2022-2023
Double licence Mathématiques Économies

Mémoire sur les Intelligences Artificielles

Langer Marin
Sous la direction de Daniel Naie

SOMMAIRE :

Table des matières

Introduction :

Depuis la nuit des temps, l'homme n'a cessé de chercher des moyens pour gagner du temps. La révolution industrielle a mis en lumière une avancée significative dans l'optimisation du temps, mais cela ne suffit pas. On peut alors se demander quand cette quête incessante du temps prendra fin. La réponse réside peut-être dans l'émergence croissante de l'intelligence artificielle dans notre société actuelle. Cependant, comme l'a dit Stephen Hawking, "Le développement de l'intelligence artificielle complète pourrait mettre fin à l'humanité... Les humains, qui sont limités par l'évolution biologique lente, ne pourraient pas rivaliser et seraient dépassés". Cette invention pourrait marquer la fin de notre course contre la montre, ainsi que la fin de l'humanité. Il est donc essentiel de comprendre cette technologie qui occupera une place de plus en plus importante dans les outils technologiques de demain.

Nous commencerons par un bref résumé de l'histoire de l'intelligence artificielle, puis nous introduirons le concept de deep learning à travers la création d'une intelligence artificielle capable de reconnaître les couleurs. Enfin, nous aborderons le cœur du projet : la création d'une intelligence artificielle capable de reconnaitre des chiffres manuscrits.

Les références principales de ce mémoire sont le livre « Python au lycée 2, livre/cours d'Arnaud Baudin » et le livre « Deep Learning and the Game of Go, Max Pumperla ». L'intégralité du code et des ressources utilisés pour ce mémoire sont partagées sur ce GitHub : https://github.com/Niram1/Project-IA-LANGER-Marin.git

Première partie : Introduction à l'intelligence artificielle

L'histoire de l'intelligence artificielle

De nos jours le système de deep learning que l’on précisera dans la prochaine partie est devenu incontournable pour faire apprendre au programme un grand nombre de données sans intervenir spécifiquement. Nous sommes alors à l’aube d(une nouvelle ère de l’intelligence artificielle.
Je vais corriger et clarifier votre texte pour une meilleure compréhension :

Il est courant que le grand public confonde l'algorithme informatique avec l'intelligence artificielle. Pour mieux expliquer ces termes, il est important de les définir. Selon la CNIL un algorithme informatique est la description d'une suite d'étapes permettant d'obtenir un résultat à partir d'éléments fournis en entrée. D'autre part, toujours d’après la CNIL l'intelligence artificielle représente « tout outil utilisé par une machine afin de « reproduire des comportements liés aux humains, tels que le raisonnement, la planification et la créativité ».

Prenons le jeu d'échecs comme exemple. Le jeu consiste en un ensemble de pièces et de pions, chacun ayant une valeur relative en fonction de son importance dans la partie. Pour évaluer une position, on utilise souvent comme référence la valeur d'un pion, qui vaut 1, et celle d'une tour, qui vaut par exemple 5. Bien que le jeu soit bien plus complexe que cela, cela permet aux programmes d'évaluer une position.

En 1949, Shannon a publié un article emblématique intitulé "Programming a Computer For Playing Chess", dans lequel il décrit un algorithme pour une machine à jouer aux échecs. L'objectif était simple : calculer toutes les possibilités pour le prochain coup et choisissait celui qui était le plus avantageux en termes de valeur. Cependant, cette méthode avait des limites évidentes. Plus la puissance de calcul de l'ordinateur était élevée, plus le programme pouvait prédire un certain nombre de coups, mais il ne pouvait évaluer que le score et n'était pas capable de prendre en compte d'autres avantages, tels que l’avantage positionnel ou l’avantage d’espace. Le principe de sacrifice, courant aux échecs, était impossible à intégrer pour ce genre de programme, qui fonctionnait sur le principe d'algorithme plutôt que d'intelligence artificielle.

Bien que les premiers réseaux de neurones aient été découvert dès 1949 et que les premiers vrais programmes d'intelligence artificielle soient apparus dès 1958, il a fallu beaucoup de temps avant qu'une machine puisse battre les meilleurs joueurs d'échecs sans être inquiétée. Même le fameux match de Deep Blue (un système expert d'IBM) contre Garry Kasparov, champion du monde en titre, en mai 1997, était basé sur un algorithme systématique de force brute, dans lequel tous les coups envisageables étaient évalués et pondérés.

