**C. Description de l’approche :**

1. **Généralités sur les réseaux de neurones :**

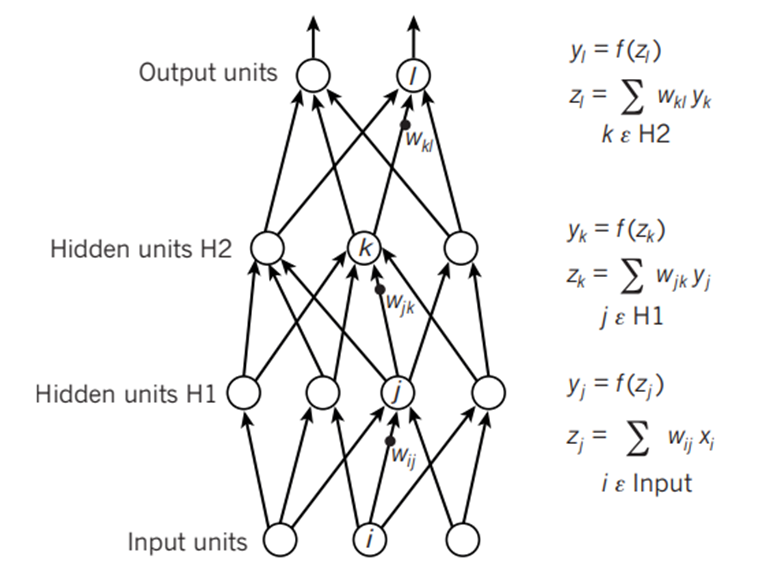
Les réseaux de neurones ont été inspirés des neurones biologiques du cerveau humain. Il s'agit d’un ensemble de neurones formels interconnectés dont la structure permet la résolution de problèmes complexes.

Cette structure peut être définie comme une succession de couches dont chacune prend ses entrées de la sortie de la couche précédente et où chaque couche est composée d’un nombre précis de neurones, on parle de :

**Couche d’entrée (Input layer)** : qui est la première couche qui reçoit en entrée les données.

**Couche de sortie (output layer)** : qui est la dernière couche qui renvoie le résultat.

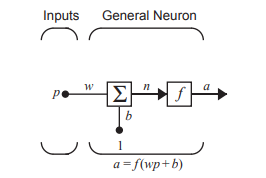
**Couches cachées (hidden layer)** : qui sont toutes les couches se trouvant entre la couche d’entrée et la couche de sortie.

**Poids (weights) :** Les liens entre les neurones.

*Figure 1 : Réseau de neurones*

Typiquement dans les problèmes de classification on utilise des réseaux de neurones à une ou 2 couches cachées, cela s'avère suffisant.

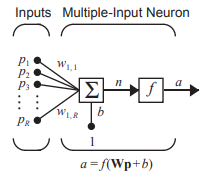
**Structure de fonctionnement d’un neurone :**



L'entrée **p** est multipliée par le poids **w** et on ajoute ensuite le biais **b**. Le résultat **n** passe par une fonction d’activation qui produit la sortie du neurone, **a**.

*Figure 2: Structure de fonctionnement d’un*

*neurone avec une seule entrée*



Généralement on a plusieurs entrées dans un seul neurone, on aura donc :



Ou plus simplement écrit:

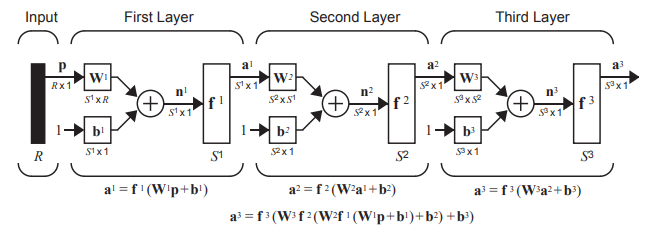


Ce qui nous donne:

*Figure 3 : Structure de fonctionnement d’un*

*neurone avec plusieurs entrée*

Pour un réseau de neurones à 4 couches cachées on aura donc une structure comme celle-ci :



**Les fonctions d’activation :**

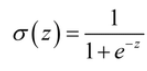
La fonction d’activation peut être une fonction linéaire ou non linéaire. Une fonction d’activation particulière est choisie pour satisfaire une spécification du problème que le neurone tente de résoudre.

Typiquement, une fonction d'activation non linéaire différentiable est utilisée dans les couches cachées d'un réseau de neurones. Cela permet au modèle d'apprendre des fonctions plus complexes qu'un réseau formé à l'aide d'une fonction d'activation linéaire.

