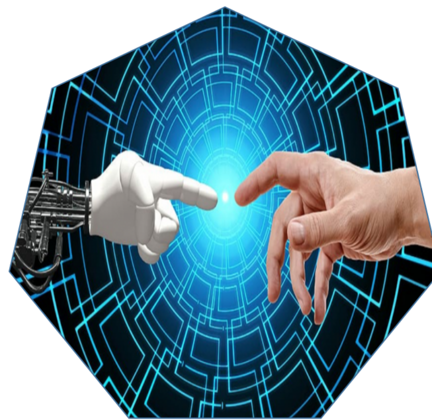
Intelligence Artificielle (IA)

Cours

v0.2

CPGE - IBN TIMIYA - MARRAKECH



Yassine FARTOUH
Professeur agrégé en Sciences Industrielles de l'Ingénieur
Ingénierie Mécanique
-SII-IM-
fartouh.cpge@gmail.com
Année scolaire : 2022/2023

CC BY NC SA

Table des matières

1 Introduction : « intelligence artificielle » (IA) 3

1.1 Vocabulaire et Définitions 3
1.2 Classification des algorithmes de machine learning 3
1.3 Synthèse 6
1.4 Les 4 librairies Python à maîtriser pour le Machine Learning 6

2 K plus proches voisins (K-Nearest Neighbour :KNN) 7

2.1 Illustration de la méthode des k plus proches voisins 7
2.2 Algorithme KNN (K Nearest Neighbors) : Classification 7
2.3 Algorithme KNN (K Nearest Neighbors) : Régression 8
2.4 Evaluation d'un système de classification 10
2.5 Exemple 11
2.6 Algorithme des k plus proches voisins : Recherche de la meilleure valeur de k 13
2.7 Matrice de confusion 14
2.8 Sensibilité et spécificité 16
2.9 Importance de la mise en forme des données 17

3 Régression linéaire 18

3.1 Intérêt de déterminer un modèle de données 18
3.2 Régression linéaire simple : Mise en équations 19
3.3 Ecriture des équations sous forme matricielle 23

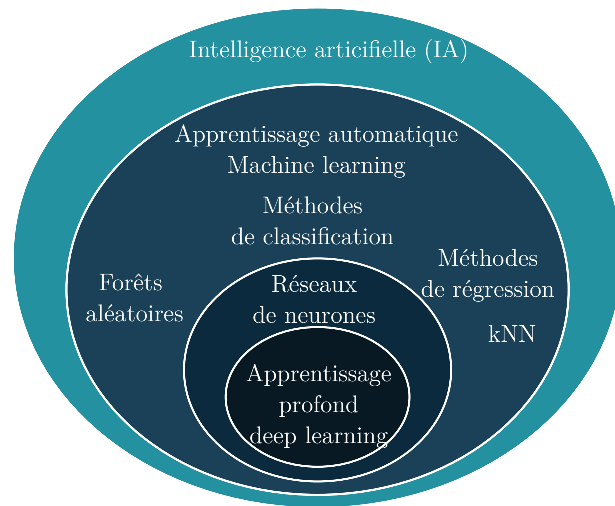
4 Réseaux de Neurones (Neural Network) 30

4.1 Modèle du neurone biologique : 30
4.2 Modèle d'un neurone artificiel : Perceptron 30

Y.FARTOUH - fartouh.cpge@gmail.com

1 Introduction : « intelligence artificielle » (IA)

1.1 Vocabulaire et Définitions



Sous le terme intelligence artificielle (IA) on regroupe l'ensemble des **théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence.**
l'intelligence artificielle a pour but de simuler un ou des comportements humains. À l'heure actuelle, il est encore difficile de développer des intelligences artificielles dites fortes. Dans le domaine de l'intelligence artificielle, des algorithmes sont développés et notamment le machine learning regroupant notamment les méthodes à réseaux de neurones dans lesquelles le deep-learning est développé. Le deep-learning est notamment utilisé pour la reconnaissance d'images.

1.2 Classification des algorithmes de machine learning

Dans le programme de CPGE, on s'intéresse à quelques algorithmes de machine learning :
- Méthode des k plus proches voisins ;
- Réseaux de neurones ;

Cependant, ces deux algorithmes ne sont qu'une partie de l'ensemble des algorithmes possibles pour réaliser du machine learning.

Il existe néanmoins trois grandes catégories d'algorithmes de machine learning qui ont principalement deux buts : la classification ou la régression (voir Figure 1).

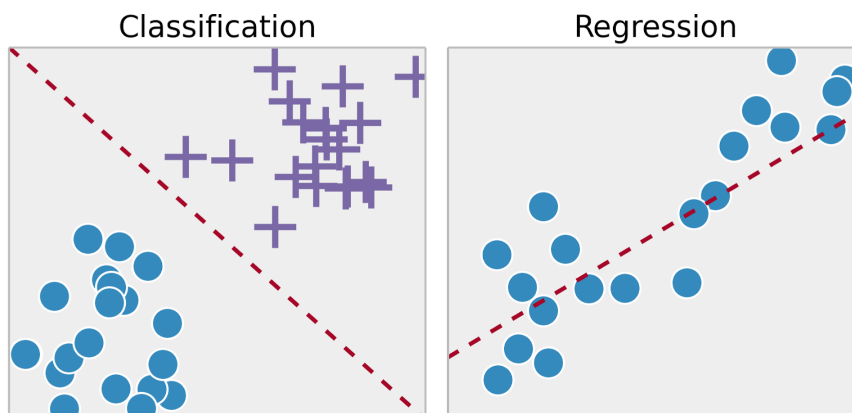


Figure 1 – Deux grandes types de problèmes : classification ou régression

BY NC SA

1.2.1 Apprentissage supervisé(Supervised Learning)

C'est un apprentissage à partir d'une base de données **(Dataset)** en connaissant les sorties relatives à chaque combinaison d'entrées.La machine reçoit des données caractérisées par des variables X et annotées d'une variable y, on parle de données avec étiquettes.

- La variable y porte le nom de **target** (la cible). C'est la valeur que l'on cherche à prédire.
- La variable X porte le nom de **features** (caractéristiques) que l'on regroupe dans une matrice.

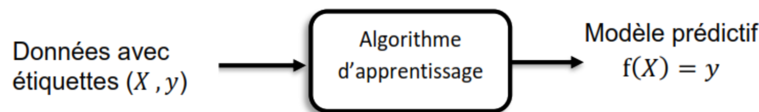
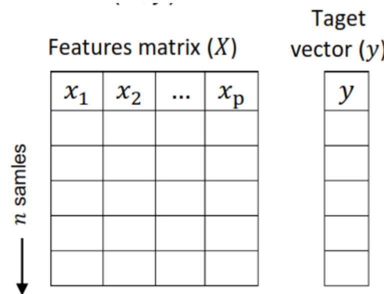


FIGURE 2 – Principe de l'apprentissage supervisé

Données : On dispose de n observations expérimentales des variables (X, y) :

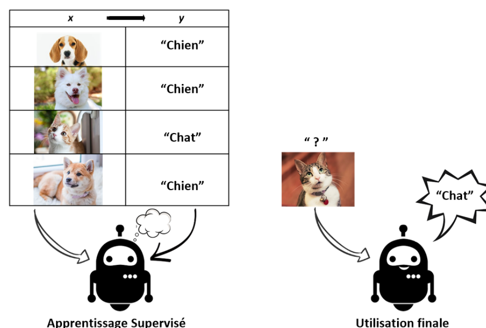
$$X = (x_1, x_2, \ldots, x_\mathrm{p})\,;X \in \mathbb{R}^p$$

Avec :
p : le nombre de variables d'entrées ;
n : la taille de la base de données.
Modèle : on cherche une fonction f tel que $f(X) = y$.



- Si $y \in \mathbb{R}$: la sortie est un réel, on parle de régression ;
- Si $y \in \{C_1, C_2, \ldots\}$ la sortie est un ensemble de classes, on parle de classification ;
- Si $y \in \{0, 1\}$ c'est la classification binaire.

Objectif : pour une nouvelle donnée $\tilde{X}$, faire une prédiction de la valeur de $\tilde{y} = \mathrm{f}(\tilde{X})$.
Exemple :Détection du contenu d'une image.



1.2.2 Apprentissage non supervisé (Unsupervised Learning)

Dans la base de donnée, on ne possède que les variables d'entrées, On ne connait pas les sorties, on dit la base de donnée n'est pas étiquetée. L'algorithme doit faire des regroupements sur des critères pour obtenir des similarités entre les données, créer des clusters (partitions ou regroupement).

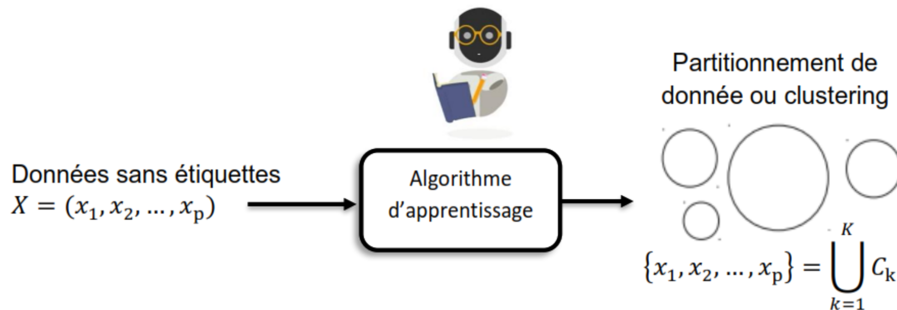
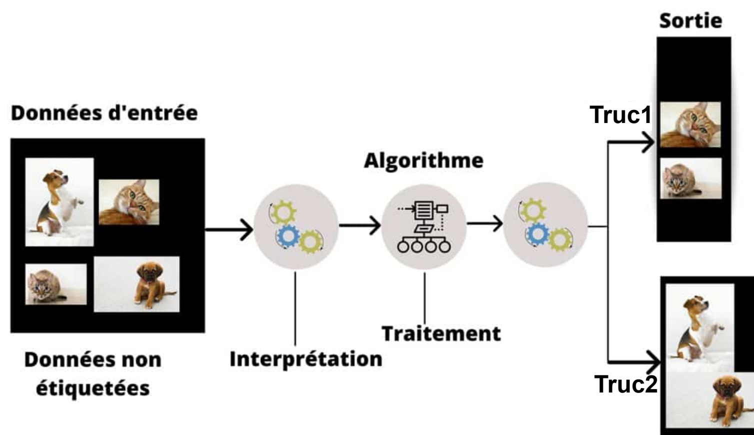


FIGURE 3 – Principe de l'apprentissage non supervisé

Exemple :



1.2.3 Apprentissage par renforcement (Reinforcement Learning)

L'apprentissage par renforcement permet de créer une base de données en fonction de la supervision de l'environnement (capteurs).
Au cours du fonctionnement du système, l'algorithme donne une récompense positive à la situation recherchée et une récompense négative à des situations non voulues. Ainsi, par apprentissage et en fonction des récompenses le système peut savoir les bonnes tâches et les mauvaises, en plus, à partir de la base de données enrichi par le nombre de situations, le système peut prédire d'autres tâches.

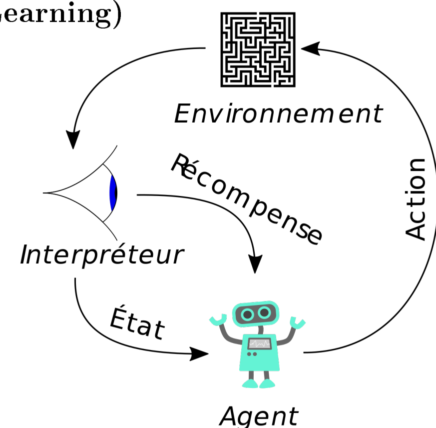


FIGURE 4 – Principe de l'apprentissage par renforcement

Exemple : Les tâches d'un robot ; Le déplacement d'une voiture dans un parcours...

1.3 Synthèse

la figure ci dessous présente un schéma synoptique des différents algorithmes de machine learning existants.