Il a fallu attendre l'augmentation de la puissance de calcul et des bases de données massives pour que l'intelligence artificielle puisse être appliquée aux jeux d'échecs. En 2010, une équipe de Google a créé AlphaGo, qui a utilisé un nouveau système innovant pour apprendre à jouer au programme. Cette approche est devenue inductive, ce qui signifie qu'elle ne consiste plus à coder les règles puis de programmer un algorithme qui permet au mieux d’exploiter ces règles, mais à laisser les ordinateurs les découvrir par corrélation et classification, sur la base d'une quantité massive de données.

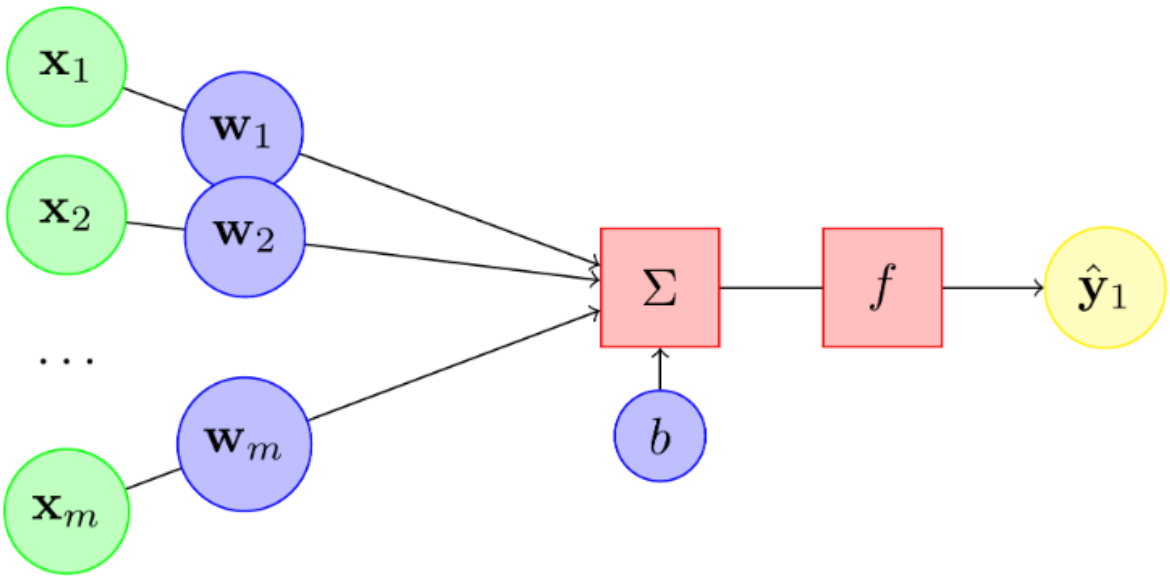
Le deep learning est né de cette évolution, bien que ce ne soit pas précisément pour les échecs. Ce renouveau a permis au programme AlphaGo de surpasser toute concurrence en matière de match entre ordinateurs.

De nos jours, le deep learning est devenu incontournable pour permettre aux programmes d'apprendre à partir d'un grand nombre de données sans intervention humaine. Nous sommes donc à l'aube d'une nouvelle ère de l'intelligence artificielle.

Seconde partie : Introduction au concept de deep learning

Le principe du neurone

Cette seconde partie se concentrera sur la mise en pratique d'un seul neurone. On s'appuiera sur le livre « Python au lycée 2, livre/cours d'Arnaud Baudin ». Tout d'abord quelle est le but et le principe d'un neurone ?
Le perceptron, encore appelé neurone artificiel ou neurone formel, cherche à reproduire le fonctionnement d'un neurone biologique. Il existe différents niveaux d'abstraction, suivant la précision de la modélisation voulue.
Voila comment on représenterai un neurone d'un point de vue mathématique.



