CYCLE 5

RESOLUTION DE PROBLEMES PAR UTILISATION DE L'INGENIERIE NUMERIQUE OU L'APPRENTISSAGE AUTOMATISE

COURS - PSI

CHAPITRE 1
ALGORITHMES D'INTELLIGENCE ARTIFICIELLE

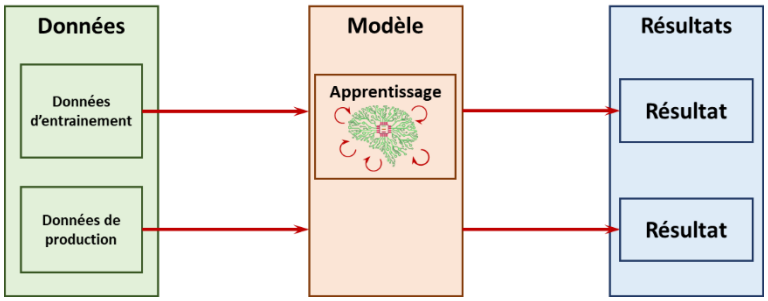
- ➔ Analyser les principes d'intelligence artificielle.
  - Régression et classification, apprentissages supervisé et non supervisé.
  - Phases d'apprentissage et d'inférence.
  - Réseaux de neurones (couches d'entrée, cachées et de sortie, neurones, biais, poids et fonction d'activation).
- ➔ Interpréter et vérifier la cohérence des résultats obtenus expérimentalement, analytiquement :
- ➔ Résoudre un problème en utilisant une solution d'intelligence artificielle
  - Apprentissage supervisé.
  - Choix des données d'apprentissage.
  - Mise en oeuvre des algorithmes (*k* plus proches voisins et régression linéaire multiple et réseaux de neurones).
  - Phases d'apprentissage et d'inférence.

L'objectif de ce cours est d'introduire l'Intelligence Artificielle, en mettant en pratique les algorithmes classiques de l'IA.

1 INTRODUCTION :

Vous pouvez visionner le cours d'informatique sur l'introduction à l'Intelligence Artificielle qui se trouve à l'adresse suivante :
drive.google.com/file/d/1SbiNheUKIhYQVewPIJCccmtA4Mo3U_4u/view?usp=sharing

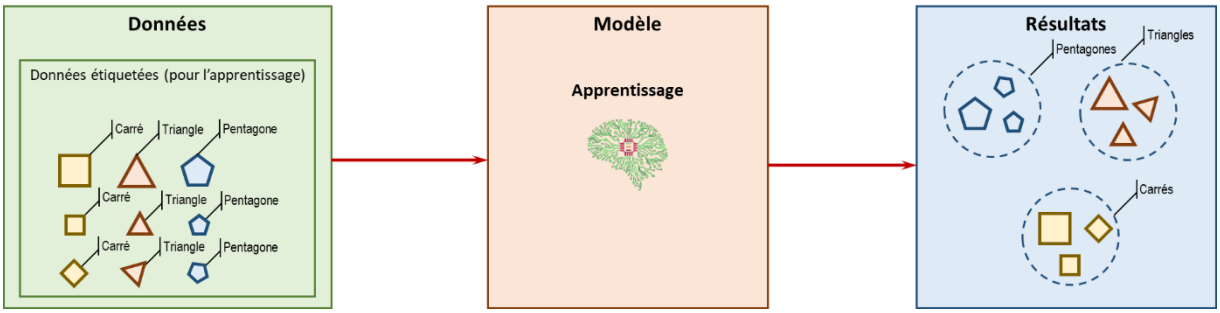
En résumé, le **Machine learning** (l'apprentissage automatique) est un champ de l'intelligence artificielle dont l'objectif est d'analyser un grand volume de données afin de déterminer des motifs et de réaliser un modèle prédictif.
L'apprentissage comprend deux phases :
• l'entraînement (ou apprentissage) est une phase d'estimation du modèle à partir de données d'observations ;
• la mise en production du modèle (inférence) est une phase pendant laquelle de nouvelles données sont traitées dans le but d'obtenir le résultat souhaité.
L'entraînement peut être poursuivi même en phase de production.



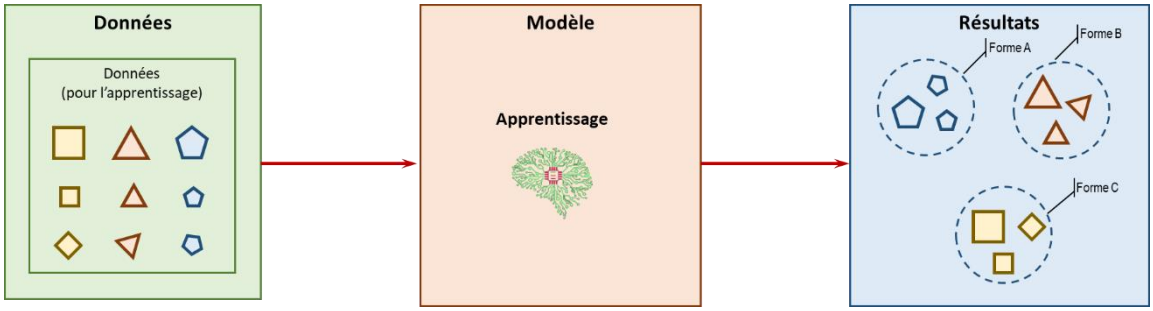
On distingue en général 3 grandes catégories :