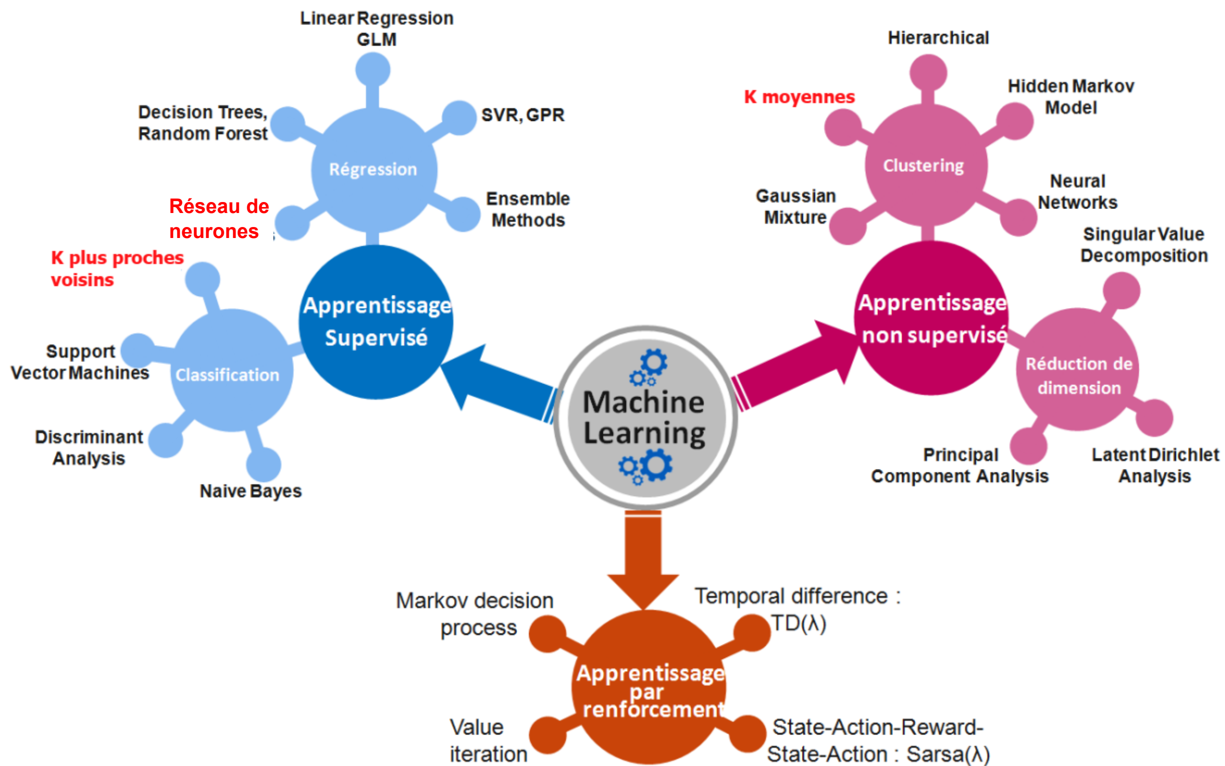


Figure 5 – Les différents algorithmes d'IA

1.4 Les 4 librairies Python à maîtriser pour le Machine Learning

- **Numpy** : La librairie qui permet de créer et manipuler des matrices simplement et avec efficacité. En Machine Learning, on insère le plus souvent notre Dataset dans des matrices.Ainsi le calcul matriciel représente l'essentiel du Machine Learning.

- **Matplotlib** : La librairie qui permet de visualiser nos Datasets, nos fonctions, nos résultats sous forme de graphes, courbes et nuages de points.

- **Sklearn** : La librairie qui contient toutes les fonctions de l'état de l'art du Machine Learning. On y trouve les algorithmes les plus importants ainsi que diverses fonctions de pre-processing.

- **Pandas** est une excellente librairie pour importer vos tableaux Excel (et autres formats) dans Python dans le but de tirer des statistiques et de charger votre Dataset dans Sklearn.

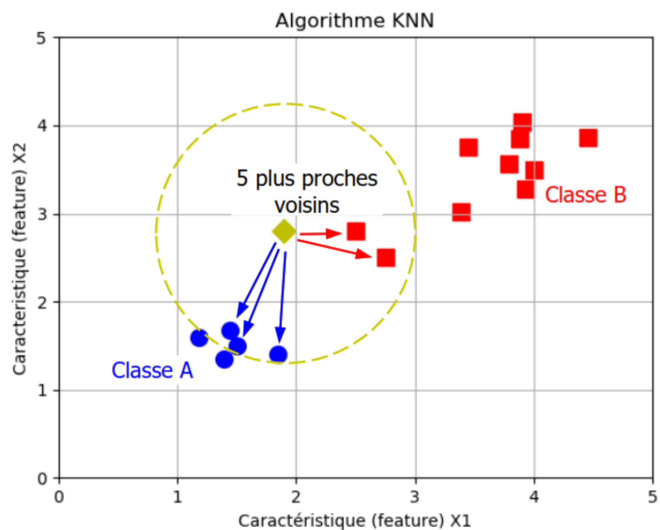
2 K plus proches voisins (K-Nearest Neighbour :KNN)

2.1 Illustration de la méthode des k plus proches voisins

L'algorithme des k plus proches voisins (***k-nearest neighbors* : kNN en anglais**) est un algorithme qui peut être utilisé pour de la classification ou de la régression. C'est effectivement un algorithme utilisant des données labellisées/étiquetées/classifiées (il est donc bien supervisé). C'est certainement l'un des algorithmes d'intelligence artificielle le plus simple à appréhender.

2.2 Algorithme KNN (K Nearest Neighbors) : Classification

L'algorithme des K plus proches voisins est un algorithme d'apprentissage supervisé que l'on peut utiliser pour la classification de données. Le principe est extrêmement simple : si les K plus proches voisins d'un élément sont majoritairement d'une certaine classe, alors cet élément sera associé à cette classe.
Par exemple, sur la figure ci-contre, les 5 plus proches voisins de l'élément que l'on cherche à classer sont majoritairement de classe A. On peut en déduire que cet élément sera probablement lui aussi de classe A.



Remarque

- Le paramètre K est appelé **hyper-paramètre**.Son choix conditionne l'efficacité de l'algorithme.
- On peut remarquer que si on avait choisi K=3 sur l'exemple précédent, la conclusion aurait été différente.
- La proximité avec des voisins peut être évaluée de différente façons (distance euclidienne, distance de Manhattan, distance de Tchebychev, etc.).

En **CPGE**, c'est la **distance euclidienne** qui est préconisée.

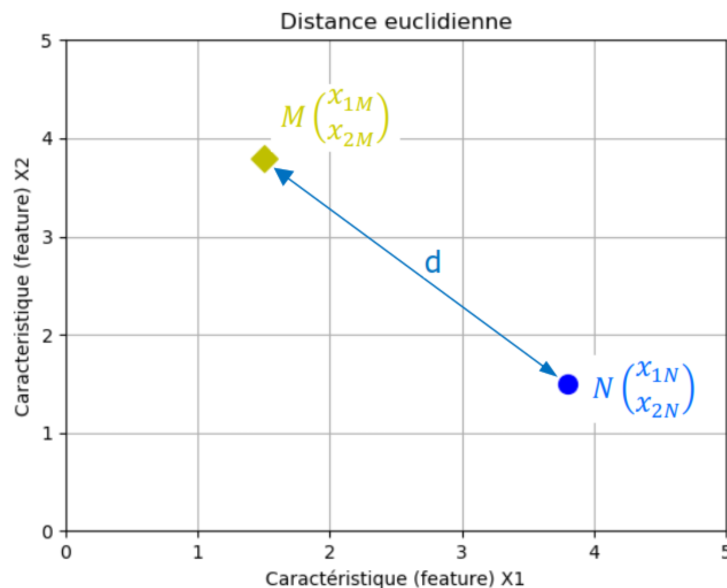
En deux dimensions, cela donne :

$$MN = \sqrt{(x_{2N} - x_{2M})^2 + (x_{1N} - x_{1M})^2}$$

Plus généralement, dans un espace de dimension n :

$$MN = \sqrt{\sum_{i=1}^{n} (x_{iN} - x_{iM})^2}$$

Y.FARTOUH - fartouh.cpge@gmail.com



On présente sous forme de pseudo-code l'algorithme des k plus proches voisins dans le cas de la classification .

Algorithme 1 : Algorithme des k plus proches voisins : Classification

Initialisation;
Données : $\mathcal{D} = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \cdots, (\mathbf{x_m}, y_m) = (\mathbf{x}, y)\}$;
− m observations en n dimensions : $\mathbf{x} \in \mathbb{R}^{m*n}$;
− m étiquettes : $\mathbf{y} \in \mathbb{R}^{m*1}$;
Objectif : Déterminer l'étiquette associée à $\mathbf{x}^*$;
pour *i variant de* 1 *à* m **faire**
  Établir le tableau de taille m des distances entre $\mathbf{x}^*$ et $\mathbf{x_i}$;
fin
Déterminer les k points $\mathbf{x_i}$ les plus proches de $\mathbf{x}^*$;
Par le vote majoritaire parmi les k plus proches voisins, on détermine l'étiquette de $\mathbf{x}^*$. Cette étiquette est notée y^* ;
return y^*

2.3 Algorithme KNN (K Nearest Neighbors) : Régression

On considère une fonction f :

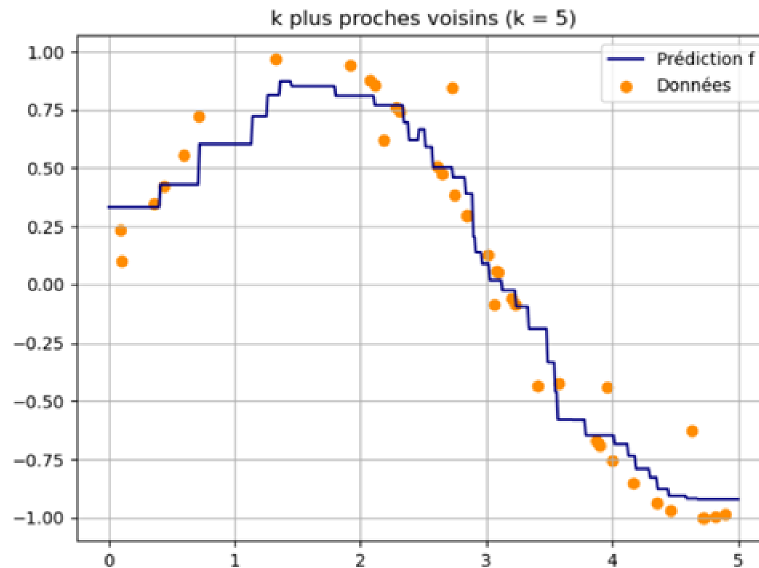
$$f: \begin{array}{ccc} \mathbb{R}^n & \longrightarrow & \mathbb{R} \\ \mathbf{x} & \longmapsto & f(\mathbf{x}) \end{array}$$

$\mathbf{x}$ est un vecteur constitué de n composantes $\mathbf{x} = (x_0, x_1, \cdots, x_{n-1})$. Ainsi, f est une fonction dite vectorielle. Soit l'ensemble $\{\mathbf{x_1}, \mathbf{x_2}, \cdots, \mathbf{x_m}\}$ de m points évalués par la fonction f. Ainsi, $f(\mathbf{x_1})$, $f(\mathbf{x_2})$ etc. sont des valeurs connues. Par la méthode des k plus proches voisins, on cherche à estimer la valeur de $\mathbf{x}^*$, avec $\mathbf{x}^*$ un point dont on cherche à estimer la valeur. On note $\{\mathbf{x_{k1}}, \mathbf{x_{k2}}, \cdots, \mathbf{x_{kk}}\} \subset$

$\{\mathbf{x_1}, \mathbf{x_2}, \cdots, \mathbf{x_m}\}$ les k plus proches de $\mathbf{x}^*$. Par la méthode des k plus proches voisins, on estime :

$$f(\mathbf{x}^*) = \frac{1}{\mathbf{k}} \sum_{\mathbf{i}=1}^{\mathbf{k}} f(\mathbf{x_{ki}})$$

Voici, un exemple sur une fonction f :



Remarque

On constate que la prédiction de la fonction f (on peut noter cette prédiction $\tilde{f}$) n'est pas interpolante. Cette approche est intéressante lorsque les données sont bruitées, la fonction $\tilde{f}$ obtenue sert donc potentiellement de filtre.

On présente sous forme de pseudo-code l'algorithme des k plus proches voisins dans le cas de la de la régression.