Où nous considérons :

- Des entrées notées x sous forme de vecteur qui sont nos variables
- Une sortie nommée y
- Des paramètres w et b influençant le fonctionnement du neurone
-

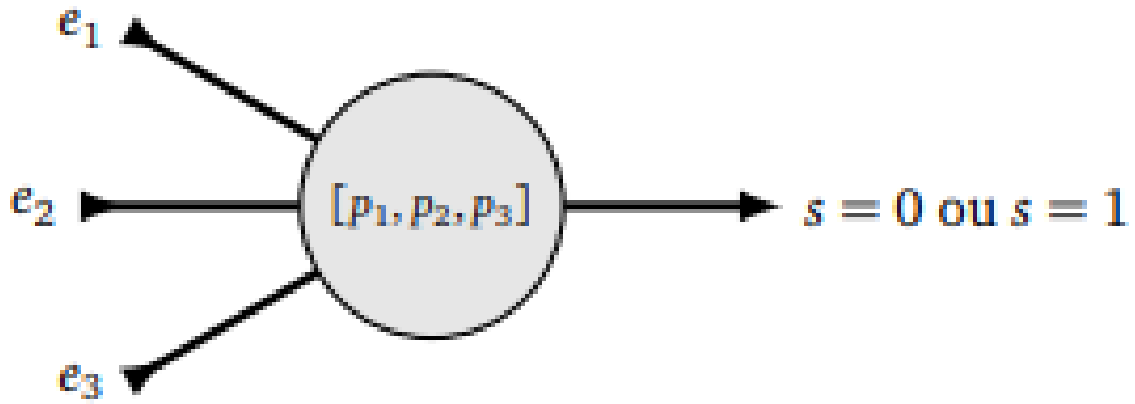
L'équation du neurone devient alors
Y= f(<w,x>+b)

Cette équation dit comment la sortie est calculée. Chaque entrée est multipliée par un poids, un coefficient w Toutes les entrées sont alors sommées et additionnées à un biais b. Le résultat de la somme passe à travers une fonction de transfert f (le plus souvent non linéaire) qui est une fonction d'activation. Cette fonction produit alors la sortie voulue.
Nous expliquerons le principe de poids et de biais à travers l'exemple.

Exemple avec la reconnaissance d'une couleur dominante dans une image

Notre neurone fonctionnera avec les paramètres RGB (red,green ,blue) qui caractérise une image. On aura alors 3 variables X en entrées allant de 0 à 255. Pour plus de lisibilité nous transformerons ces valeurs pour les réduire aux domaines de définitions [0,1].

Ainsi notre neurone ressemblera à ça :



Avec e1,e2,e3 les variables X, [p1,p2,p3] l'état du neurone que l'on utilisera avec une fonction d'activation qui sortira un résultat booléen avec 0 le neurone n'est pas activée et 1 le neurone est activée.
On pourra suivre l'avancée du code à travers le fichier se nommant
« Scirpt reconnaissance d'une couleur dans une image.py »