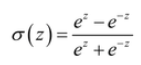
“Afin d'accéder à un espace d'hypothèses beaucoup plus riche qui bénéficierait de représentations profondes, vous avez besoin d'une fonction de non-linéarité ou d'activation.” — Page 72, Deep Learning with Python, 2017.

Il y a trois fonctions d'activation qui sont très utilisées pour les couches cachées :

* Rectified Linear Activation (**ReLU**): f(z)=max(0,z)

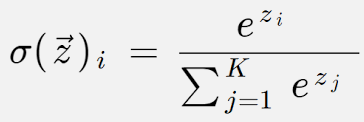


* Logistic (**Sigmoid**):



* Hyperbolic Tangent (**Tanh**) :

La fonction ReLU reste la plus utilisée car elle est à la fois simple à mettre en œuvre et efficace pour surmonter les limitations d'autres fonctions d'activation auparavant populaires, telles que Sigmoid et Tanh.

On a enfin **la fonction softmax** qui est utilisée au niveau de la couche de sortie afin de normaliser le vecteur de sortie, c'est-à-dire pour avoir des valeurs entre 0 et 1. Elle est définie ainsi : 

1. **Structure choisie :**

**Nombre de couches :** 4

Couche d’entrée : cette couche aura 13 nœuds, puisqu'on a 13 attributs pour chaque personne. Chaque nœud d’entrée recevra donc un attribut.

Couche cachée 1 : elle aura 10 nœuds. On utilisera la fonction d’activation ReLU.

Couche cachée 2 : cette couche cachée aura 2 nœuds puisqu’on vise à avoir par la suite 2 classes. On utilisera la fonction d’activation ReLU.

Couche de sortie : Cette couche aura également 2 nœuds qui contiendrait la probabilité finale d'appartenance à l’une des 2 classes. On utilisera une fonction d’activation softmax.

Les valeurs des poids (w) et des biais (b) sont choisies aléatoirement.

1. **Entrainement :**

L'entraînement est composé de 3 parties principales :

* Forward propagation : elle consiste à propager l'entrée vers l'avant via le réseau exactement comme décrit dans le fonctionnement d’un neurone.

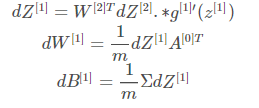
La couche d’entrée ne contient que les entrées donc :

Ensuite chaque couche qui suivra utilisera les résultats de la couche précédente pour calculer ses propres résultats : 

Ainsi de suite jusqu’à arriver à la couche finale qui contiendra les résultats.

* Backward propagation : Après avoir terminé une propagation vers l’avant des données (forward propagation), on compare les résultats obtenus de la dernière couche avec les résultats qu’on devrait avoir : 

On calcule l’erreur relative au : poids (dW), biais (dB) et fonction d’activation g (dZ) ( ici les numéro 0, 1 et 2 indiquent l’ordre des couches, la couche 0 vient avant la couche 1)



A: matrice des valeurs de la couche 0

W: matrice des poids

g: fonction d’activation

* Mise à jour des paramètres : on met à jour les paramètres comme suit :

Paramètre = paramètre- alpha\*(estimation de l’erreur relative au paramètre)

**D. Prétraitement des données:**

Les données sont initialement disponibles sur un fichier csv. Les lignes sont relatives à chaque personne et les colonnes représentent les attributs. Les données ont les dimensions (303 x 14)

On importe les données à partir du fichier csv et on les transforme en une matrice bidimensionnelle (303 x 14) pour pouvoir les manipuler.

On mélange les données aléatoirement avant de procéder à les diviser en 2 parties : 20% pour les tests et 80% pour l'entraînement. Pour chacune des 2 matrices obtenues, on calcule sa transposée (pour les manipuler plus facilement) puis on isole le vecteur contenant le label du résultat attendu.

On obtient :

* Matrice d'entraînement ( 13 x 243 ) , vecteur label ( 1 x 243 )
* Matrice de test ( 13 x 60 ) , vecteur label (1 x 60 )

Les lignes de chaque matrice représentent les attributs, on a donc chaque colonne qui représente une personne ce qui convient comme entrée au réseau de neurones.

**Pour le Code :**

On commence par importer les données grâce à la bibliothèque **pandas**. On utilise la fonction **read\_csv(‘file.csv’).**

On effectue les opérations nécessaires (décrites ci- dessus ) à la préparation des données grâce à la bibliothèque **numpy**. Pour la transformation des données en matrice on utilise la fonction **array(),** pour le mélange aléatoire la fonction **random.shuffle()** puis pour le calcul de la transposée d’une matrice il suffit d’ajouter **.T** après le nom de la matrice.