➔ **Apprentissage supervisé** avec 2 taches :

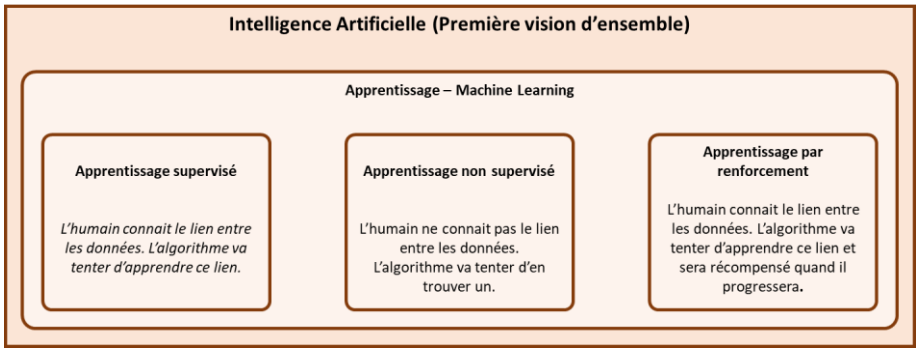
- L'apprentissage : Tâche d'apprentissage au cours duquel l'algorithme (ou fonction de prédiction) va, à partir d'un ensemble de **données étiquetées**, déterminer un lien entre un ensemble les données et les étiquettes.
- L'inférence : Tâche au cours duquel l'algorithme (ou fonction de prédiction) va, à partir d'un ensemble de données non étiquetées, prédire l'étiquette.



➔ **Apprentissage non supervisé : Clustering.** Tâche d'apprentissage au cours duquel l'algorithme (ou fonction de prédiction) va, à partir d'un ensemble de données **non étiquetées**, déterminer un lien entre les données (et les regrouper).