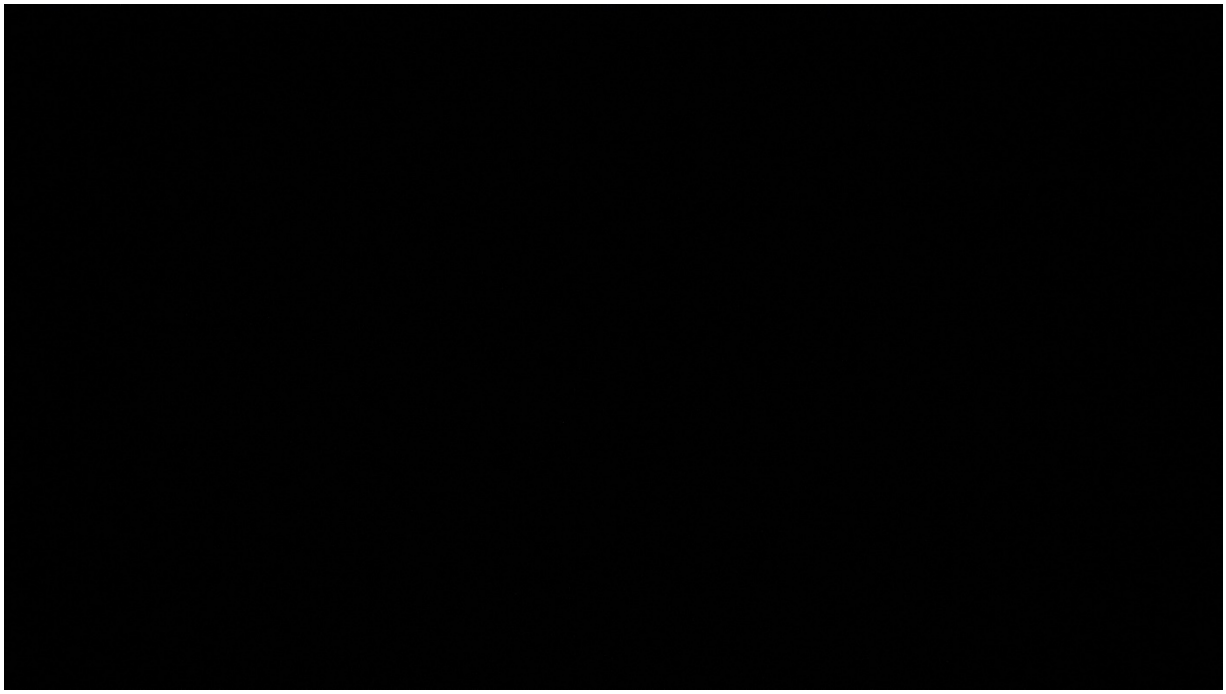
Code et explication

Fonction qui calcule la couleur moyenne d'une image et qui nous en ressort un code rgb

```
# Définition de la fonction pour calculer la couleur moyenne d'une image
def couleur_moyenne(image_path):
    # Ouverture de l'image
    image = Image.open(image_path)
    # Chargement des pixels de l'image
    pixels = image.load()
    # Calcul de la taille de l'image
    largeur, hauteur = image.size
    # Initialisation des variables pour stocker la somme des valeurs de rouge,
vert et bleu de chaque pixel
    rouge_total = vert_total = bleu_total = 0
    # Parcours de tous les pixels de l'image
    for x in range(largeur):
        for y in range(hauteur):
            # Extraction des valeurs de rouge, vert et bleu du pixel courant
            r, v, b = pixels[x, y]
            # Ajout des valeurs de rouge, vert et bleu à la somme totale
            rouge_total += r
```

```
            vert_total += v
            bleu_total += b
    # Calcul de la taille totale de l'image
    taille = largeur * hauteur
    # Calcul de la couleur moyenne de l'image
    rouge_moyen = rouge_total // taille
    vert_moyen = vert_total // taille
    bleu_moyen = bleu_total // taille
    # Retour de la couleur moyenne sous la forme d'un tuple de 3 entiers
représentant les valeurs de rouge, vert et bleu
    return (rouge_moyen, vert_moyen, bleu_moyen)
```

Par exemple si on prend l'image :



Qui est une image de couleur noir donc de couleur RGB (255,255,255) donc de paramètres (1,1,1)
Et que l'on test avec

```
print(couleur_moyenne("C:/Users/Marin/Documents/Université/Memoire/Couleur/Banque_images_couleurs/Couleur_noir.jpg") )
```

On obtient bien (1, 1, 1)
Cette fonction nous aidera par la suite pour choisir quelles images sont de la couleur que l'on veut que l'IA reconnaisse ou inversement.

Fonction permettant de choisir les images sur lesquelles on veut entrainer l'IA

Dans le dossier banque image on peut voir plusieurs images avec des couleurs monochrome pour simplifier la compréhension. Cette banque d'image peut évidemment être plus développé pour affiner la qualité du neurone

```
# Définir le chemin d'accès au dossier contenant les images
dossier_images =
"C:/Users/Marin/Documents/Université/Memoire/Couleur/Banque_images_couleurs"

# Créer une liste de toutes les images dans le dossier
liste_images = os.listdir(dossier_images)

# Créer une liste de tuples contenant les informations sur chaque image
liste_entrees_objectifs = []
for image in liste_images:
    # Afficher l'image
    img = Image.open(os.path.join(dossier_images, image))
    img.show()

    # Demander à l'utilisateur si l'image doit être utilisée pour entraîner le
neurone
    reponse = input("Cette image doit-elle être utilisée pour entraîner le
neurone ? (o/n) ")

    # Ajouter un tuple à la liste des entrées et objectifs en fonction de la
réponse de l'utilisateur
    if reponse.lower() == "o":
        objectif = 1
    else:
        objectif = 0
    codeRGB = couleur_moyenne(os.path.join(dossier_images, image))
    liste_entrees_objectifs.append((codeRGB, objectif))
```