**E. Implémentation :**

**1. Importer les librairies nécessaires :**

On commence par importer les librairies :

numpy : pour la manipulation des vecteurs

pandas : pour importer les données

matplotlib : pour les visualisations

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt

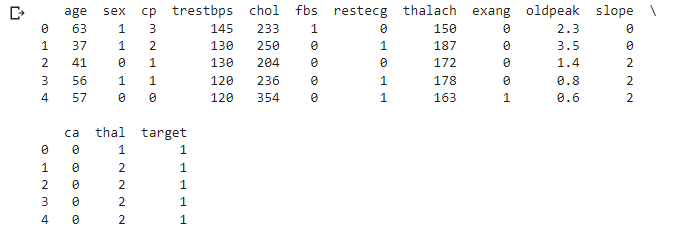
**2. Importer et traiter les données :**

On commence par importer les données à partir du fichier csv et on essaie de visualiser quelques lignes pour voir la structure de celles-ci.

data = pd.read\_csv('dataset.csv')

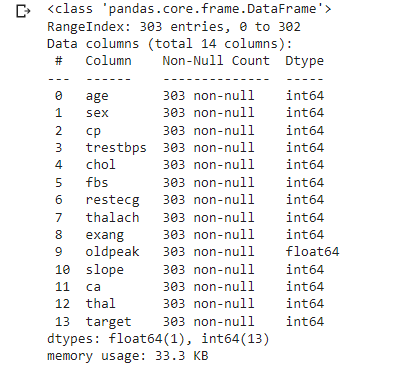
pd.set\_option('display.max\_columns', None)

print(data.head())



On affiche les détails concernant chaque colonne, ainsi on peut vérifier qu'il n'y a pas de valeurs NULL dans quel cas on devrait les remplacer.

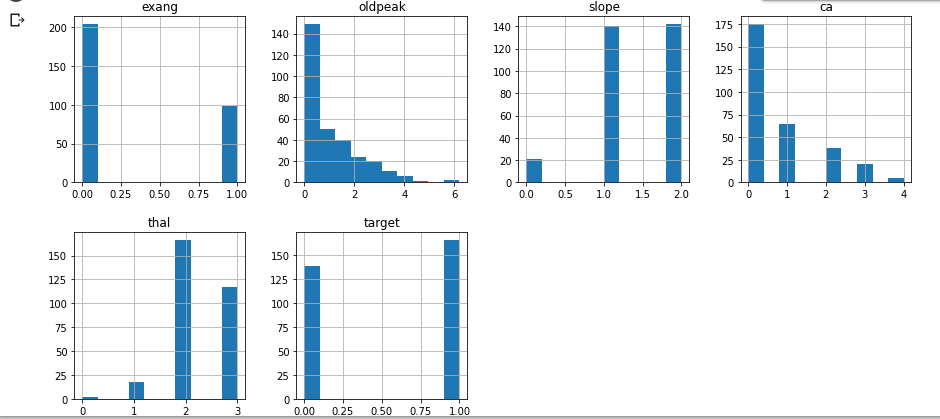
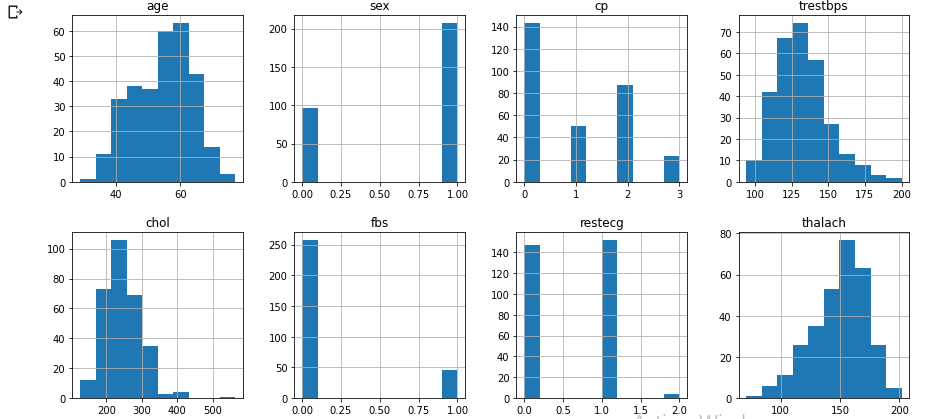
data.info(verbose=True)



On prend également connaissance de la grandeur et de la distribution des données par rapport à chaque attribut :

print(data.describe().T)

data.hist(figsize = (20,20))



On entame la transformation des données récupérées en matrice qu'on réorganise aléatoirement.

data = np.array(data)

m, n = data.shape

print("dimensions : (",m,"x", n,")\n AVANT:", data)

np.random.shuffle(data)

print(" \n APRES :", data)

dimensions : ( 303 x 14 )

AVANT: [[63. 1. 3. ... 0. 1. 1.]