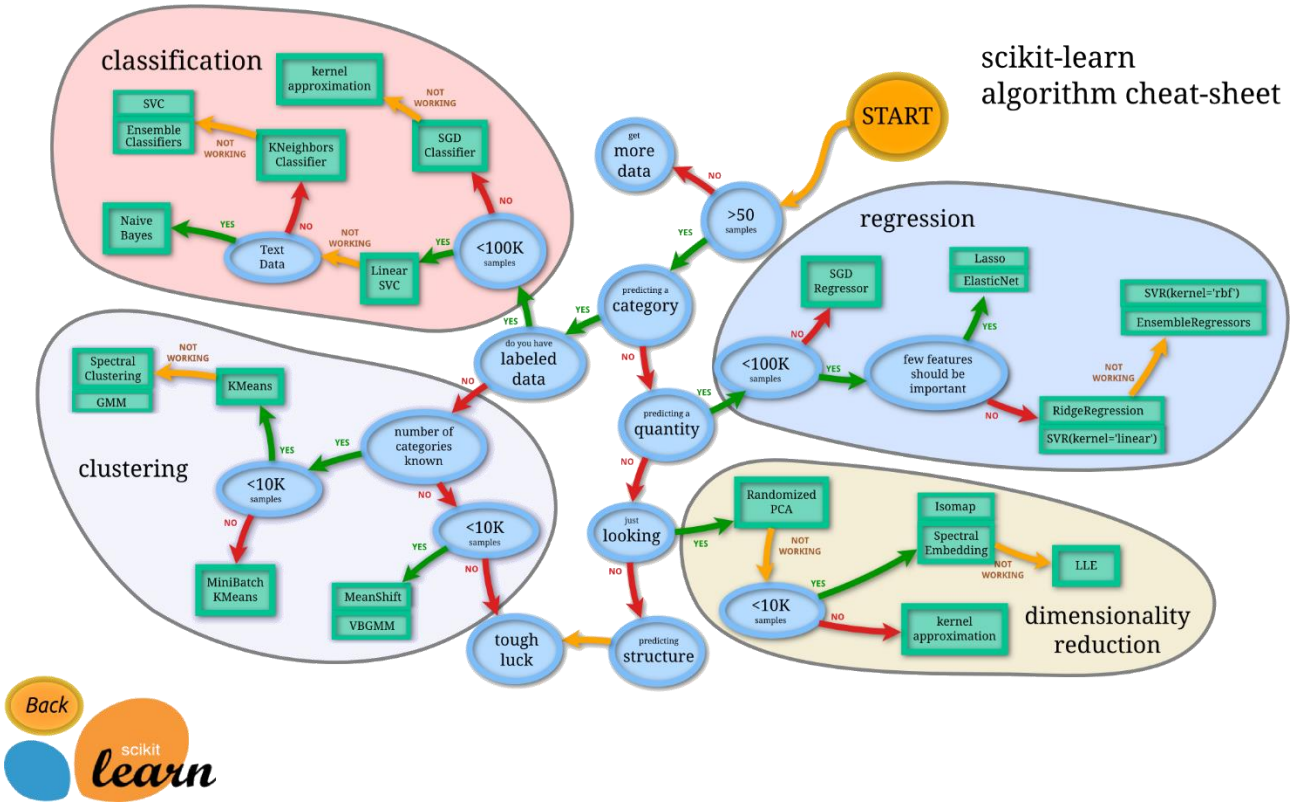


➔ **Apprentissage par renforcement**. Si au cours de l'apprentissage supervisé, un mécanisme de récompense est mis en oeuvre pour améliorer les performances du modèle, on parle d'apprentissage par renforcement.



2 MECANISMES D'APPRENTISSAGE

La bibliothèque `scikit-learn` propose une classification de divers algorithmes en fonction des apprentissages automatiques (hors réseaux de neurones).



Une fois un algorithme d'apprentissage choisi, on se pose la question de la validation du modèle. Quels critères et outils vont nous permettre de considérer que notre apprentissage est « bon » ?

Lors d'un problème de classification, il est par exemple possible de déterminer les écarts entre les valeurs prédites par l'algorithme et les valeurs cibles.

```
import sklearn.metrics as skm
# X(numpy.ndarray) : données d'entrées
# Y(numpy.ndarray) : données cibles correspondantes
# Y_pred(numpy.ndarray) : données prédites par l'algorithme de classification
print(skm.accuracy_score(Y, Y_pred))
```

Matrices de confusion : La matrice de confusion est une métrique permettant de déterminer la qualité d'une classification. En abscisses sont indiquées les valeurs réelles, et en ordonnée les valeurs prédites.

Critère de validation : erreur moyenne quadratique (*mean squared error*)

$$msq = \frac{1}{N}\sum_{i=1}^{n} \left\| Y_i - \hat{Y}_i \right\|^2$$

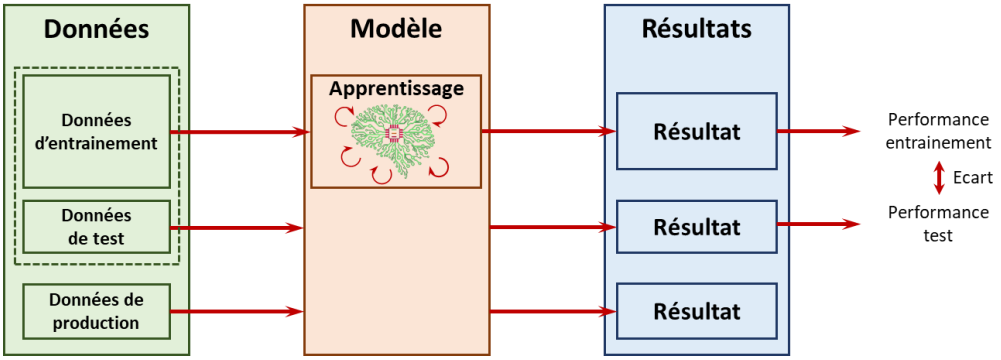
en notant Y_i la valeur réelle (étiquetée) et $\hat{Y}_i$ la valeur estimée par l'algorithme

Lorsque l'on souhaite disposer d'un modèle défini par un algorithme d'IA, il faut en premier lieu disposer de données. En apprentissage supervisé, il est nécessaire de connaître des données d'entrées ainsi que les sorties correspondantes (appelées aussi cibles ou *target*).

Dans le cadre d'une programmation `Python` avec la bibliothèque `scikit-learn` les données d'entées et de sorties peuvent être implémentées en utilisant des `numpy.ndarray`.

Ainsi, si les données d'entrées, stockées dans la variable data sont composées de n observations elles mêmes composées de p catégories, le `shape` de data sera `(n,p)`.