Algorithme 2 : Algorithme des k plus proches voisins : Régression

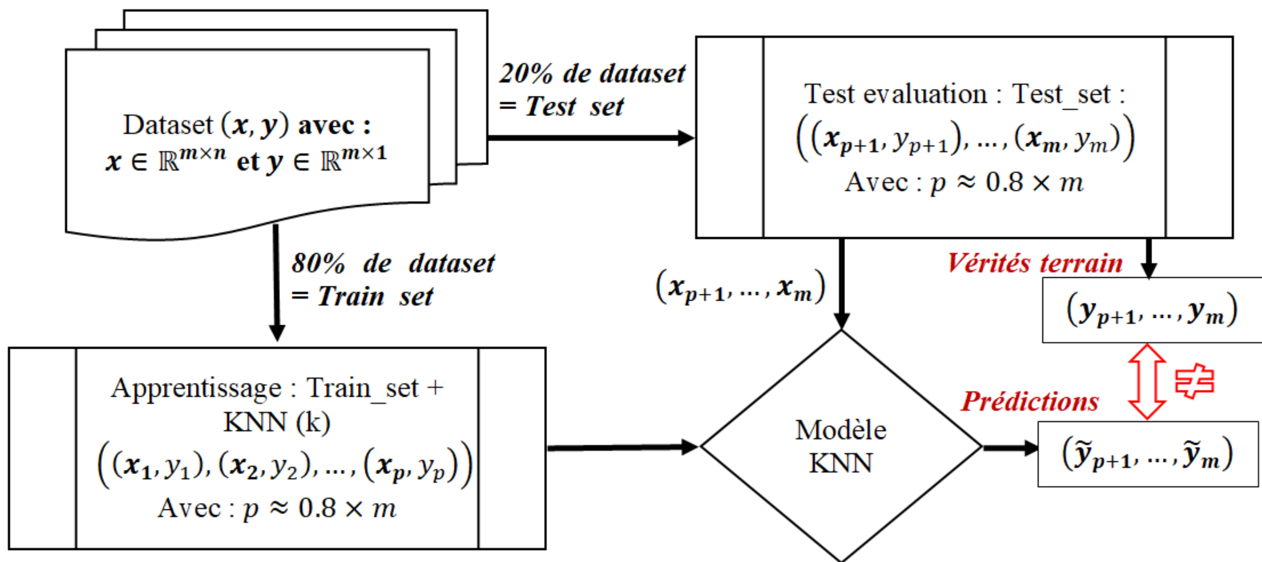
Initialisation;
Données : $\mathcal{D} = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \cdots, (\mathbf{x_m}, y_m) = (\mathbf{x}, y)\}$;
− m observations en n dimensions : $\mathbf{x} \in \mathbb{R}^{m*n}$;
− m étiquettes : $f(\mathbf{x_i}) = y_i \in \dagger$,$\mathbf{y} \in \mathbb{R}^{m*1}$;
Objectif : Estimer la valeur de $f(\mathbf{x}^*)$;
pour i *variant de* 1 *à* m **faire**
| Établir le tableau de taille m, des distances entre $\mathbf{x}^*$ et $\mathbf{x_i}$;
fin
Déterminer les k points $\mathbf{x_{ki}}$ les plus proches de $\mathbf{x}^*$ avec $\mathbf{x_{ki}}$ le $i^{\text{ème}}$ point le plus proche de $\mathbf{x}^*$ parmi les k plus proches ;
Estimer la moyenne $f(\mathbf{x}^*) = \frac{1}{\mathbf{k}} \sum_{\mathbf{i}=1}^{\mathbf{k}} f(\mathbf{x_{ki}})$;
return $f(\mathbf{x}^*)$

2.4 Evaluation d'un système de classification

Le Dataset initial doit être divisée en deux :

- **Train_set** (pour l'apprentissage) généralement 80%
- **Test_set** (pour tester et évaluer le modèle) généralement 20%

L'ensemble de test « **Test_set** » doit être différent de l'ensemble d'entraînement « **Train-set** ». On compare les prédictions de l'ensemble « **Test_set** » aux vraies valeurs (vérités terrains).



Il est possible de juger l'efficacité de l'algorithme à l'aide d'un outil appelé **matrice de confusion**.

		Nombre d'éléments prédits dans chaque classe		
		0 Setosa	1 Versicolor	2 Virginica
Nombre d'éléments réellement contenus dans chaque classe	0 Setosa	10	1	0
	1 Versicolor	0	8	1
	2 Virginica	0	0	10

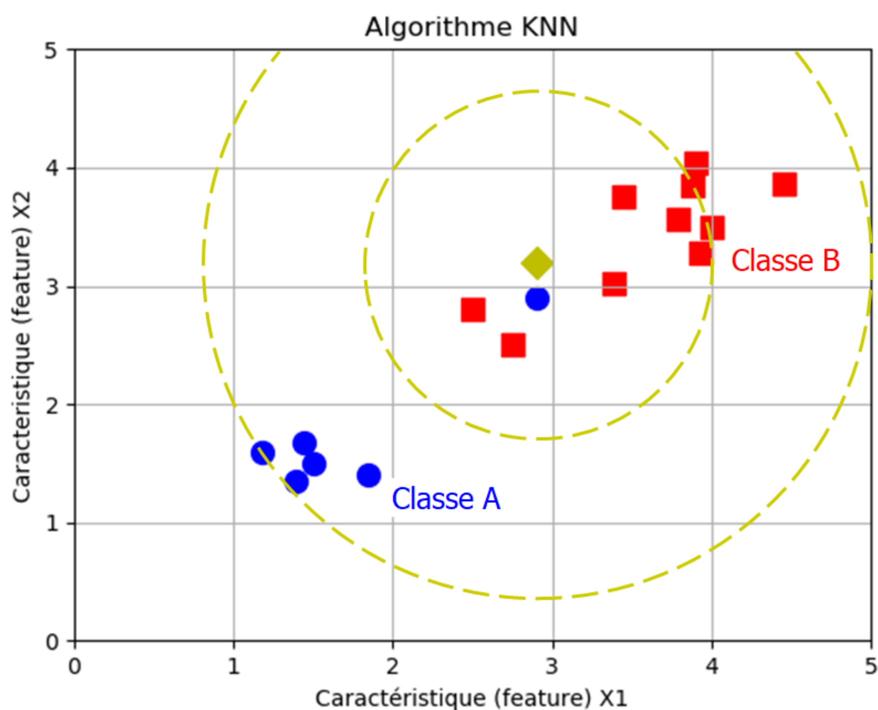
- Les lignes renseignent le nombre d'éléments de chaque classe réellement contenus dans l'échantillon de test ;
- Les colonnes renseignent le nombre d'éléments prédits dans chaque classe ;

Par exemple, dans la matrice ci-dessous :

- Sur 11 données du type 0 (Setosa), 10 ont bien été prédites (Setosa), mais une a été classée à tort dans la catégorie 1 (Versicolor) ;
- Sur 9 données de type 1 (Versicolor), 8 ont été bien prédites et une a été mal placée dans la catégorie (Virginica) ;
- Cependant, toutes les prédictions sur les données du type 2 (Virginica) s'avèrent exactes.

Remarque

L'efficacité du modèle dépend fortement de l'**hyper-paramètre K.** En effet, K vaut 1, l'algorithme ne se réfère qu'au plus proche voisin de la nouvelle observation et rend la prédiction très sensible. Au fur et à mesure que l'on augmente K, la prédiction va se stabiliser, mais s'il est trop grand on va aller « chercher » des voisins très éloignés et fausser les résultats. Donc il est fortement recommandé de chercher la valeur optimale de k.



2.5 Exemple

L'algorithme des k plus proches voisins est ici présenté pour la reconnaissance de caractères. On utilise pour cela la base de données `mnist` disponible sous Python.

2.5.1 Échantillonnages des données

- Cette base de données est composée de 70 000 images de taille 28×28 pixels
- On utilisera ici que 15% des images, soit 10 500 (pour des raisons de temps de calcul)
- Pour réaliser le code des k plus proches voisins, on utilise les fonctions de la bibliothèque `sklearn`.

FIGURE 6 – Échantillon de la base de données mnist

```
#\# Importation de la bibliothèque mnist. Cela peut prendre du temps.
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
#\# Importation d'une fonction permettant de séparer les données en deux groupes.
from sklearn.model_selection import train_test_split

vect_x_utile,vect_x_non_utile,vect_y_utile,vect_y_non_utile=
train_test_split(mnist.data, mnist.target, train_size=0.15) # 15\% des données seront utilisées
    et stockées dans vect\_x\_utile (28x28 pixels) et vect\_y\_utile le label de l'image
```

- La fonction `train_test_split` permet de séparer les données en deux groupes.
- Les images et leurs étiquettes utiles sont stockées dans les variables `vect_x_utile` et `vect_y_utile`.
- `vect_x_utile` représente les pixels de l'image. Il s'agit donc d'une matrice de taille 28×28 (en réalité, c'est un vecteur de taille 784×1) dont les valeurs sont comprises entre 0 et 255 (image en niveau de gris).
- `vect_y_utile` est l'étiquette associée à l'image. Ici, il y a 10 possibilités : $0, 1, 2, \cdots, 9$.

Pour tester la qualité du modèle des k plus proches voisins :

- 80% des données sont utilisées comme données d'entraînement. Cela signifie tout simplement que ce sont les données $\mathbf{x_i}$ connues pour la réalisation de notre algorithme des k plus proches voisins. (`vect_x_train`)
- 20% des données restantes seront les données de tests ($\mathbf{x}^*$), c'est-à-dire celles dont on va chercher à déterminer les étiquettes. Or, on connait les "vraies" étiquettes de ces données et on pourra comparer celles-ci avec les prédictions établies par l'algorithme et ainsi tester la qualité du modèle.(`vect_x_test`)
- On a donc au total $10500 \times 0.8 = 8400$ images connues (`vect_x_train`) et $10500 \times 0.2 = 2100$ images dont on va chercher à prédire le chiffre écrit (`vect_x_test`).

```
    vect_x_train, vect_x_test, etiquettes_train, etiquettes_test =
    train_test_split(vect_x_utile, vect_y_utile, train_size=0.8)
```

2.5.2 Algorithme des k plus proches voisins : Pour un nombre k fixé

```
    from sklearn import neighbors # Importation de la bibliothèque utile pour l'algorithme des k
     plus proches voisins
    knn = neighbors.KNeighborsClassifier(3) # Création du modèle. Ici on fixe k=3
```


CC BY NC SA

```
3     knn.fit(vect_x_train,vect_y_train) # Sauvegarde des données utiles : phase d'apprentissage.
4     predictions=knn.predict(vect_x_test) # Prédictions des étiquettes pour les données
      vect_x_test
5     liste_prediction_correcte=[predictions[i]==
6     etiquettes_test.values[i] for i in range(len(predictions))] # liste
7     pourcentage_erreur=(1-sum(liste_prediction_correcte)/len(liste_prediction_correcte))*100
8     # Pourcentage d'erreur dans la prédiction
```

- À la ligne 2, on crée le modèle que l'on souhaite utiliser. Ici, on va utiliser l'algorithme pour $k = 3$. Cette ligne permet d'initialiser l'algorithme.
- À la ligne 3, l'algorithme ici n'effectue aucune optimisation mais va juste sauvegarder toutes les données en mémoire. C'est sa manière d'apprendre en quelque sorte.
- À la ligne 4, on effectue les prédictions (phase d'inférence) pour chaque image de `vect_x_test`.
- À la ligne 5, on construit la liste des prédictions qui sont correctes. On effectue la création de cette liste par compréhension en vérifiant que la prédiction `prediction[i]` est bien égale à la valeur de l'étiquette `etiquettes_test.values[i]`. La liste `liste_prediction_correcte` est donc une liste de booléen `True` et `False`.
- À la ligne 7, on calcule le pourcentage d'erreur dans les prédictions faites à la ligne précédente. Pour cela, on calcule la moyenne des réussites à l'aide de la commande `sum(liste_prediction_correcte)/len(liste_prediction_correcte)`. On rappelle que `True+True=2`, le typage est automatiquement forcé, c'est pour cela que l'on peut utiliser la fonction `sum`.

Le résultat obtenu est un pourcentage d'erreur de l'ordre de 6%.

2.6 Algorithme des k plus proches voisins : Recherche de la meilleure valeur de k

```
1     # Les données vect_x_train, etiquettes_train, vect_x_test et etiquettes_test sont
      disponibles.
2     errors = []
3     for k in range(2,15):
4     knn = neighbors.KNeighborsClassifier(k) # Création du modèle
5     knn.fit(vect_x_train, etiquettes_train) # Sauvegarde des données utiles
6     errors.append(100*(1 - knn.score(vect_x_test, etiquettes_test))) # Calculs de l'erreur pour
      chaque valeur de k
7     plt.plot(range(2,15), errors, 'o-')
8     plt.xlabel("Nombre k")
9     plt.ylabel("Erreur en %")
10    plt.show()
```

On obtient alors :

Dans ce code, ce que nous n'avons pas expliqué est la méthode `score`. Cette fonction réalise ce qui a été réalisé aux **lignes 5 à 7** du code précédent. Cette méthode `score` calcule donc **le taux de réussite du classifieur**.

Remarque

On constate qu'il y a deux valeurs de k optimales, 5 ou 6. Ces valeurs sont optimales sur les échantillons d'images traités. D'ailleurs, avec d'autres échantillons les résultats sont différents. Cependant on trouve très régulièrement que $k = 5$ est le (ou l'un des) meilleur(s) résultat(s) possible(s) sur ce problème de classification.