Cette fonction permet d'aller chercher chaque image et que l'utilisateur réponde par oui ou par non pour savoir si l'image doit être utilisé pour le neurone ou non.
Lorsque que l'on lance le programme une fenêtre apparait pour afficher l'image et le programme nous demande alors « Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) » où il faut répondre par o ou n. Le résultat sera alors donné sous la forme d'une liste avec un tupples et un objectif de cette forme

```
liste_entrees_objectifs = [
([1,0,0],1), ([0,1,1],0), ([1,1,0],0),
([1,0,0.2],1), ([0,1,0],0), ([0,0,0],0),
([1,0,1],0), ([0.7,0,0],1), ([0.5,0.5,0.5],0),
([0.9,0.2,0],1), ([0.9,0,0],1), ([1,1,1],0),
([0.2,1,0],0), ([0.8,0.2,0],1), ([0.7,0.1,0.1],1) ]
```

Avec par exemple pour la première valeur de la liste ([1,0,0],1) qui correspond à une liste du code RGB et un chiffre qui est soit 1 soit 0 qui sera l'objectif qu'on utilisera pour la fonction d'activation pour la suite du code.

Fonction d'activation

Cette fonction est le cœur du neurone. Pour savoir quand le neurone va s'activer En fonction de l'entrée (e1 , e2 , e3) qui serait ([1,0,0]) dans l'exemple précédent et de l'état du neurone (p1 , p2 , p3) on commence par calculer une valeur q selon la formule :

$$q = p1\ e1 + p2\ e2 + p3\ e3$$

Ensuite on regarde si q est plus grand ou plus petit que 1 pour déterminer si le neurone est activé (c'est-à-dire la valeur s renvoyée par le neurone) :
s = 0 si q < 1,
s = 1 si q > 1.

Le code est alors :

```
# Fonction d'activation du neurone
def activation (neurone, entree):
    # Initialisation de la somme pondérée à 0 et de la sortie à 0
    somme = 0
    s = 0
    # Pour chaque élément de l'entrée, on ajoute le produit du poids
correspondant et de l'entrée à la somme pondérée
    for i in range(3):
        somme += neurone[i] * entree[i]
    # Si la somme pondérée est supérieure ou égale à 1, la sortie est 1, sinon
elle est 0
    if somme >= 1:
        s = 1
    else:
        s = 0
    # On retourne la sortie calculée
    return s

"""
La fonction activation calcule la sortie du neurone en effectuant une somme
pondérée des entrées multipliées par les poids correspondants.
Si la somme pondérée est supérieure ou égale à 1, la sortie est 1, sinon elle
est 0.
Cette fonction est utilisée pour déterminer la réponse du neurone à une entrée
donnée, en comparant sa sortie à l'objectif attendu.
"""
```

D'un point de vue mathématiques la question à laquelle répond le neurone est « ce point de coordonnées (e1 , e2 , e3) est-il au-dessus ou en dessous de ce plan P ? » Le plan P dont il est question est le plan d'équation p1x + p2y + p3z = 1 déterminé par l'état (p1 , p2 , p3) du neurone.
Le problème principal est, qu'au départ, on ne connaît pas le plan P qui répond au problème que l'on se pose, autrement dit on ne sait pas quel état (p1 , p2 , p3) convient.
Nous allons donc voir comment calculer les poids à travers la fonction d'apprentissage.

Fonction d'apprentissage

On part d'un état initial quelconque par exemple (p1 , p2 , p3) = (1,1,1). Pour faire évoluer l'état du neurone jusqu'à la bonne valeur de (p1 , p2 , p3) on va l'entraîner comme un enfant : on lui montre une couleur et on lui dit « c'est du bleue», puis on lui montre une autre couleur et on lui dit « ce n'est pas du bleue ». Cela peut être le cas pour chaque couleur du spectre chromatique. À chaque étape l'état du neurone (p1 , p2 , p3) est modifié.

Voici une étape d'entraînement : on donne une entrée (e1 , e2 , e3) et l'objectif b = 0 ou b = 1 qui est la sortie attendue si le neurone était parfaitement paramétré (correspondant au 1 de l'exemple ([1,0,0],1). On calcule la sortie s = 0 ou s = 1 que renvoie le neurone selon l'entrée (e1 , e2 , e3) dans son état actuel, ensuite plusieurs cas sont possibles :

• Si l'objectif b est égal à la sortie s alors le neurone fonctionne bien pour cette entrée, on ne change pas l'état du neurone.