La donnée cible sera quant à elle un vecteur de *n* valeurs.
Lors du démarrage d'un apprentissage supervisé, on dispose d'un jeu de données comprenant les données d'entrée et les cibles correspondantes. Il est d'usage de séparer ces données en deux parties :
• les données d'entraînement permettant... d'entraîner le modèle. Dans la pratique on utilise entre 60 et 80% des données de base;
• les données de te test, permettant de valider l'apprentissage (ou le modèle), dans la pratique entre 20 et 40% des données de base.
On commence donc par réaliser un apprentissage sur les données d'entraînement. Cet entraînement produit des résultats. Connaissant les cibles correspondant aux données d'entraînement, on peut donc en déduire une performance du modèle sur le traitement des données d'entraînement.
On utilise alors le modèle en lui donnant les données de test. De même, on peut donc en déduire une performance du modèle sur le traitement des données de test.



scikit-learn permet de séparer aléatoirement des données en fixant le pourcentage de données de validation.

```
from sklearn.model_selection import train_test_split
# X(numpy.ndarray) : données d'entrées
# Y(numpy.ndarray) : données cibles correspondantes
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33)
```

3 ALGORITHMES D'APPRENTISSAGE

3.1 Algorithmes des k plus proches voisins

L'algorithme des k plus proches voisins (aussi appelé *k-nearest neighbors algorithm – k-NN*) permet de réaliser des opérations de régression et de classification sur un ensemble de données.
Pour une première approche, cette méthode permet d'estimer la classe d'une nouvelle donnée en déterminant la norme entre cette nouvelle donnée et l'ensemble des données du jeu d'entraînement. Parmi les k plus proches voisins, on recherche la classe majoritaire et on attribue cette classe à la nouvelle donnée.

Première étape – Calculer la distance

Il faut tout d'abord définir une fonction de distance. On peut par exemple utiliser la distance euclidienne. Pour deux vecteurs x_1 et x_2 de taille N, $d = \sqrt{\sum_{i=1}^{n} \left(x_{1,i} - x_{2,i}\right)^2}$.

```
import math as m
def distance(x1:list, x2:list) -> float :
    distance = 0.
    for i in range(len(x1)):
        distance += (x1[i]-x2[i])**2
    return m.sqrt(distance)
```

Deuxième étape – Détermination des plus proches voisins

Pour cette étape, on dispose de données pour l'entraînement `data_train` et d'une donnée à tester `data_test`. Il va falloir calculer la distance entre la donnée à tester et les données d'entraînement puis trier les données.

Enfin, on retournera les k lignes les plus proches de `data_test`.

```
def get_voisins(data_train:list, data_test, k:int) -> list :
    distances =[]
    for ligne in data_train :
        d = distance(ligne,data_test)
        distances.append([ligne,d])
    # Tri de la liste suivant la seconde colonne (distances)
    distances.sort(key=lambda t : t[1])
    voisins = []
    for i in range(k) :
        voisins.append(distance[i][0])
    return voisins
```

Troisième étape – Prédiction

Une fois qu'on connaît les k plus proches voisins, on regarde quelle est la classe majoritaire parmi ces voisins.

```
def prediction(data_train:list, data_test, k:int) -> list :
    voisins = get_voisins(data_train, data_test, k)
    sorties = [ligne[-1] for ligne in voisins]
    predict = max(set(sorties), key=sorties.count)
    return predict
```

Prédiction avec `scikit-learn`

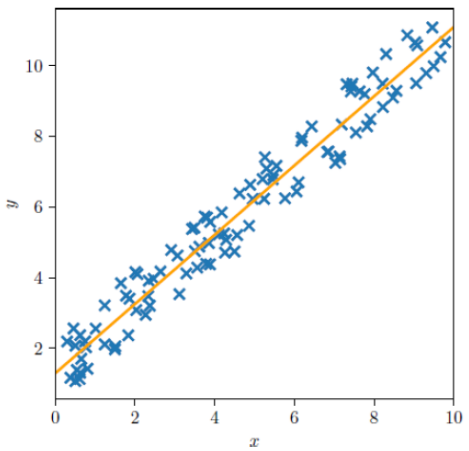
Un exemple rapide pour réaliser une classification k-NN avec `scikit-learn`.

```
import numpy as np
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier

iris_X, iris_y = datasets.load_iris(return_X_y=True)

 # A random permutation, to split the data randomly
 np.random.seed(0)

 # Chargement des données et séparation des données
 indices = np.random.permutation(len(iris_X))
 iris_X_train = iris_X[indices[:-10]]
 iris_y_train = iris_y[indices[:-10]]
 iris_X_test = iris_X[indices[-10:]]
 iris_y_test = iris_y[indices[-10:]]

 # Création du classifieur
 knn = KNeighborsClassifier() # KNeighborsClassifier(n_neighbors=k) pour choisir k
 # Entraînement du classifieur
 knn.fit(iris_X_train, iris_y_train)
 # Prédiction sur le jeu de test (à comparer avec iris_y_test)
 knn.predict(iris_X_test)
```

3.2 Régression univariée

Dans le cadre de la régression linéaire unvariée, on dispose de n observations : $\mathscr{D} = \{(x_1, y_1), (x_2, y_2), ..., (x_i, y_i), ..., (x_n, y_n)\}$ et des étiquettes y_i associées. La régression linéaire consiste à choisir une fonction de prédiction f de la forme, $\forall x \in \mathbb{R}, f(x) = ax + b$ avec$(a, b) \in \mathbb{R}^2$. Il va donc falloir minimiser l'erreur entre les prédictions et la base de données d'entraînement. La minimisation se note :

$$(\hat{a}, \hat{b}) = \arg\min_{(a,b)\in\mathbb{R}^2} \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b))^2.$$



(Méthode des moindres carrés)

On montre que si $\sum_{i=1}^{n} x_i^2 \neq \left(\sum_{i=1}^{n} x_i\right)^2$ le problème à une unique solution. Sinon le système est indéterminé et il y a une infinité de solutions.