CC BY NC SA

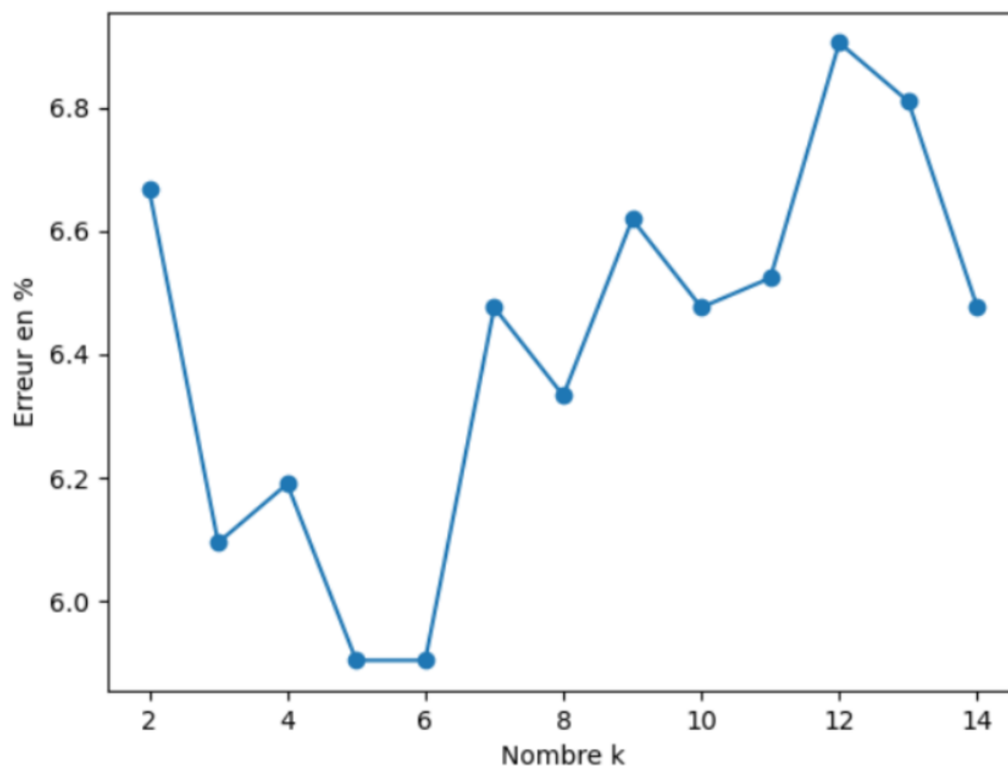


FIGURE 7 – Erreur de prédiction en fonction de k

2.7 Matrice de confusion

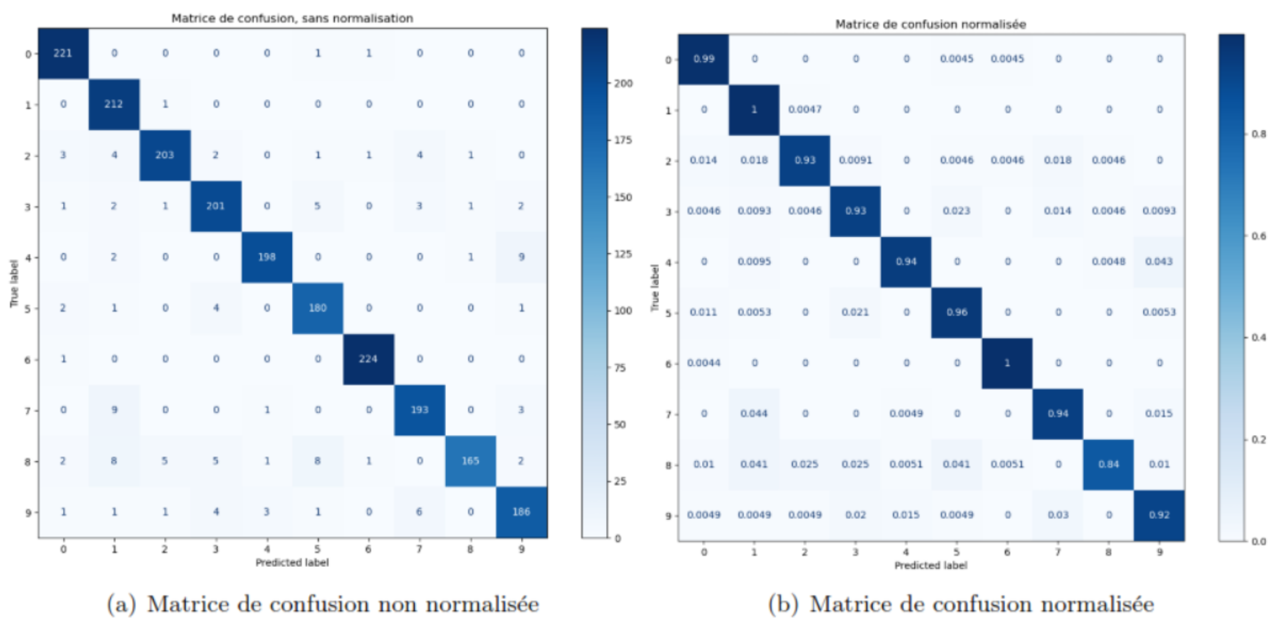
Grâce à la fonction `score`, on peut connaître la qualité de prédiction du classifieur. Cependant, il pourrait être très intéressant de savoir où les erreurs sont produites et comment. Pour cela, on va utiliser la **matrice de confusion**.

```
1  # Les données vect\_x\_train, etiquettes\_train, vect\_x\_test et etiquettes\_test sont
   disponibles.
2  knn = neighbors.KNeighborsClassifier(3) # Création du modèle
3  knn.fit(vect_x_train, etiquettes_train)
4  class_names=list(range(10)) # liste de nombre de 0 à 9
5  disp = plot_confusion_matrix(knn, vect_x_test, etiquettes_test,display_labels=
6  class_names,cmap=plt.cm.Blues,normalize=None)
7  disp.ax_.set_title("Matrice de confusion, sans normalisation")
8  print(title)
9  print(disp.confusion_matrix)
10 disp = plot_confusion_matrix(knn, vect_x_test, etiquettes_test,display_labels=
11 class_names,cmap=plt.cm.Blues,normalize='true')
12 disp.ax_.set_title("Matrice de confusion normalis\'ee")
13 print(title)
14 print(disp.confusion_matrix)
```

Dans la fonction `plot_confusion_matrix`, l'option `cmap` permet de colorer la matrice de confusion, et on peut normaliser les données ou non.

Les résultats obtenus sont les suivants :

Dans le cadre de l'évaluation des performances d'une fonction de prédiction sur un problème de ***classification multiclasse*** à t étiquettes, on souhaite disposer d'une mesure de performance permet-



(a) Matrice de confusion non normalisée (b) Matrice de confusion normalisée

FIGURE 8 – Les matrices de confusions

tant d'inspecter en détail le comportement de la fonction de prédiction et notamment ses difficulté à prédire chacune des étiquettes. La matrice de confusion est un tableau de dimension $t \times t$ qui répertorie toutes les informations concernant les prédictions effectuées et les vraies étiquettes. À chaque étiquette possible est associée un entier de 1 à t permettant de l'identifier. Le tableau se lit alors de la manière suivante :

- Chaque élément du tableau, placé à la ligne $i \in \{1, \ldots, t\}$ et à la colonne $j \in \{1, \ldots, t\}$ (noté M_{ij}) correspond au nombre de fois où la fonction de prédiction a produit la sortie j alors que la vraie étiquette était i.
- Ainsi les éléments sur la diagonale correspondent au nombre de prédictions justes (l'élément (i, i) correspond au nombre de fois où l'étiquette i a été correctement prédite).

On peut construire une matrice de confusion normalisée en divisant chaque élément par la somme des éléments sur la même ligne. En notant Mn la matrice de confusion normalisée, on a :

$$\forall i, j \in \{1, \ldots, t\},\ Mn_{ij} = \frac{M_{ij}}{\sum_{c=1}^{t} M_{ic}}.$$

Ces deux mesures de performance portent des informations différentes :

- La matrice de confusion non normalisée permet de visualiser les plus importantes sources d'erreurs. Dans le cas (fréquent) où la répartition entre les différentes étiquettes n'est pas équilibrée, les erreurs seront majoritairement contenues dans les lignes et les colonnes correspondant à ces étiquettes.
- La matrice de confusion normalisée ne tient plus compte du déséquilibre entre étiquettes en divisant chaque ligne i par le nombre d'apparition de l'étiquette i dans la base de données (c'est à dire la somme des éléments de la ligne i de la matrice de confusion non normalisée). Celle-ci

permettra d'observer les erreurs commises par la fonction de prédiction, indépendamment de la répartition des étiquettes.

2.8 Sensibilité et spécificité

Dans le cadre d'un test de prédiction d'une classe d'une donnée, on cherche à valider si celle-ci a été correctement faite ou non et à comprendre la répartition des erreurs. Prenons l'exemple de prédiction du chiffre 8 présentée dans la section précédente.

Il en résulte la matrice de confusion ci-dessous :

		nature du chiffre prédit	
		correcte	incorrecte
nature réelle du chiffre	correcte	VP	FN
	incorrecte	FP	VN

Où les valeurs du tableaux représentent

- **VP (Vrais Positifs)** est le nombre de chiffres 8 corrects effectivement reconnus comme tels,
- **FP (Faux Positifs)** représente le nombre de chiffres réellement différents de 8 mais qui ont été jugés comme étant égal à 8,
- **FN (Faux Négatifs)** représente le nombre de chiffres 8 réellement corrects mais qui n'ont pas été correctement classés,
- **VN (Vrais Négatifs)** représente le nombre de chiffres réellement différents de 8 qui ont été catégorisés comme tels.

Pour le chiffre 8, on obtient la matrice suivante :

		nature du chiffre 8 prédit	
		correcte	incorrecte
nature réelle du chiffre 8	correcte	165	32
	incorrecte	3	1900

Remarque

Les termes *positifs* et *négatifs* sont hérités du vocabulaire médical où les critères permettent de conclure sur la positivité d'un malade à une maladie. En ingéniérie, on décidera arbitrairement quelle catégorie est *positive*. Ici, il s'agit de *correcte*.

- La **sensibilité d'un critère** est la probabilité que celui-ci détecte la catégorie *positive* parmi les exemples qui le sont réellement. Il s'agit donc du rapport $\frac{VP}{VP+FN}$.
- La **spécificité d'un critère** est la probabilité que celui-ci détecte la catégorie *négative* parmi les exemples qui le sont réellement. Il s'agit donc du rapport $\frac{VN}{FP+VN}$.
- Un critère *sensible* (de sensibilité élevée) est réglé de manière à trouver tous les éléments associés à la catégorie positive, quitte à se tromper et à générer des faux positifs (FP) ce qui dégrade sa spécificité.

- Un critère *spécifique* (de spécificité élevée) est réglé de manière à trouver tous les éléments associés à la catégorie négative, quitte à générer des faux négatifs (FN) ce qui dégrade sa sensibilité.

Remarque

Avec ces notations, la justesse (*Accuracy*) vaut

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN}$$

et porte donc une information différente.

2.9 Importance de la mise en forme des données

De par les échelles de certaines données, celles-ci peuvent avoir un poids très important qui n'est pas forcément justifié. Par exemple, dans le domaine médical si on s'intéresse à la fréquence cardiaque (autour de 70 battements/min) et au taux de cholestérol (variant autour de 1,5 à 2,5 g/L). On comprend qu'une variation d'une unité sur le battement cardiaque aura une faible influence sur l'état de santé du patient. Par contre, une variation d'une unité sur le taux de cholestérol a une influence significative. Voici les données de quatre patients (ceci est bien sûr un exemple fictif) :

Numéro	Fréquence cardiaque (bat/min)	Taux de cholestérol (g/L)	État santé
1	80	1,8	Bon
2	70	2,3	Mauvais
3	65	1,6	Bon
4	90	2,1	Mauvais

Pour réaliser un algorithme des k plus proches voisins de bonne qualité, il est nécessaire que les données soient dans des unités ou à une échelle comparable. Pour cela, on retiendra principalement deux possibilités :

- Ramener les données à une échelle commune (généralement entre 0 et 1)
- Normer les données par rapport à leur écart-type (représentant la dispersion moyenne de celles-ci)

L'ensemble des données sont stockées dans une matrice M. Pour notre exemple, $M = \begin{pmatrix} 80 & 1,8 \\ 70 & 2,3 \\ 65 & 1,6 \\ 90 & 2,1 \end{pmatrix}$

Normalisation dans l'intervalle $[0, 1]$

On note $col_j(M)$ la colonne j de la matrice M. Un élément de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne est noté M_{ij}. La normalisation de la donnée M_{ij} est notée M_{ij}^{norm}. On calcule :

$$M_{ij}^{norm} = \frac{M_{ij} - \min(col_j(M))}{\max(col_j(M)) - \min(col_j(M))}$$

On trouve alors :

$$M^{norm} = \begin{pmatrix} 0,6 & 0,286 \\ 0,2 & 1 \\ 0 & 0 \\ 1 & 0,714 \end{pmatrix}$$

CC BY NC SA

Normalisation dans l'intervalle grâce à l'écart-type des données

On note $\sigma_{col_j(M)}$, l'écart-type des valeurs de la colonne $col_j(M)$ et $\overline{col_j(M)}$, la moyenne des valeurs de la colonne $col_j(M)$.