• Si la sortie calculée est s = 0 alors que l'objectif est b = 1, alors on change l'état du neurone (p1 , p2 , p3) en un nouvel état (p 0 1 , p 0 2 , p 0 3) selon la formule :

P1' = p1 + εe1
P2' = p2 + εe2
P3' = p3 + εe3

• Si la sortie calculée est s = 1 alors que l'objectif est b = 0, alors on change l'état du neurone selon la formule :

P1' = p1 - εe1
P2' = p2 - εe2
P3' = p3 - εe3

Le paramètre ε est un petit réel. On prendra ε = 0.2 par exemple. Ce paramètre est défini par des méthodes d'itération en calculant la qualité du modèle nous verrons par la suite comment calculer cet epsilon pour le programme de reconnaissance de d'écriture manuscrit. On répète ces étapes avec plusieurs entrées et objectifs. Géométriquement chaque apprentissage déplace un petit peu le plan P pour mieux répondre au problème. L'état du neurone va converger vers un état (p1 , p2 , p3). Une fois la phase d'apprentissage terminée on conserve cet état final (p1 , p2 , p3). Maintenant on laisse répondre le neurone en regardant s'il s'active ou pas selon des entrées quelconques (e1 , e2 , e3) (même si ces entrées ne font pas partie de la liste d'apprentissage).

```
# Fonction d'apprentissage du neurone
def apprentissage(neurone, entree, objectif, epsilon):
    neuronebis = []
    # Calcul de la sortie actuelle du neurone à l'aide de la fonction
d'activation
    s = activation(neurone, entree)
    # Si la sortie actuelle correspond à l'objectif attendu, on ne modifie pas
les poids du neurone
    if s == objectif:
        return neurone
    else:
        # Si la sortie actuelle est fausse, on ajuste les poids du neurone en
fonction de l'erreur commise
        if s == 0 and objectif == 1:
            # Si la sortie actuelle est 0 mais l'objectif attendu est 1, on
ajoute un petit poids positif à chaque élément du neurone
            for i in range(3):
                neuronebis.append(neurone[i] + epsilon * entree[i])
        else:
            # Si la sortie actuelle est 1 mais l'objectif attendu est 0, on
soustrait un petit poids positif à chaque élément du neurone
            for i in range(3):
```

```
                neuronebis.append(neurone[i] - epsilon * entree[i])
        # On retourne les nouveaux poids du neurone après l'ajustement
        return neuronebis


"""
La fonction apprentissage ajuste les poids du neurone en fonction de l'erreur
commise par le neurone lors de sa réponse à une entrée donnée.
Si la sortie du neurone correspond à l'objectif attendu, les poids ne sont pas
modifiés. Sinon, les poids sont ajustés en fonction de l'erreur commise.
 Si la sortie actuelle est 0 mais l'objectif attendu est 1, on ajoute un petit
poids positif à chaque élément du neurone.
 Si la sortie actuelle est 1 mais l'objectif attendu est 0, on soustrait un
petit poids positif à chaque élément du neurone.
 Le taux d'apprentissage epsilon est utilisé pour déterminer l'amplitude de
l'ajustement des poids. La fonction retourne les nouveaux poids du neurone
après l'ajustement.
"""
```

Fonction époque apprentissage

Pour que le neurone s'entraîne il faut lui procurer plusieurs données. C'est une fonction qui calcule le nouvel état du neurone après entraînement sur chaque élément de la liste que l'on avait crée juste avant avec les images de la banque d'images. Il s'agit juste d'itérer la fonction apprentissage() sur chaque élément de la liste. Un entraînement sur la totalité du jeu de tests s'appelle une époque.

```
# Fonction pour effectuer une époque d'apprentissage sur le neurone
def epoque_apprentissage(neurone_init, liste_entrees_objectifs):
    for i in range (len(liste_entrees_objectifs)):
        # On applique la fonction d'apprentissage pour chaque entrée/objectif
de la liste
        neurone_init = apprentissage(neurone_init,
liste_entrees_objectifs[i][0], liste_entrees_objectifs[i][1], epsilon)

    return neurone_init
```