[37. 1. 2. ... 0. 2. 1.]

[41. 0. 1. ... 0. 2. 1.]

...

[68. 1. 0. ... 2. 3. 0.]

[57. 1. 0. ... 1. 3. 0.]

[57. 0. 1. ... 1. 2. 0.]]

APRES : [[58. 0. 1. ... 2. 2. 0.]

[46. 0. 1. ... 0. 2. 1.]

[67. 0. 0. ... 2. 2. 1.]

...

[42. 1. 2. ... 0. 3. 1.]

[64. 1. 0. ... 1. 3. 1.]

[52. 1. 0. ... 0. 0. 0.]]

On divise ensuite la matrice générée en 2 autres matrices :

X\_dev: matrice qui conteindra les données de test , le valeurs de la classe objective sont par contre isolées vers le vecteur Y\_dev

X\_train: matrice qui conteindra les données d'entrainement et pareille, Y\_train contiendra les valeurs de la classe objective

On calcule les transoposées de celles-ci de tel sorte à avoir les différents attributs concernant chaque personne sur un vecteur colonne, ce qui est convenable comme entrée au réseau de neurone.

data\_test = data[0:60].T

Y\_test = data\_test[n-1]

X\_test = data\_test[0:n-1]

data\_train = data[61:m].T

Y\_train = data\_train[n-1]

X\_train = data\_train[0:n-1]

**3. Phase d'entrainement**

On définit les fonctions d'activations: ReLU, softmax ainsi que la dérivée de ReLU qui sera utilisée dans back propagation

def relu(Z):

return np.maximum(Z, 0)

def softmax(Z):

A = np.exp(Z) / sum(np.exp(Z))

return A

def relu\_deriv(Z):

return Z > 0

On définit notre fonction d'entraînement def entrainer(X, Y, alpha, iterations, random) qui va itérer entre plusieurs exécution de forward propagation et backward propagation et qui va modifier les paramètres du modèle à chaque fois pour essayer d'améliorer ses performances. Le paramètre X désigne la matrice d'entraînement, Y le vecteur des targets, alpha le paramètre de modification des erreurs et enfin random est une variable qui prendra la valeur 0 pour effectuer un nouvel entraînement et la valeur 1 pour charger les paramètres d’un modèle pré entraîné.

**Tester sur les données de test :**

def calcul\_resultat(X, W1, b1, W2, b2, W3, b3):

#Forward propagation:

Z1 = W1.dot(X) + b1

A1 = relu(Z1)

Z2 = W2.dot(A1) + b2

A2 = relu(Z2)

Z3= W3.dot(A2) + b3

A3=softmax(Z3)

predictions = np.argmax(A3, 0)

return predictions

test\_res = calcul\_resultat(X\_test, W1, b1, W2, b2, W3, b3)

#Exactitude

print( np.sum(test\_res == Y\_test) / Y\_test.size)

0.8666666666666667

**Exemples de vérification :**

def test\_resultat(index, W1, b1, W2, b2, W3, b3):

cm = X\_train[:, index, None]

prediction = calcul\_resultat(X\_train[:, index, None], W1, b1, W2, b2, W3, b3) c = int(Y\_train[index])

print("Prédiction: ", prediction)

print("Classe objective réelle: ", c)

test\_resultat(56, W1, b1, W2, b2, W3, b3 )

test\_resultat(70, W1, b1, W2, b2, W3, b3 )

test\_resultat(200, W1, b1, W2, b2, W3, b3 )

Prédiction: [1]

Classe objective réelle: 1

Prédiction: [0]

Classe objective réelle: 0

Prédiction: [1]

Classe objective réelle: 1

**F. Analyse et résultats :**

On remarque après de nombreuses exécutions que dû à la nature de la structure des réseaux de neurones, plus particulièrement l’initialisation aléatoire des paramètres, la performance du modèle change d’un entraînement à un autre. Les résultats varient entre 50% et 90% selon les paramètres utilisés.

