXEMPLE 3: https://github.com/piyush2896/PolynomialRegression-Tutorial/blob/master/Regressions%2C%20Regularizations%20and%20Learning%20Curves.ipynb
>
> EXEMPLE 4: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/All%20Regression%20Together.ipynb
>
> Theory Polynomial: http://home.iitk.ac.in/~shalab/regression/Chapter12-Regression-PolynomialRegression.pdf
>
> EXEMPLE 6: https://github.com/techshot25/HealthCare/blob/master/HealthCare.ipynb
>
> Video: https://www.youtube.com/watch?v=DCZ3tsQloGU&ab_channel=AugmentedStartups EQUATION 9.9 P.324 / ADD PICTURE P.325

- Lab 4:

> Image intro: https://www.javatpoint.com/regression-vs-classification-in-machine-learning
>
> IMAGE LOGISTIC: https://www.analyticsvidhya.com/blog/2021/07/perform-logistic-regression-with-pytorch-seamlessly/
>
> Example logistic: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Logistic%20Regression.ipynb
>
> Naive bayes example: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Natural%20Language%20Processing%20(NLP).ipynb
>
> SVM example: https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/05.07-Support-Vector-Machines.ipynb
>
> Image SVM: https://www.edureka.co/blog/support-vector-machine-in-python/
>
> Image SVM: https://medium.com/@zxr.nju/what-is-the-kernel-trick-why-is-it-important-98a98db0961d
>
> Image KNN: https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn
>
> KNN example: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/All%20Classification%20Together.ipynb

---

- Lab 5:

> Image intro: https://www.extrahop.com/company/blog/2019/supervised-vs-unsupervised-machine-learning-for-network-threat-detection/
>
> Image intro: https://www.geeksforgeeks.org/clustering-in-machine-learning/
>
> Example K-mean: https://github.com/dshah98/Machine_Learning_with_Python/blob/main/K-Mean%20Clustering.ipynb
>
> HC image: https://www.mygreatlearning.com/blog/hierarchical-clustering/
>
> Example Hierarchical Clustering : https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Hierarchical%20Clustering.ipynb
>
> Example GMM: https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/05.12-Gaussian-Mixtures.ipynb
>
> Image GMM: https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95
>
> Image GMM: https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/05.12-Gaussian-Mixtures.ipynb

---

- Lab 6:

> Image introduction: https://lawtomated.com/a-i-technical-machine-vs-deep-learning/
>
> Image nature: https://www.researchgate.net/figure/Similarity-between-biological-and-artificial-neural-networks-Arbib-2003a-Haykin_fig2_326417061
>
> Image basic comoponents ANNs: https://www.smartsheet.com/neural-network-applications
>
> Image basic comoponents ANNs: (Rosebrock P.126)
>
> https://www.programmersought.com/article/73794960192/
>
> https://deeplearningdemystified.com/article/fdl-3

> https://www.semanticscholar.org/paper/Automatic-differentiation-in-machine-learning%3A-a-Baydin-Pearlmutter/da118b8aa99699edd7609fbbd081d5b93bc2e87b/figure/0
>
> SGD: http://www.its.caltech.edu/~nazizanr/papers/SMD.html
>
> MLP image: https://laptrinhx.com/multi-layer-perceptron-mlp-lightly-explained-1623637955/
>
> LSTM EXAMPLE: https://github.com/PacktWorkshops/The-Deep-Learning-with-Keras-Workshop/blob/master/Chapter09/Activity9.01/Activity9_01.ipynb
>
> RNN Sequence problem: P580 (ML book)
>
> Image LSTM: https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/
>
> Image intro part 2: https://www.springboard.com/blog/ai-machine-learning/deepfakes-hollywood/
>
> RESNET: https://debuggercafe.com/residual-neural-networks-resnets-paper-explanation/ RESNET IMAGE: https://github.com/sumanyumuku98/Action-Motivation
>
> Example CNN (+ Image): https://github.com/dshah98/Machine_Learning_with_Python/blob/main/Convolutional%20Neural%20Networks%20(CNN).ipynb
>
> IMAGE autoencoder: https://hackernoon.com/autoencoders-deep-learning-bits-1-11731e200694 https://medium.com/analytics-vidhya/generative-modelling-using-variational-autoencoders-vae-and-beta-vaes-81a56ef0bc9f https://www.google.com/search?q=noise+removal+autoencoder&tbm=isch&ved=2ahUKEwi5k7eo7J_yAhXgg3IEHbS_DkUQ2-cCegQIABAA&oq=noise+removal+autoencoder&gs_lcp=CgNpbWcQAzoGCAAQBxAeOggIABAIEAcQHjoICAAQBxAFEB46BggAEAgQHlCi0gFYt6qAQ&bih=876&biw=1600#imgrc=pNme0M4B56ASqM https://www.google.com/search?q=Convolutional+autoencoder&source=lnms&tbm=isch&sa=X&sqi=2&ved=2ahUKEwiqo-L67J_yAhUfFlkFHflwClUQ_AUoAXoECAEQAw&biw=1520&bih=824#imgrc=fc5OeqBetIomYM https://www.google.com/search?q=Variational+autoencoder&source=lnms&tbm=isch&sa=X&sqi=2&ved=2ahUKEwinpNCo7Z_yAhXJGVkFHd21CoYQ_AUoAXoECAEQAw&biw=
>
> Image GAN: https://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html
>
> Image GAN: http://3dgan.csail.mit.edu/
>
> Image GAN: https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

---

- Lab 7:

> Image intro: https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf

>

>

>

>

> Practical Cases: ROBOT CUBE: https://openai.com/blog/solving-rubiks-cube/