Exemple de régression linéaire univariée avec `scikit-learn`

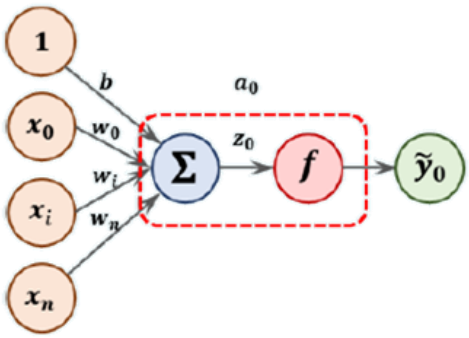
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)

# Use only one feature
diabetes_X = diabetes_X[:, np.newaxis, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes_y[:-20]
diabetes_y_test = diabetes_y[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients
print('Coefficients: \n', regr.coef_)
# The mean squared error
print('Mean squared error: %.2f'
      % mean_squared_error(diabetes_y_test, diabetes_y_pred))
# The coefficient of determination: 1 is perfect prediction
print('Coefficient of determination: %.2f'
      % r2_score(diabetes_y_test, diabetes_y_pred))

# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test, color='black')
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```

3.3 Réseaux de neurones

3.3.1 Modèle de neurone

Prenons la représentation suivante pour un neurone. On note :

- X le vecteur d'entrée et x_i les données de la couche d'entrée ;
- w_i les poids (poids synaptiques) ;
- b le biais ;
- z_0 la somme pondérée des entrée ;
- f une fonction d'activation ;
- $\tilde{y}_0$: la valeur de sortie du neurone.



On a donc, dans un premier temps :

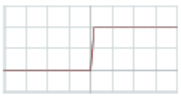
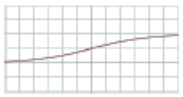
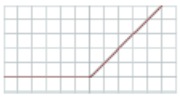
$$z_0 = b + \sum_{i=0}^{n} w_i x_i.$$

Après la fonction d'activation, on a donc en sortie du neurone :

$$\tilde{y}_0 = f(z_0) = f\left(b + \sum_{i=0}^{n} w_i x_i\right).$$

1. La notation tilde ($\tilde{y}_0$) vient du fait que la valeur de sortie d'une neurone est une valeur estimée qu'il faudra comparer à y_0 valeur de l'étiquette utilisée pour l'apprentissage supervisé.
2. Par la suite, dans la représentation graphique on ne fera pas apparaître la somme pondérée et la fonction d'activation, mais seulement la valeur de sortie du neurone (notée par exemple a_0).

Les **fonctions d'activation** sont des fonctions mathématiques appliquées au signal de sortie (z). Il est alors possible d'ajouter des non linéarités à la somme pondérée. On donne ci-dessous quelques fonctions usuelles

Identité	Heaviside	Logistique (sigmoïde)	Unité de rectification linéaire (ReLU)
			
$f(x) = x$	$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ 1 \text{ si } x \geq 0 \end{cases}$	$f(x) = \dfrac{1}{1+e^{-x}}$	$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ x \text{ si } x \geq 0 \end{cases}$

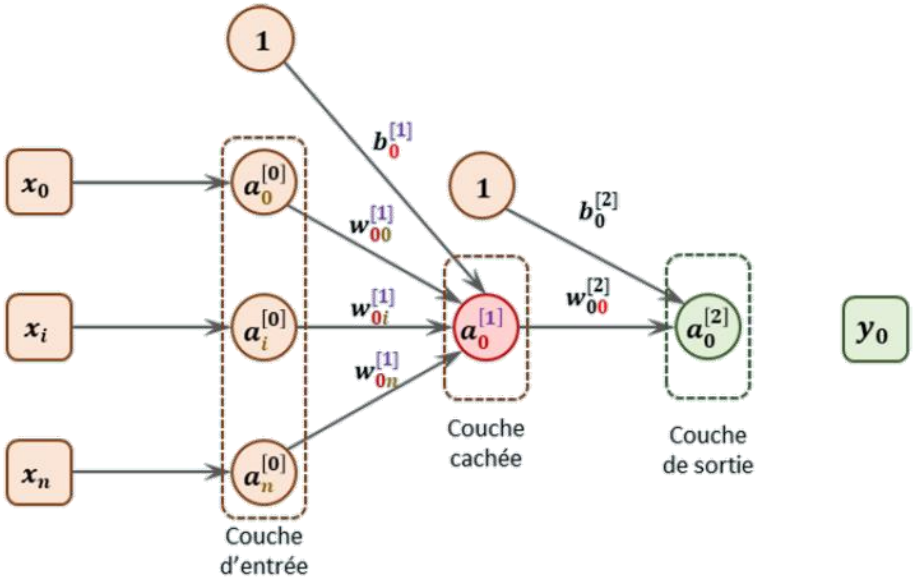
3.3.2 Réseaux de neurones

https://playground.tensorflow.org/

Un réseau de neurones est un ensemble de neurones reliés, par couches, entre eux.