On calcule :

$$M_{ij}^{norm} = \frac{M_{ij} - \overline{col_j(M)}}{\sigma_{col_j(M)}}$$

On trouve ainsi :

$$\overline{col_0(M)} = 76,25 \qquad \sigma_{col_0(M)} = 9,6 \qquad \overline{col_1(M)} = 1,95 \qquad \sigma_{col_1(M)} = 0,27$$

De ce fait :

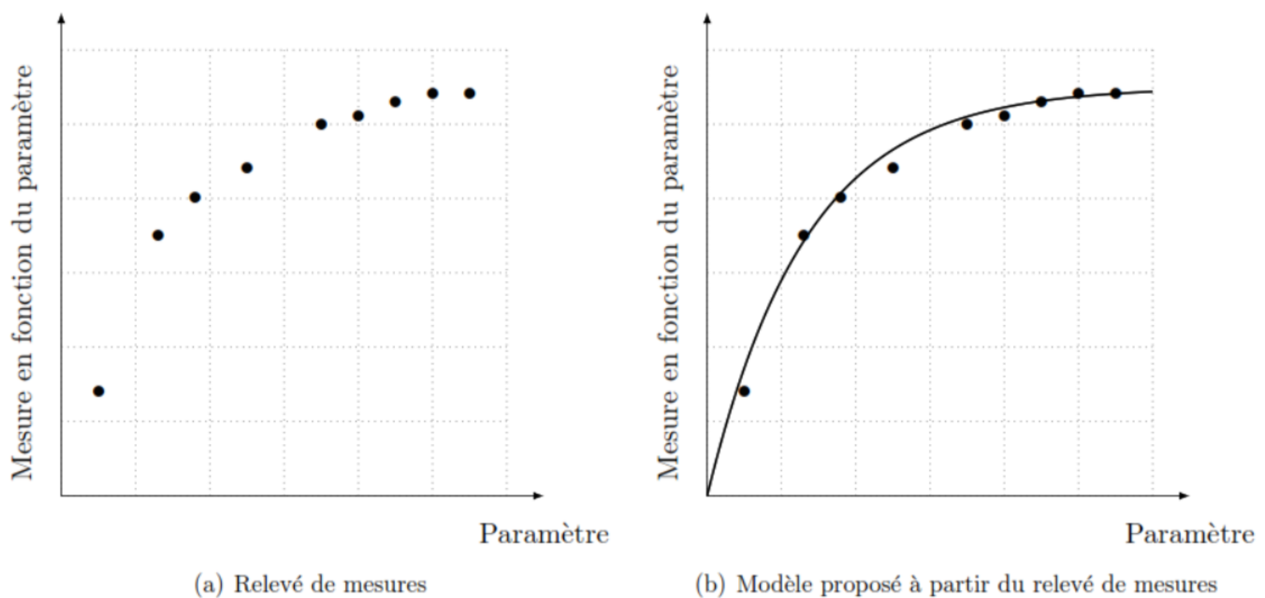
$$M^{norm} = \begin{pmatrix} 0,39 & -0,56 \\ -0,65 & 1,30 \\ -1,17 & 0-1,30 \\ 1,43 & 0,56 \end{pmatrix}$$

3 Régression linéaire

3.1 Intérêt de déterminer un modèle de données

3.1.1 Présentation

Lors d'une étude expérimentale, des points de mesures sont réalisés (en figure 9). Afin de traiter ces données, celles-ci sont alors souvent représentées graphiquement afin de mieux les visualiser.



(a) Relevé de mesures (b) Modèle proposé à partir du relevé de mesures

FIGURE 9 – Exemple (fictif) de relevé de mesures

Objectif

L'ingénieur **(ou plus largement le scientifique)** peut alors à partir de ces données tenter de définir un modèle de comportement de ce qui a été observé (figure 9). C'est-à-dire en déduire une équation liant les paramètres expérimentaux et les mesures obtenues.

Remarque

Cette démarche est une démarche tout à fait classique et que vous avez déjà probablement effectuée. Le modèle que l'on vous demandait de trouver était généralement un modèle linéaire (la droite passant au mieux par le relevé de points effectués). Cette approche déductive d'un modèle à partir d'expériences est largement répandue dans le monde scientifique.

Exemple:

Un exemple très simple et qui est abordé au collège est d'effectuer un relevé de tension aux bornes d'une résistance ainsi que l'intensité circulant dans le circuit électrique. Ce relevé de mesures de tension et intensité permet de mettre en exergue la relation $U = R.I$.

3.2 Régression linéaire simple : Mise en équations

3.2.1 Les 4 étapes indispensables en apprentissage supervisé

Pour résoudre un problème d'apprentissage supervisé, il faut procéder en 4 étapes :

- Disposer d'un Dataset $(\boldsymbol{x}, \boldsymbol{y})$ où $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{m} \times \boldsymbol{n}}$ sont les features et $\boldsymbol{y} \in \mathbb{R}^{\boldsymbol{m} \times \mathbf{1}}$ est la target
- Développer un Modèle dont les paramètres sont à déterminer
- Définir une Fonction Coût qui mesure l'erreur entre le modèle et les valeurs $y^{(i)}$ de $\boldsymbol{y}$ du Dataset
- Utiliser un Algorithme d'Apprentissage qui cherche le modèle (en réalité les paramètres) qui minimisent la Fonction Coût.

3.2.2 Le Dataset d'une Régression Linéaire simple

Dans cet exemple, on présente comment développer un modèle de Machine Learning à partir d'un Dataset à une seule variable $\boldsymbol{x_1}$ (que l'on notera simplement $\boldsymbol{x}$). On aura donc un Dataset avec $\boldsymbol{m}$ exemples et $\boldsymbol{n} = \mathbf{1}$ «feature» variable.

$\boldsymbol{y}$ (target)	$\boldsymbol{x}$ (feature)
$\boldsymbol{y}^{(\mathbf{1})}$	$x^{(1)}$
$\boldsymbol{y}^{(\mathbf{2})}$	$x^{(2)}$
...	...
$\boldsymbol{y}^{(\boldsymbol{m})}$	$x^{(m)}$

TABLE 1 – Dataset

3.2.3 Le Modèle de Régression Linéaire Simple

Prenons le nuage de point suivant, on dirait qu'il suit une tendance linéaire. Pour cela, nous allons développer un modèle linéaire de la forme $f(x) = ax + b$. Le modèle présente 2 paramètres a et b d'où l'appellation multivariable. Ces variables ne sont pas pour l'instant connues. Il est donc impossible de tracer une bonne droite sur le nuage de point, à moins de choisir des paramètres au hasard. Ce sera le rôle de la machine de déterminer ces valeurs en minimisant la fonction coût.

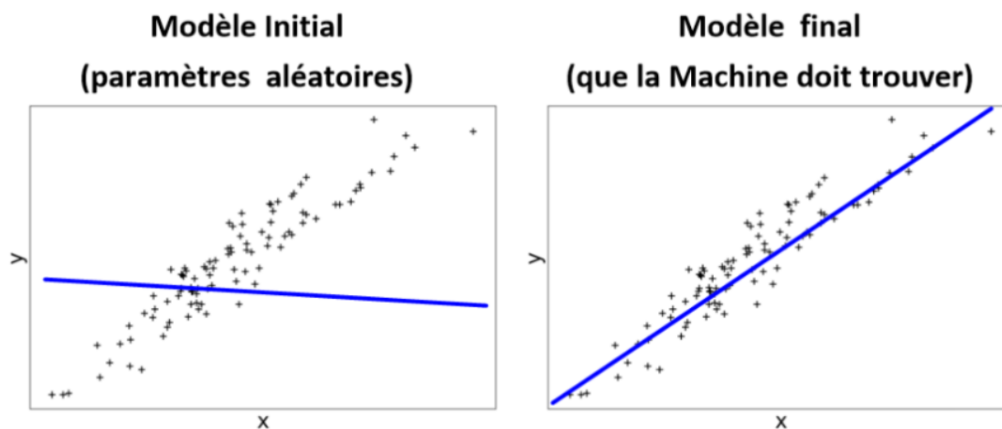


FIGURE 10 – Modèle initial et modèle finale construit par la machine

3.2.4 La Fonction Coût : L'Erreur Quadratique Moyenne (Mean Squared Error MSE)

La fonction coût $J(a, b)$ nous permet d'évaluer la performance de notre modèle en mesurant les erreurs entre $f\left(x^{(i)}\right)$ et $y^{(i)}$. Alors, pour mesurer les erreurs entre les prédictions $f\left(x^{(i)}\right)$ et les valeurs $y^{(i)}$ du Dataset, on calcule le carré de la différence : $\left(f\left(x^{(i)}\right) - y^{(i)}\right)^2$. C'est la norme euclidienne, qui représente la distance directe entre $f\left(x^{(i)}\right)$ et $y^{(i)}$

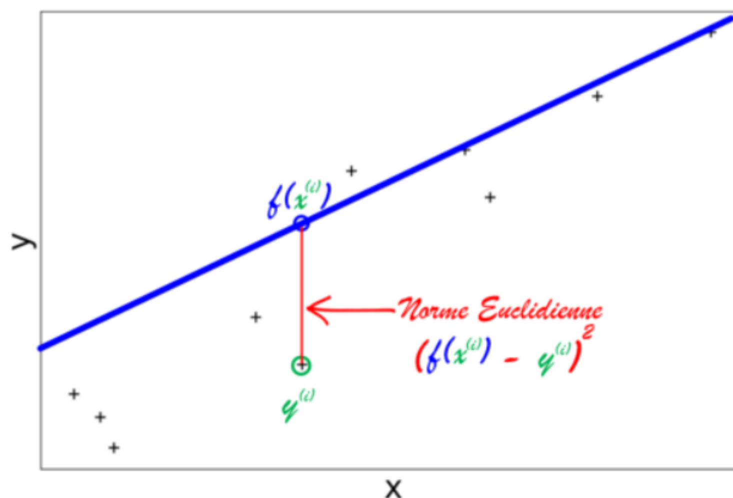


FIGURE 11 – Représentation de la norme euclidienne

CC BY NC SA

Pour la régression linéaire, la fonction Coût $J(a,b)$ est la moyenne de toutes les erreurs, soit :

$$J(a,b)=\frac{\left(f\left(x^{(1)}\right)-y^{(1)}\right)^2+\cdots+\left(f\left(x^{(m)}\right)-y^{(m)}\right)^2}{m}=\frac{1}{m}\sum_{i=1}^{m}\left(f\left(x^{(i)}\right)-y^{(i)}\right)^2$$

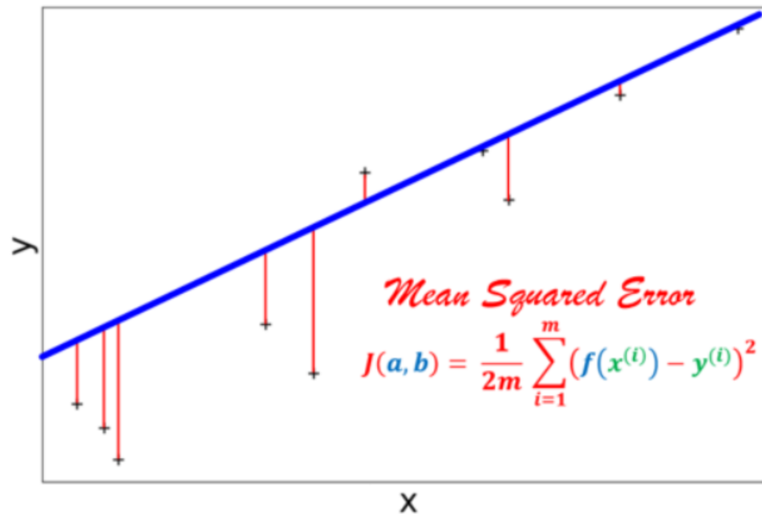


Figure 12 – la moyenne de toutes les erreurs

Pour simplifier un calcul de dérivée qui viendra après, on rajoute par convention $\frac{1}{2}$. D'où La fonction coût est donnée par :

$$J(a,b)=\frac{1}{2m}\sum_{i=1}^{m}\left(f\left(x^{(i)}\right)-y^{(i)}\right)^2=\frac{1}{2m}\sum_{i=1}^{m}\left(ax^{(i)}+b-y^{(i)}\right)^2$$

Dans le cas d'une régression linéaire, la fonction coût $J(a,b)$ qui dépend de deux paramètres a et b est bel et bien une fonction convexe. Cette propriété est très importante pour s'assurer de sa convergence vers le minimum avec par exemple l'algorithme de la descente de gradient (Gradient Descent).