Les performances les plus faibles sont très tôt observables au début de l'entraînement lorsque les premières itérations génèrent des exactitudes aussi faibles que 40%, cela indique que l'initialisation est très peu convenable. Ces exécutions sont donc directement interrompues et un nouvel entraînement est lancé. En général les meilleures performances sont obtenues pour des premières itérations qui donnent des exactitudes d'environ 60%, celles-ci aboutissent à des exactitudes aussi élevées que 87% à la fin de l'entraînement.

Avec un paramètre alpha=0.001 et un nombre d'itérations de 10.000.000, on peut avoir une exactitude d'environ 89%

Le test sur les données de test permet de savoir à quel point le modèle arrive à généraliser les prédictions pour différentes données d’entrée. On obtient des taux d'exactitude de plus de 85% donc notre modèle arrive à bien généraliser les prédictions à partir des données d'entraînement, c'est à dire que même si on a de nouvelles données, le modèle nous donnera majoritairement des prédictions correctes.

**Influence du paramètre Alpha :**

Le paramètre alpha à initialement été initialisé à 0.5 (valeur inspirée de l’état de l’art ), et ensuite plusieurs valeurs ont été essayées: 0.8, 0.7, 0.6, 0.2, 0.1, 0.01, 0.001.

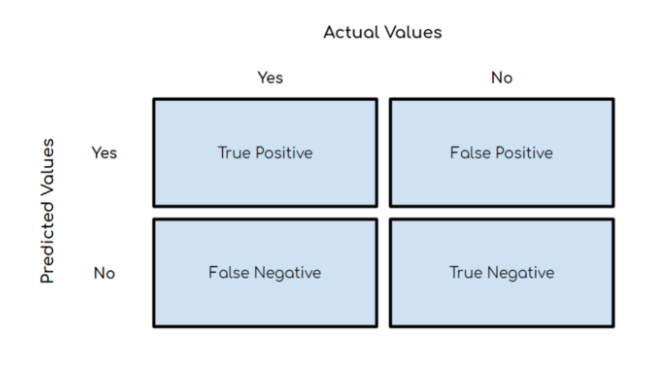
|  |  |
| --- | --- |
| Valeurs | Remarque |
| Valeurs élevées ( 0.8, 0.7, 0.6 ) | Le modèle fini avec des performances généralement faibles. Le modèle bloque des fois sur une exactitude précise et ne modifie plus les paramètres quel que soit le nombre d’itérations restantes. |
| Petites valeurs (0.2, 0.1, 0.01, 0.001) | Plus la valeur est petite, plus la capacité d'appréhension du modèle augmente. De petites rectifications sont effectuées d’une itération à une autre mais le modèle bloque moins dans une exactitude fixe et apprend mieux tant qu’on augmente le nombre d’itérations. |

**Influence du nombre d’itérations :**

|  |  |  |
| --- | --- | --- |
| Nombre d’itérations | Résultat (moyenne) | Temps d'entraînement |
| 20 | Entre 40% et 57% | <1s |
| 100 | Entre 40% et 65% | <1s |
| 500 | Entre 57% et 76% | 1s |
| 1000 | Entre 57% et 84% | 2s |
| 100 000 | Entre 61% et 84% | 4min |
| 2 000 000 | Entre 80% et 86% | 18 min |
| 6 000 000 | Entre 87% et 90% | 45 min |
| 10 000 000 | Entre 88% et 90% | 1h5min |

**La matrice de confusion :**

La matrice de confusion est une manière de juger la performance du modèle par rapport à notre besoin, par exemple si on s’intéresse plus à détecter les personnes malades ou si on priorise la détection des personnes non malades.



True positives: Le nombre de personnes malades prédites correctement

False positive: Le nombre de personnes non malades prédites comme malade

False négative: Le nombre de personnes malades prédites comme non malades

True Negatives: Le nombre de personnes non malades prédites correctement

**Conclusion :**

Chaque problème peut admettre une structure précise de réseau de neurones pour sa résolution et selon chaque problème les paramètres sont réglés à fur et à mesure de la performance du modèle.

On déduit pour notre cas qu’avec un paramètre alpha assez petit, une performance satisfaisante des paramètres aléatoires dès le début de l'entraînement et un nombre d’itérations assez grand, ce réseau de neurone donne de bonnes performances par rapport à la classification de personnes qui ont des problèmes cardiaques ou pas car on aboutit à des modèles très bien entraînés qui peuvent atteindre les 90% en exactitude.