Dans un réseau de neurones **dense** tous les neurones de la couche i seront reliés à tous les neurones de la couche $i+1$.

- Couche d'entrée : cette couche est une copie de l'ensemble des données d'entrées. Le nombre de neurones de cette couche correspond donc aux nombre de données d'entrées. On note $\mathbf{X} = (x_0, \ldots, x_n)$ le vecteur d'entrées.
- Couche cachée (ou couche intermédiaire) : il s'agit d'une couche qui a une utilité intrinsèque au réseau de neurones. Ajouter des neurone dans cette couche (ou ces couches) permet donc d'ajouter de nouveaux paramètres. Pour une couche, la même fonction d'activation est utilisée pour tous les neurones. En revanche la fonction d'activation utilisée peut être différente pour deux couches différentes. Les fonctions d'activations des couches intermédiaires sont souvent non linéaires.
- Couche de sortie : le nombre de neurones de cette couche correspond au nombre de sorties attendues. La fonction d'activation de la couche de sortie est souvent linéaire. On note $\mathbf{Y} = (y_0, \ldots, y_y)$ le vecteur des sorties.



En utilisant la loi de comportement du modèle de perceptron, on peut donc exprimer $\mathbf{Y} = \mathscr{F}(\mathbf{X})$ où $\mathscr{F}$ est une fonction dépendant des entrées, des poids et des biais.

Notations :

- on note $w_{jk}^{[\ell]}$ les poids permettant d'aller vers la couche ℓ depuis le neurone k vers le neurone j ;
- $b_j^{[\ell]}$ le biais permettant d'aller sur le neurone j de la couche ℓ ;
- $f^{[\ell]}$ la fonction d'activation de la couche ℓ ;
- $n^{[\ell]}$ le nombre de neurones de la couche ℓ.

Equation de propagation : Pour chacun des neurones, on peut donc écrire l'équation de propagation qui lui est associé :

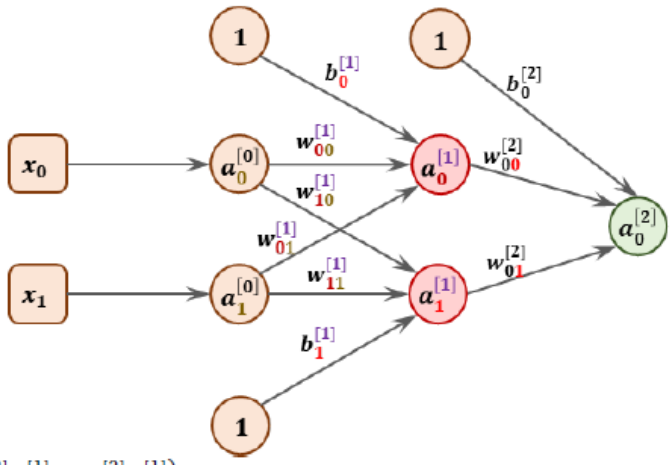
$$a_j^{[\ell]} = f^{[\ell]}\left(\sum_{k=0}^{n^{[\ell-1]}}\left(w_{jk}^{[\ell]} a_k^{[\ell-1]}\right) + b_j^{[\ell]}\right) = f^{[\ell]}\left(z_j^{[\ell]}\right).$$

Exemple :

Prenons un réseau de neurones à 3 couches :

- 1 couche d'entrée à 2 neurones ;
- 1 couche cachée à 2 neurones, de fonction d'activation f_1 ;
- 1 couche de sortie à 1 neurone, de fonction d'activation f_2 ;

Initialisation les poids et le biais avec des valeurs aléatoires : $w_0 = -0{,}3$, $w_1 = 0{,}8$ et $b = 0{,}2$.



Il est possible d'écrire que $y_0 = a_0^{[1]} = f_2\left(b_0^{[2]} + w_{00}^{[2]} a_0^{[1]} + w_{01}^{[2]} a_1^{[1]}\right)$.

Par ailleurs : $a_0^{[1]} = f_1\left(b_0^{[1]} + w_{00}^{[1]} a_0^{[0]} + w_{01}^{[1]} a_1^{[0]}\right)$ et $a_1^{[1]} = f_1\left(b_1^{[1]} + w_{10}^{[1]} a_0^{[0]} + w_{11}^{[1]} a_1^{[0]}\right)$.