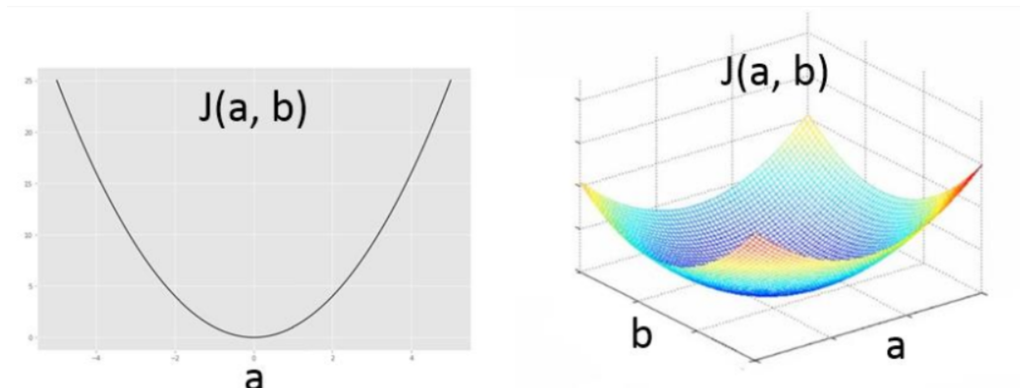


Figure 13 – Fonction cout

3.2.5 L'algorithme d'apprentissage

On dit que la machine apprend quand elle trouve quels sont les paramètres du modèle qui minimisent la fonction Coût. On cherche donc à développer un algorithme de minimisation. Il existe plusieurs méthodes de minimisation (méthode des moindres carrés, méthode de Newton, Gradient Descent, Simplex, etc.). Pour un modèle de régression linéaire, on utilise le plus souvent la méthode des moindres carrés (quand le problème est simple) et l'algorithme de Gradient Descent (pour les régressions plus compliquée). Le Gradient Descent permet de converger progressivement vers le minimum de n'importe quelle fonction convexe (comme la Fonction Coût) en suivant la direction de la pente (le gradient) qui descend, d'où son nom de Gradient Descent.

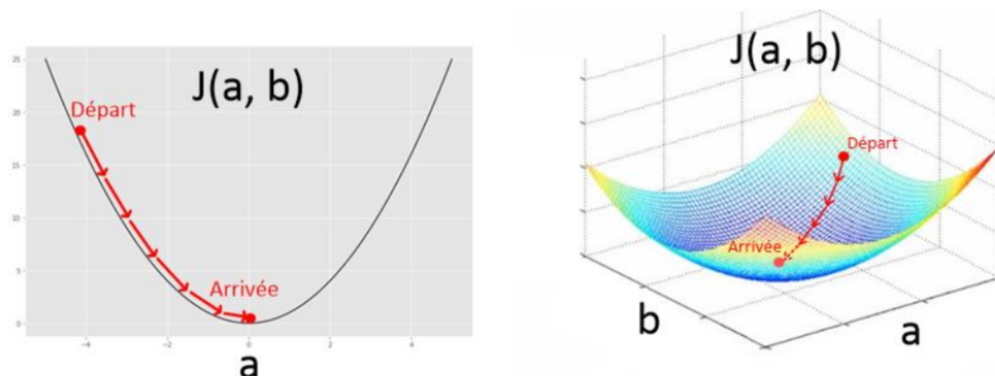


Figure 14 – algorithme de la descente de gradient gradient descent

Calcul des gradients : La fonction coût est donnée par : $J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(ax^{(i)}+b-y^{(i)}\right)^2$ Les gradients sont les suivants :

- $\frac{\partial J(a,b)}{\partial a} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(ax^{(i)}+b-y^{(i)}\right)$
- $\frac{\partial J(a,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} 1\left(ax^{(i)}+b-y^{(i)}\right)$

Algorithme de la descente de gradient à pas fixe :

- $a := a - \alpha\frac{\partial J(a,b)}{\partial a}$
- $b := b - \alpha\frac{\partial J(a,b)}{\partial b}$
- $\alpha > 0$: est le pas d'apprentissage

3.2.6 Régression linéaire : Résumé des équations :

- Dataset : (x, y) avec m exemples où $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{m\times n}}$ sont les features et $\boldsymbol{y} \in \mathbb{R}^{\boldsymbol{m\times 1}}$ est la target ;
- Modèle $f\left(x^{(i)}\right) = ax^{(i)} + b$;
- Fonction coût : $J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(ax^{(i)}+b-y^{(i)}\right)^2$
- Gradients : $\begin{cases} \frac{\partial J(a,b)}{\partial a} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(ax^{(i)}+b-y^{(i)}\right) \\ \frac{\partial J(a,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} 1\left(ax^{(i)}+b-y^{(i)}\right) \end{cases}$
- Algorithme de descente de gradient : $\begin{cases} a := a - \alpha\frac{\partial J(a,b)}{\partial a} \\ b := b - \alpha\frac{\partial J(a,b)}{\partial b} \end{cases}$

3.2.7 Application :

Soit le dataset (x, y) avec 3 exemples.

$$x = (0, 1, 2) : \text{ Feature}$$
$$y = (1, 2.3, 3.9) : \text{ Target}$$

$\boldsymbol{y}$ (target)	$\boldsymbol{x}$ (feature)
$\boldsymbol{y^{(1)} = 1,0}$	$x^{(1)} = 0$
$\boldsymbol{y^{(2)} = 2,3}$	$x^{(2)} = 1$
$\boldsymbol{y^{(3)} = 3,9}$	$x^{(3)} = 2$

TABLE 2 – Dataset

Travail demandé : à partir du dataset donné précédemment, établir :
en premier lieu un modèle de régression monovariable de type $f(x) = ax$ et en second lieu un modèle multivariable de type $f(x) = ax + b$. *Vous êtes invité à résoudre le problème analytiquement.*

3.3 Ecriture des équations sous forme matricielle

Reformuler les équations sous forme matricielle permet de calculer d'un coup le Gradient, la Fonction Coût, et le modèle sur toutes les données du Dataset. Cela permet aussi de simplifier les calculs et développer des modèles plus complexes comme la régression polynomiale.

3.3.1 Transférer le Dataset dans des matrices X, Y

En présence d'un Dataset (x, y) qui contient m exemples et n features, on transfert toutes les données $y^{(i)}$ dans un vecteur $\boldsymbol{Y}$ de dimension $(m \times 1)$ et toutes les données $x^{(i)}$ dans une matrice $\boldsymbol{X}$ à laquelle on ajoute une colonne biais (c'est une colonne remplie de «1»). La colonne biais est de grande importance pour rendre le calcul matriciel faisable.

- La matrice $\boldsymbol{X}$ est de dimension $(m \times (n + 1))$, soit $(m \times 2)$ dans notre cas.
- Dans le cas de la régression linéaire à une seule variable $x^{(i)}$, on a $n = 1$ et donc la matrice $\boldsymbol{X}$ est de dimension $(m \times 2)$.

$$Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}, \quad X = \begin{bmatrix} x^{(1)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix}$$

3.3.2 Le modèle $F = X.\theta$

Il est à noter que le modèle linéaire $f\left(x^{(i)}\right) = ax^{(i)} + b$ ne permet que d'effectuer le calcul d'une prédiction à la fois, selon une valeur $x^{(i)}$ unique. Pour remédier à ce problème, il suffit de créer un vecteur $\boldsymbol{F}$ de dimension $(m \times 1)$ qui contient toutes les prédictions :

CC BY NC SA

$$F = \begin{bmatrix} f\left(x^{(1)}\right) \\ \vdots \\ f\left(x^{(m)}\right) \end{bmatrix}$$

Pour calculer ce vecteur, on utilise la formule suivante : $\boldsymbol{F} = \boldsymbol{X}.\boldsymbol{\theta}$ où $\boldsymbol{\theta} = \begin{bmatrix} a \\ b \end{bmatrix}$ est le vecteur paramètre qui contient tous les paramètres de notre modèle. On a donc le produit matriciel suivant :

$$\boldsymbol{F} = \boldsymbol{X}.\boldsymbol{\theta} = \begin{bmatrix} x^{(1)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax^{(1)} + b \\ \vdots \\ ax^{(m)} + b \end{bmatrix} = \begin{bmatrix} f\left(x^{(1)}\right) \\ \vdots \\ f\left(x^{(m)}\right) \end{bmatrix}$$

Remarque

la colonne biais de la matrice $\boldsymbol{X}$ permet donc d'effectuer le produit matriciel $\boldsymbol{X}.\ \boldsymbol{\theta}$. On dit également que le paramètre b est un paramètre biais.

3.3.3 La Fonction Coût $J(\theta)$

La fonction coût utilisée pour la régression linéaire est l'erreur quadratique moyenne (Mean Squared Error MSE) :

$$J(a,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(ax^{(i)} + b - y^{(i)}\right)^2$$

- $ax^{(i)} + b$ peut s'écrire sous la forme $\boldsymbol{X}.\boldsymbol{\theta} = \begin{bmatrix} ax^{(1)} + b \\ \vdots \\ ax^{(m)} + b \end{bmatrix}$ de dimension $(m \times 1)$
- $y^{(i)}$ peut s'écrire sous la forme d'un vecteur $\boldsymbol{Y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$ de dimension $(\boldsymbol{m} \times \mathbf{1})$
- Les vecteurs $\boldsymbol{Y}$ et $\boldsymbol{X}.\ \boldsymbol{\theta}$ sont de dimensions $(\boldsymbol{m} \times 1)$, on peut donc effectuer le calcul $\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y}$.
- Puis, on prend le carré de chaque élément du vecteur $\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y}$, puis on fait la somme de chacun de ces éléments, avant de diviser le tout par $\frac{1}{2m}$.
- La forme matricielle de la fonction coût est alors : $\boldsymbol{J}(\boldsymbol{\theta}) = \frac{1}{2m}(\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y})^2$

Remarque

J n'est ni une matrice, ni un vecteur. C'est un scalaire (un nombre). A la limite, on pourrait dire que l'on cherche à écrire $J(\theta)$ comme une matrice de dimension $(\mathbf{1} \times \mathbf{1})$, c'est-à-dire une matrice ne contenant qu'un nombre.

3.3.4 Les Gradients

- Les gradients sont déterminés comme suit : $\begin{cases} \frac{\partial J(a,b)}{\partial a} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\left(ax^{(i)}+b-y^{(i)}\right) \\ \frac{\partial J(a,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} 1\left(ax^{(i)}+b-y^{(i)}\right) \end{cases}$
- Il est possible de rassembler le calcul de ces 2 gradients en un vecteur gradient : $\frac{\partial J(\theta)}{\partial \theta} = \begin{bmatrix} \frac{\partial J(a,b)}{\partial a} \\ \frac{\partial J(a,b)}{\partial b} \end{bmatrix}$ de dimension $(n+1)\times 1$;
- Pour obtenir ce vecteur on écrit sous la forme matricielle la formule suivante :

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}X^T.(X.\theta - Y)$$

En effet :

- $X = \begin{bmatrix} x^{(1)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix}$ soit $X^T = \begin{bmatrix} x^{(1)} & \dots & x^{(m)} \\ 1 & \dots & 1 \end{bmatrix}$;
- $\left(ax^{(i)}+b-y^{(i)}\right)$ est commun aux deux gradients et s'écrit sous forma matricielle $(\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y})$;
- $\frac{\partial J(\theta)}{\partial \theta} = \begin{bmatrix} \frac{\partial J(a,b)}{\partial a} \\ \frac{\partial J(a,b)}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}^{(i)}\left(ax^{(i)}+b-y^{(i)}\right) \\ \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{1}\left(ax^{(i)}+b-y^{(i)}\right) \end{bmatrix} = \frac{1}{m}X^T \cdot (X\cdot\theta - Y).$

3.3.5 La Descente de Gradient

L'algorithme de la Descente de Gradient met à jour les paramètres a et b de notre modèle de façon itérative avec 2 lignes de calculs :
Mais à présent que nous avons créé un vecteur paramètre $\boldsymbol{\theta} = \begin{bmatrix} a \\ b \end{bmatrix}$ et un vecteur gradient $\frac{\partial J(\theta)}{\partial \theta}$ il est possible de résumer l'algorithme de Descente de Gradient sur une seule ligne soit :
On a :

$$\theta = \theta - \alpha\frac{\partial J(\theta)}{\partial \theta}$$

Remarque

les dimensions de θ sont évidemment préservées : $(n+1)\times 1$

3.3.6 La Vraie Régression Linéaire : résumé

En exprimant la Régression Linéaire sous forme matricielle, nous connaissons désormais les équations telles qu'elles sont réellement implémentées dans un programme de Machine Learning :
Pour développer un modèle linéaire (ou polynomial !) avec la descente de gradient, il faut implémenter les 4 fonctions clefs suivantes :

- **La fonction de notre modèle** : $\boldsymbol{F} = \boldsymbol{X}.\boldsymbol{\theta}$

- **La fonction Cout** : $J(\theta) = \frac{1}{2m}(\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y})^2$
- **Le gradient** : $\frac{\partial(\theta)}{\partial\theta} = \frac{1}{m}X^T.(X.\theta - Y)$
- **La descente de gradient** : $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\alpha}\frac{\partial(\theta)}{\partial\theta}$

3.3.7 Régression Linéaire Python : Bibliothèque Numpy

Dans cette partie, nous allons développer un **algorithme de descente de gradient** pour résoudre un problème de **régression linéaire** avec **Python** et sa librairie **Numpy**. Dans la pratique, les Data Scientists utilisent le **package sklearn**, qui permet d'écrire un tel code en 4 lignes, mais ici nous écrirons chaque fonction mathématique de façon explicite, ce qui est un très bon exercice pour améliorer la compréhension du **Machine Learning**.