Au final, on a donc

$$y_0 = a_0^{[1]} = f_2\left(b_0^{[2]} + w_{00}^{[2]}\left(f_1\left(b_0^{[1]} + w_{00}^{[1]} a_0^{[0]} + w_{01}^{[1]} a_1^{[0]}\right)\right) + w_{01}^{[2]}\left(f_1\left(b_1^{[1]} + w_{10}^{[1]} a_0^{[0]} + w_{11}^{[1]} a_1^{[0]}\right)\right)\right)$$

Les paramètres du réseau de neurones sont les poids et les biais, autant de valeurs que l'entraînement devra déterminer.

Soit un jeu de données étiquetées avec n entrées et p sorties.

On construit un réseau possédant ℓ couches et a_ℓ le nombre de neurones de la couche ℓ. Dans ce cas, la première couche est la couche d'entrée ($a_1 = n$) et la dernière couche et la couche de sortie ($a_\ell = p$).

Nombre de poids : $n_w = \sum_{i=1}^{\ell-1}(a_i \times a_{i+1})$.

Nombre de bais : $n_b = \sum_{i=2}^{\ell}(a_i)$.

Au final, le nombre total de paramètre à calculer est donné par $N = n_w + n_b$.

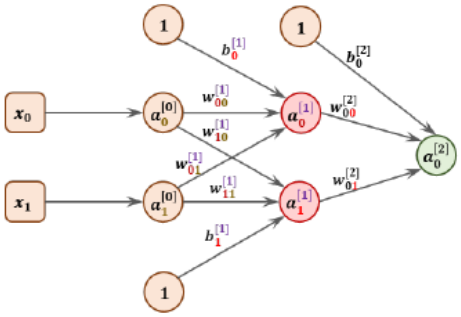
➔ **L'objectif est de minimiser l'écart entre la sortie du réseau de neurones et la valeur réelle de la sortie,** on utilise une **fonction coût (ou fonction de perte).** Il est possible de définir plusieurs types de fonctions, notamment en fonction du type de problème à traiter (classification ou régression par exemple**).**

$$C = \frac{1}{n\,b}\sum_{i=1}^{nb}\left(\tilde{\mathbf{Y}}_\mathbf{i} - \mathbf{Y}_\mathbf{i}\right)^2$$

L'objectif est dès lors de déterminer les poids et les biais qui minimisent la fonction coût.

3.3.3 Descente de gradient

En réutilisant l'exemple ci-contre, nous allons présenter succinctement comment est minimisée la fonction coût. Pour cela, il va falloir dériver la fonction coût par rapport à chacune des variables.



Cherchons uniquement à déterminer le coût que par rapport à un seul vecteur d'entrée du jeu d'entraînement. On a alors :

- $C = \left(\tilde{\mathbf{Y}}_\mathbf{1} - \mathbf{Y}_\mathbf{1}\right)^2 = \left(a_0^{[2]} - y\right)^2$;
- $a_0^{[2]} = f^{[2]}\left(z_0^{[2]}\right)$;
- $z_0^{[2]} = \sum_{k=0}^{1}\left(w_{0k}^{[2]} a_k^{[1]}\right) + b_0^{[2]}$.

Commençons par déterminer la dérivée partielle par rapport à un poids de la couche de sortie :

$$\frac{\partial C}{\partial w_{00}^{[2]}} = \frac{\partial C}{\partial a_0^{[2]}} \frac{\partial a_0^{[2]}}{\partial z_0^{[2]}} \frac{\partial z_0^{[2]}}{\partial w_{00}^{[2]}}.$$

De même, on peut calculer la dérivée partielle du coût par rapport au biais : $\frac{\partial C}{\partial b_0^{[2]}} = \frac{\partial C}{\partial a_0^{[2]}} \frac{\partial a_0^{[2]}}{\partial z_0^{[2]}} \frac{\partial z_0^{[2]}}{\partial b_0^{[2]}}$.

Calculons les dérivées nécessaires :

- $\frac{\partial C}{\partial a_0^{[2]}} = 2\left(a_0^{[2]} - y\right)$;
- $\frac{\partial a_0^{[2]}}{\partial z_0^{[2]}} = f'^{[2]}\left(z_0^{[2]}\right)$;
- $\frac{\partial z_0^{[2]}}{\partial w_{00}^{[2]}} = a_0^{[1]}$.
- $\frac{\partial z_0^{[2]}}{\partial b_0^{[2]}} = 1$.

On a donc, $\frac{\partial C}{\partial w_{00}^{[2]}} = 2\left(a_0^{[2]} - y\right) f'^{[2]}\left(z_0^{[2]}\right) a_0^{[1]}$ et $\frac{\partial C}{\partial b_0^{[2]}} = 2\left(a_0^{[2]} - y\right) f'^{[2]}\left(z_0^{[2]}\right)$.

Prenons le cas ou la fonction d'activation est la fonction identité. On a alors $\frac{\partial C}{\partial w_{00}^{[2]}} = 2\left(a_0^{[2]} - y\right) a_0^{[1]}$ et $\frac{\partial C}{\partial b_0^{[2]}} = 2\left(a_0^{[2]} - y\right)$...

On va ainsi pouvoir exprimer $\frac{\partial C}{\partial w_{00}^{[2]}}$, $\frac{\partial C}{\partial w_{01}^{[2]}}$, $\frac{\partial C}{\partial b_0^{[2]}}$, ... On pourrait ici écrire 9 équations en fonction des différents poids, des biais et des entrées x_0 et x_1.

À partir de cela, on va modifier les poids comme suit :

- $w_{00,i+1}^{[2]} = w_{00,i}^{[2]} - \eta \frac{\partial C}{\partial w_{00,i}^{[2]}}$;
- $b_{0,i+1}^{[2]} = b_{0,i}^{[2]} - \eta \frac{\partial C}{\partial b_{0,i}^{[2]}}$.

On réitère ensuite les opérations précédentes jusqu'à ce que la fonction coût ait été suffisamment réduite.

Taux d'apprentissage.

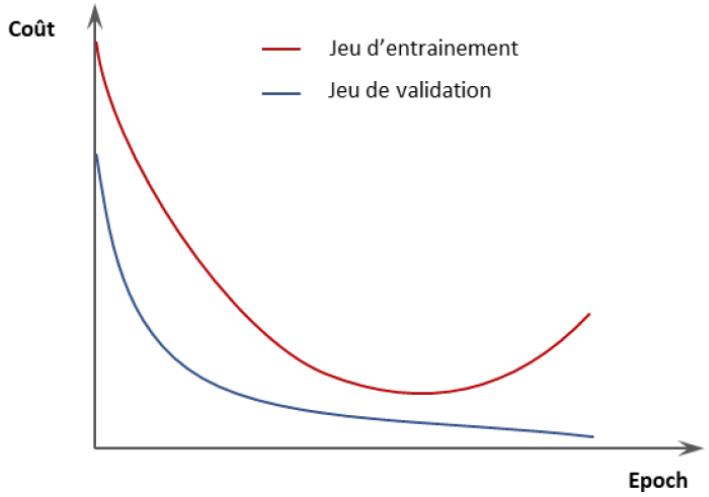
On définit l'hyperparamètre $\eta \in [0,1[$ comme étant le taux d'apprentissage. Si ce taux d'apprentissage est très grand, l'algorithme d'apprentissage mettra beaucoup de temps à trouver le minimum. S'il est trop grand, le minimum peut ne jamais être trouvé.

Dés lors, dans le cas présenté, on peut définir le gradient comme le vecteur gradC = $\begin{bmatrix} \frac{\partial C}{w^{[1]}} \\ \frac{\partial C}{b^{[1]}} \\ \vdots \\ \frac{\partial C}{w^{[l]}} \\ \frac{\partial C}{b^{[l]}} \end{bmatrix}$

3.3.4 Fin d'apprentissage

On appelle *epoch* un cycle d'apprentissage où tous les poids et tous les biais ont été mis à jour en faisant passer toutes les données du jeu d'entraînement dans les algorithmes de propagation et de rétropropagation.

Les méthodes pour stopper l'apprentissage sont essentiellement empiriques. On pourrait en effet fixer un nombre d'*epoch* ou une valeur d'erreur admissible et s'arrêter à ce moment là. Dans la pratique, se focaliser sur l'erreur n'est généralement pas satisfaisant. En effet, il y a risque de « **suraprentissage** ».



Dans la figure ci-dessus, on réalise un entraînement sur le jeu de données (une epoch) à la fin de l'epoch on dispose d'un premier modèle de réseau de neurones. On détermine alors l'erreur commise en utilisant le jeu d'entraînement puis l'erreur commise sur le jeu de validation. On calcule ensuite l'erreur commise par le modèle sur le jeu de validation. (On rappelle que le jeu de validation ne sert pas à modifier les poids et les biais.)
On réalise de même à la fin de l'epoch suivante etc...
« Logiquement » l'erreur décroît toujours avec le jeu d'entraînement car la descente du gradient a pour objectif de réduire cette erreur. Sur le jeu de validation, l'erreur décroît pendant un certain nombre d'epoch puis augmente. Il existe en fait un stade à partir duquel le réseau de neurones se spécialise sur le jeu d'entraînement et devient donc incapable de réaliser des prédictions fiables sur un nouveau je de données. On parle de surentraînement (ou d'overfitting).

▪ **Exemple** *Wikipedia*

La ligne verte représente un modèle sur-appris et la ligne noire représente un modèle régulier. La ligne verte classifie trop parfaitement les données d'entraînement, elle généralise mal et donnera de mauvaises prévisions futures avec de nouvelles données. Le modèle vert est donc au final moins bon que le noir.