Importer les packages Numpy et Matplotlib.pyplot :

Avant toute chose, il est nécessaire d'importer les **packages Numpy** et **Matplotlib.pyplot**.

```
1    import numpy as np
2    import matplotlib.pyplot as plt
```

Génération d'un dataset (x, y) linéaire :Figure 15

```
1    from sklearn.datasets import make_regression
2    x,y = make_regression(n_samples=100, n_features=1, noise=10)
3    print(x.shape)
4    print(y.shape)
5    y=y.reshape(y.shape[0],1)
6    print(y.shape)
7    plt.scatter(x, y) # afficher les résultats. X en abscisse et y en ordonnée
8    plt.xlabel('x')
9    plt.ylabel('y')
10   plt.title('dataset(x,y)')
11   plt.show()
```

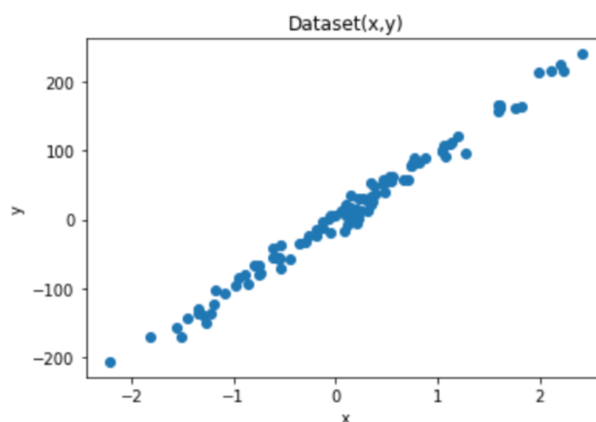


FIGURE 15 – dataset (x, y) linéaire

Une fois le Dataset est généré, il faut ajouter une colonne de biais au tableau X, c'est-à-dire une colonne de 1 , pour le développement du futur modèle linéaire $f(x) = a \times x^{(i)} + b \times 1$, puis initialiser des paramètres a, b dans un vecteur theta.

```
# ajout de la colonne de biais a X
X = np.hstack((x, np.ones(x.shape)))
print(X.shape)
# création d'un vecteur parametre theta
theta = np.random.randn(2, 1)
print(theta)
```

Développement des fonctions de la Descente de gradient :

Pour développer un modèle linéaire (ou polynomial !) avec la descente de gradient, il faut implémenter les 4 fonctions suivantes :

- La fonction de notre modèle : $\boldsymbol{F} = \boldsymbol{X}.\ \boldsymbol{\theta}$
- La fonction Cout : $\boldsymbol{J}(\boldsymbol{\theta}) = \frac{1}{2m}(\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y})^2$
- Le gradient : $\frac{\partial(\theta)}{\partial\theta} = \frac{1}{m}\boldsymbol{X}^T.(\boldsymbol{X}.\boldsymbol{\theta} - \boldsymbol{Y})$
- La descente de gradient : $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\alpha}\frac{\partial(\theta)}{\partial\theta}$

```
### 3. Développement des fonctions de Descente de gradient
def model(X, theta):
return X.dot(theta)

def cost_function(X, y, theta):
m = len(y)
return 1/(2*m) * np.sum((model(X, theta) - y)**2)

def grad(X, y, theta):
m = len(y)
return 1/m * X.T.dot(model(X, theta) - y)

def gradient_descent(X, y, theta, learning_rate, n_iterations):
# création d'un tableau de stockage pour enregistrer l'évolution du Cout du modèle
cost_history = np.zeros(n_iterations)

# mise a jour du parametre theta (formule du gradient descent)

for i in range(0, n_iterations):
theta = theta - learning_rate * grad(X, y, theta)
# on enregistre la valeur du Cout à l itération i dans cost_history[i]

cost_history[i] = cost_function(X, y, theta)
return theta, cost_history
```

Entrainement du modèle :

Une fois les fonctions ci-dessus implémentées, il suffit d'utiliser la fonction **gradient_descent** en indiquant un nombre d'itérations ainsi qu'un **learning rate** (valeur de α), et la fonction retournera les paramètres du modèle après entrainement, sous forme de la variable theta_final. Vous pouvez ensuite visualiser votre modèle grâce à Matplotlib.

```python
### 4. Entrainement du modèle
n_iterations = 100
learning_rate = 0.01
theta_final, cost_history = gradient_descent(X, y, theta, learning_rate, n_iterations)
print(theta_final) # voici les paramètres du modèle une fois que la machine a été entrainée
# création d'un vecteur prédictions qui contient les prédictions de notre modèle final
predictions = model(X, theta_final)
# Affiche les résultats de prédictions (en rouge) par rapport à notre Dataset (en bleu)
plt.scatter(x, y)
plt.plot(x, predictions, c='r')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Dataset(x,y)')
plt.show()
```

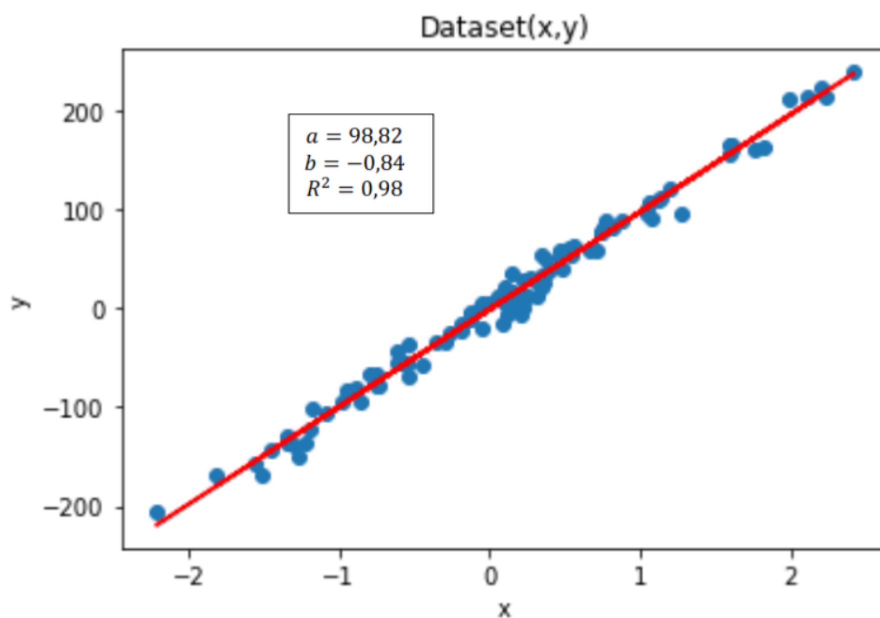


FIGURE 16 – Entrainement du modèle

Pour finir, vous pouvez visualiser l'évolution de la descente de gradient en créant un graphique qui trace **la fonction_cout** en fonction du nombre d'itération. Si votre descente de gradient a bien fonctionné, vous devez obtenir une courbe qui diminue progressivement jusqu'à converger vers un certain minimum (pas nécessairement 0). Si vous n'observez pas de stabilisation, alors cela signifie que le modèle n'a pas terminé son apprentissage et qu'il faut soit augmenter le nombre d'itérations de la descente de gradient ou bien modifier le pas d'apprentissage (learning_rate).

```python
plt.plot(range(n_iterations), cost_history)
plt.xlabel("Nombre d'itération")
plt.ylabel('Fonction Coûot')
plt.title("Evolution de la fonction Coûot")
plt.show()
```

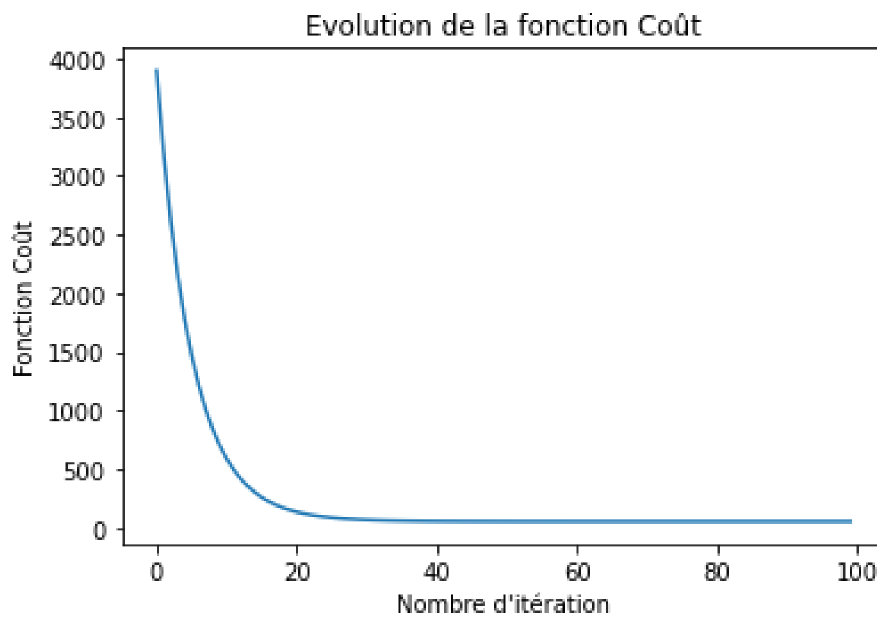


Figure 17 – Évolution de la fonction cout

Coefficient de détermination R^2

En statistique, le coefficient de détermination linéaire, noté $\boldsymbol{R}^2$, est une mesure de la qualité de la prédiction d'une régression linéaire. Il est défini par :

$$R^2 = 1 - \frac{\sum_{i=1}^{m}\left(y^{(i)} - f\left(x^{(i)}\right)\right)^2}{\sum_{i=1}^{m}\left(y^{(i)} - \bar{y}\right)^2}$$

Plus R^2 est proche de 1 plus le modèle est bon. Déterminons une fonction qui permettra de déterminer le coefficient de détermination afin d'évaluer la qualité de la prédiction.

```
def coef_determination (y, y_pred):
u=((y-y_pred)**2).sum()
v=((y-y.mean())**2).sum()
return 1-u/v
print(coef_determination(y, predictions))
```

4 Réseaux de Neurones (Neural Network)

4.1 Modèle du neurone biologique :

De façon très réductrice, un neurone biologique est une cellule qui se caractérise par :

- des synapses, les points de connexion avec les autres neurones, fibres nerveuses ou musculaires ;
- des dentrites ou entrées du neurones ;
- les axones, ou sorties du neurone vers d'autres neurones ou fibres musculaires ;
- le noyau qui active les sorties en fonction des stimulations en entrée.

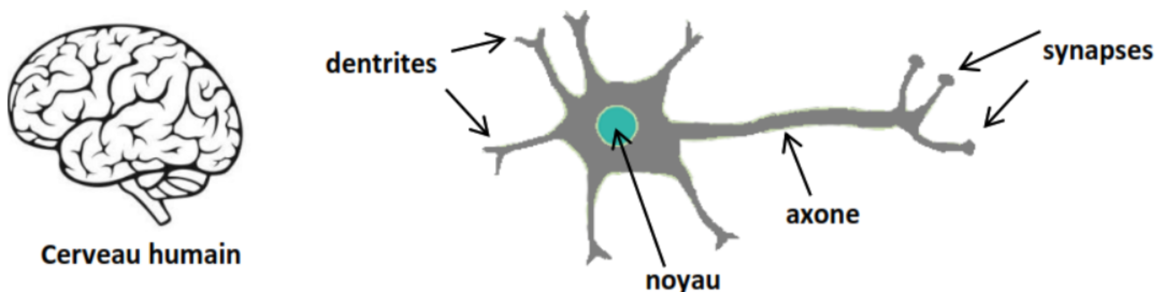
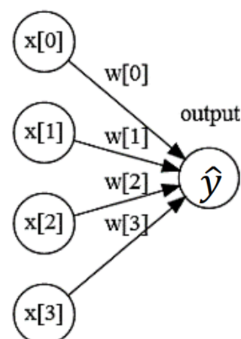


Figure 18 – Modèle neurone biologique

4.2 Modèle d'un neurone artificiel : Perceptron

Quand on réalise une régression linéaire, nous avons vu qu'on affecte un poids $w[i]$ à chaque caractéristique $X[i]$ afin d'obtenir la cible $\hat{y}$. Ceci peut être représenté selon le schéma ci-dessous.



L'idée est de répéter le processus plusieurs fois. On obtient un réseau de neurones artificiel qui se modélise par un modèle emprunté à la théorie des graphes. Cette représentation sous forme de nœuds et d'arcs est appelée perceptron.

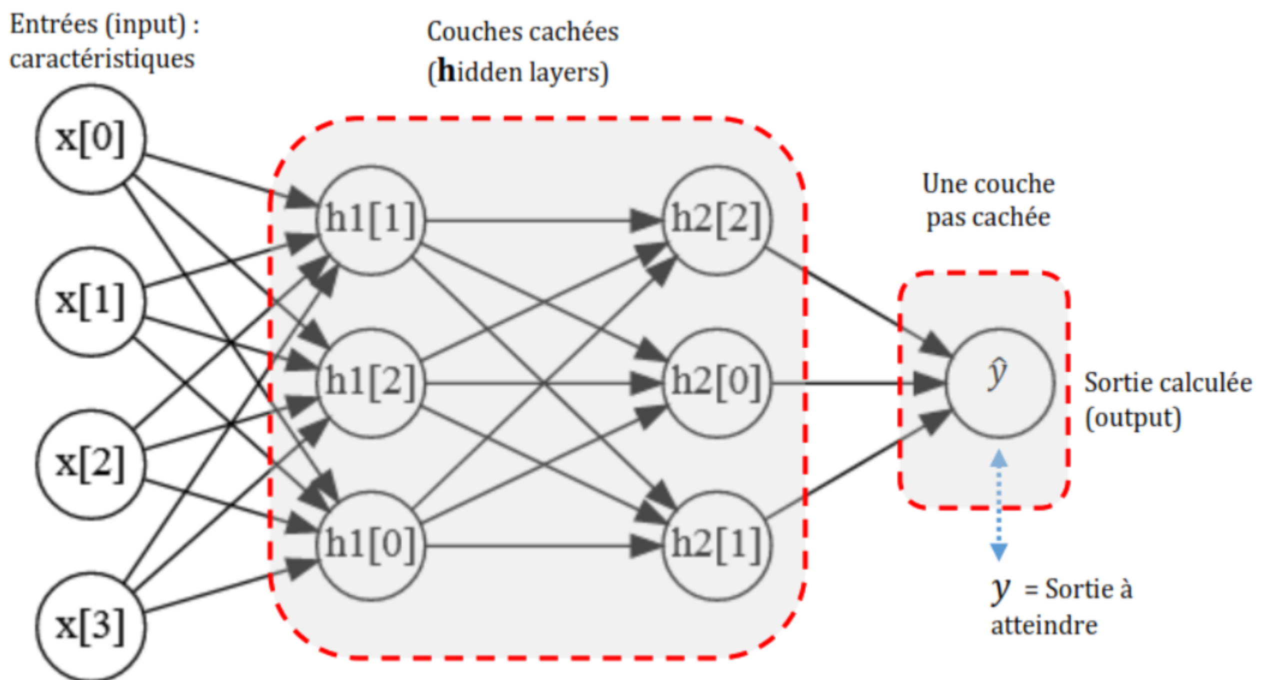


Figure 19 – Réseau de neurones artificiel (perceptron)

Remarque

$\hat{y}[X]$ avec le chapeau ^, est la cible calculée par le modèle. y sans le chapeau est la valeur réelle à atteindre.

4.2.1 Fonctionnement

Le réseau de neurones ci-dessus est un **perceptron** à **trois couches** dont deux **couches cachées** et **six neurones**. On parle de **perceptron multicouche** (MLP, Multi Layer Perceptron). Les couches cachées (hidden layers) sont les étapes intermédiaires qui vont ajouter de la non linéarité au modèle.

On applique au résultat de la somme pondérée une fonction non linéaire appelée **fonction d'activation**. On utilise les fonctions d'activation classiques :

- Tangente hyperbolique, $\tanh(x)$
- Unité de rectification linéaire RELU($g(x) = \max(0, x)$
- Sigmoïde $\text{sig}(x) = \frac{1}{1+e^{-x}}$ (aussi appelée fonction logistique)

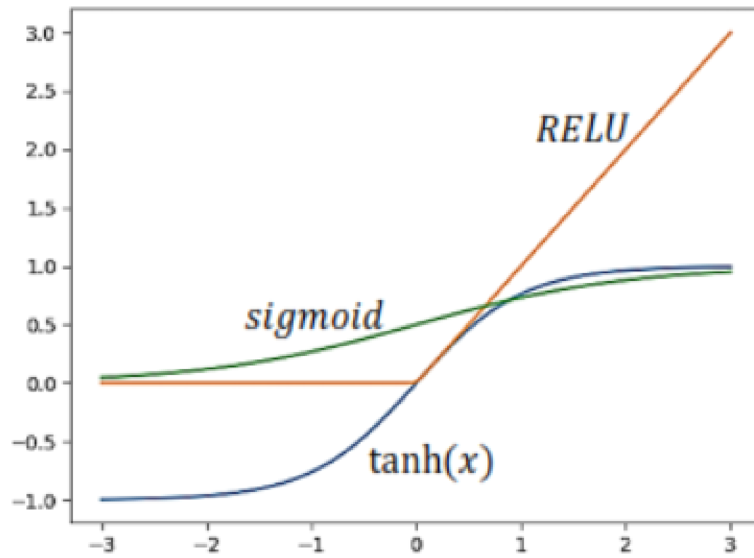


Figure 20 – Représentations des fonctions d'activation classiques

Pour bien distinguer le calcul on représente chaque neurone dans le réseau, de manière plus détaillée, de la manière suivante :

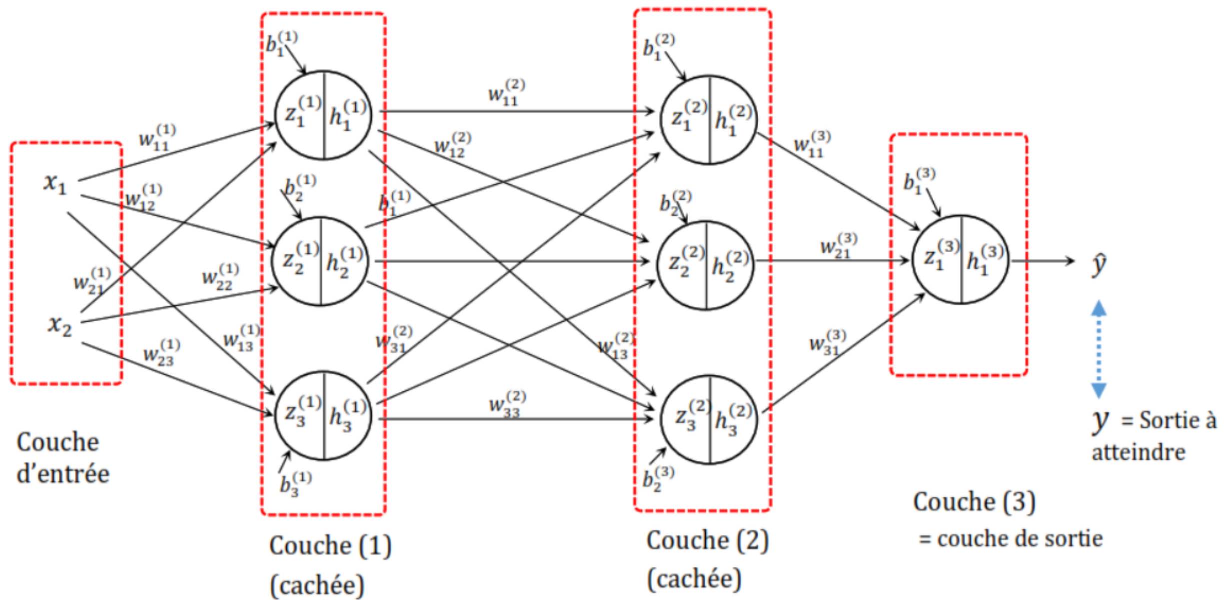


Figure 21 – Réseau de neurones artificiel en détaille

Plusieurs nombres caractérisent chaque nœud :

- La valeur numérique b est le biais du nœud et les valeurs w entrant sont les poids.
- z est la valeur à l'entrée du neurone, h est la valeur en sortie après activation.
- Le neurone est repéré par sa couche (c) et son numéro dans la couche.

Exemple:

Analysons le neurone n°3 de la couche (2) :

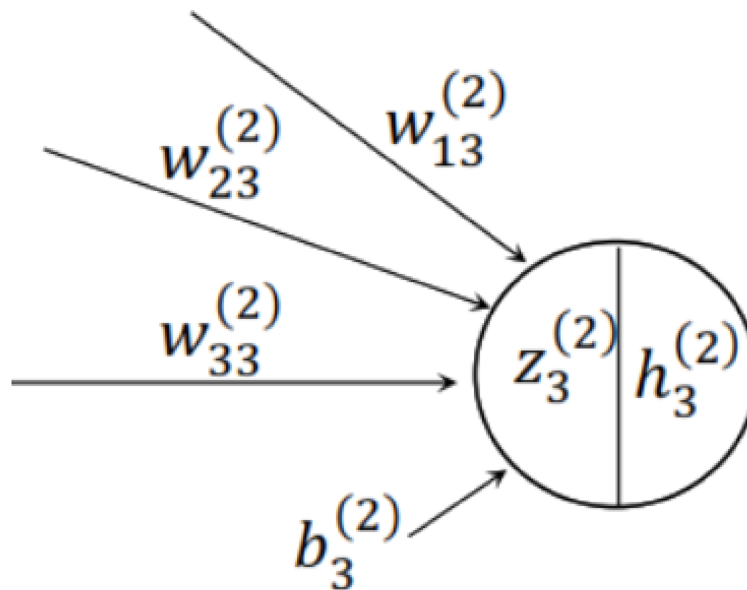


FIGURE 22 – Le neurone n°3 de la couche (2)

- $w_{13}^{(2)}$ est le poids associé au neurone 3 de la couche (2) et provenant du neurone 1 de la couche précédente.
- $b_3^{(2)}$ est le biais associé au neurone 3 de la couche (2).
- $z_3^{(2)}$ est l'entrée calculée du neurone 3 de la couche (2). Donc... avant l'activation.
- $h_3^{(2)}$ est la sortie du neurone après activation.

Pour ce neurone, la sortie $h_3^{(2)}$ se calcule donc en deux temps.

D'abord l'entrée z en fonction des sorties de la couche précédente :

$$z_3^{(2)} = \boldsymbol{w}_{13}^{(2)} \times h_1^{(1)} + \boldsymbol{w}_{23}^{(2)} \times h_2^{(1)} + \boldsymbol{w}_{33}^{(2)} \times h_3^{(1)} + \boldsymbol{b}_3^{(2)}$$

Ensuite l'application de la fonction d'activation, sigmoïde par exemple :

$$h_3^{(2)} = \text{sig}\left(z_3^{(2)}\right)$$

- Etape 1 : On initialise le réseau avec des poids $w_{ij}^{(c)}$ et biais $b_j^{(c)}$ aléatoires pour initialiser la démarche.

- Etape 2 : on calcule les sorties $h_j^{(c)}$ avec ces poids initiaux $\Rightarrow$ on se déplace vers l'avant du réseau (forward propagation).
- Etape 3 : on compare la valeur de sortie obtenue $\hat{y}$ à la sortie connue y => calcul d'erreur.
- Etape 4 : on calcule l'erreur en sortie de chaque neurone en repartant vers l'arrière (rétroaction : back propagation).
- Etape 5 : on modifie alors les paramètres $w_{ij}^{(c)}$ et $b_j^{(c)}$ afin de minimiser les erreurs avec la descente de gradient.
- Etape 6 : on réitère jusqu'à la convergence offrant une erreur minimale.

Après **plusieurs itérations** de marche avant/rétroaction, la fonction coût J se stabilise et atteint un minimum asymptotique au-dessous de laquelle elle ne descend pas. L'IA a alors atteint sa performance optimale.

Il faut donc calculer autant de fonctions coût J et ses dérivées partielles qu'il y a de neurones et de paramètres $w_{ij}^{(c)}$ et $b_j^{(c)}$ (! on utilise la descente de gradient.)

4.2.2 Et en Python ?..

La programmation en langage **Python** d'un algorithme de prédiction par réseau de neurone est tout à fait abordable pour un étudiant de 2me année de prépa scientifique, mais au prix de quelques heures de travail et d'une maîtrise parfaite du calcul matriciel. Heureusement la bibliothèque Scikit-learn possède des instructions incluant l'algorithme de prédiction par réseau de neurones.

L'instruction **MLPClassifier (Multi Layer Perceptron Classifier)** permet de créer très facilement un modèle d'apprentissage en réseau de neurones.

Si dessous pour un réseau à 2 couches cachées dont la première possède 20 neurones, et la deuxième possède 10 neurones avec un jeu de donnés d'entrainement $X_$**train,** y_train.

```
1        from sklearn.neural_network import MLPClassifier
2        mlp = MLPClassifier(solver=' lbfgs ', random_state $=0$, hidden_layer_sizes=[20, 10])
3        mp_ly
4        mlp.fit(X_train, y_train
5
```

Au lancement du code, MLPClassifier lancera une série de forward propagations et back propagations qui donne une performance optimisée de la prédiction sur le jeu de données **_train, y_train**.