Fonction plusieurs époques apprentissage

La fonction `plusieur_epoque` applique 50 époques d'apprentissage sur le neurone en appelant la fonction `epoque_apprentissage` à chaque itération.
L'objectif est d'améliorer les performances de classification du neurone en affinant les poids à chaque époque d'apprentissage.

```
# Fonction pour effectuer plusieurs époques d'apprentissage sur le neurone
def plusieur_epoque(neurone_init, liste_entrees_objectifs):
```

```
    for i in range(50):
        # On applique plusieurs époques d'apprentissage sur le neurone
        neurone_init = epoque_apprentissage(neurone_init,
liste_entrees_objectifs)
    return neurone_init
```

Test

SI on choisit que les images rouges sans la couleur orange on aura alors ce code :
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) o
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) o
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) o
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n
Cette image doit-elle être utilisée pour entraîner le neurone ? (o/n) n

Et le résultat est un neurone égal à [41.60000000000001, -67.8, -59.80000000000001] ainsi lorsqu'on utilisera la fonction activation

```
print(activation([41.60000000000001, -67.8, -59.80000000000001]
),couleur_moyenne("C:/Users/Marin/Documents/Université/Memoire/Couleur/Banque_
images_couleurs/Couleur_orange.jpg"))
```

Le résultat sera 0. En effet le code RGB de l'image orange est (255, 128, 65) donc on a
q=41.6*255+(-67.8)*128+(-59.8)*65=-1957.4<1
alors que pour :

```
print(activation([41.60000000000001, -67.8, -59.80000000000001]
),couleur_moyenne("C:/Users/Marin/Documents/Université/Memoire/Couleur/Banque_
images_couleurs/Couleur_rouge.jpg"))
```

Le résultat sera 1. En effet le code RGB de l'image orange est (255, 0, 0) donc on a
q=41.6*255+0+0=10608 > 1 donc s=1

Conclusion

Nous avons donc vue en quoi consistait le principe de neurone d'un point de vue mathématique et comment créer une IA avec un seul neurone pour reconnaitre une couleur monochrome d'une image avec une liste que l'on aura prédéfinis. Par la suite nous verrons comment se concrétise une accumulation de couche de neurones et les principes qu'il y a derrière des algorithmes de deep learning.

2022-2023
Double licence Mathématiques Économies

Troisième partie : Création d'une Intelligence artificielle de type OCR (reconnaissance optique de caractère) sur des écritures de chiffres manuscrites

Introduction

Nous allons par la suite principalement utiliser le livre « Deep Learning and the Game of Go, Max Pumperla » et plus précisément le chapitre 5 intitulé « Getting started with neural networks ».
Le but de cette partie est de comprendre le principe de couches de neurones. Certaines parties du code sont relativement assez complexe et difficile à assimiler. En comprendre tous les détails rédactionnels relèvent d'années d'expériences en IA. Le principe du code quant à lui reste accessible.
Le code se trouve dans le Githbub avec le nom « IA fait main »

Au cœur des réseaux de neurones artificiels (ANN) se trouve l'idée de s'inspirer de la neuroscience et de modéliser une classe d'algorithmes qui fonctionnent de manière similaire à la façon dont nous supposons que certaines parties du cerveau fonctionnent. En particulier, nous utilisons la notion de neurones comme blocs atomiques pour nos réseaux artificiels. Les neurones forment des groupes appelés couches, et ces couches sont connectées les unes aux autres de manière spécifique pour former un réseau. En utilisant des données d'entrée, les neurones peuvent transférer des informations couche par couche via des connexions, et nous disons qu'ils s'activent si le signal est suffisamment fort. De cette manière, les données se propagent à travers le réseau jusqu'à ce que nous arrivions à la dernière étape, la couche de sortie, à partir de laquelle nous obtenons nos prédictions. Ces prédictions peuvent ensuite être comparées à la sortie attendue pour calculer l'erreur de prédiction, que le réseau utilise pour apprendre et améliorer les prédictions futures.

Chaque image de notre ensemble de données OCR est composée de pixels disposés sur une grille, et on va devoir analyser les relations spatiales entre les pixels pour déterminer quel chiffre elle représente.

Cas d'utilisation concret

Avant d'approfondir les réseaux de neurones, commençons par un cas d'utilisation concret. Avec un chiffre précis Tout au long de ce chapitre, nous allons développer une application capable de prédire assez précisément les chiffres à partir de données d'images de chiffres manuscrits, avec une précision d'environ 95%. Pour ce faire, nous utiliserons l'ensemble de données de chiffres manuscrits du Modified National Institute of Standards and Technology (MNIST), une base de données conséquente d'image manuscrite labelisé.

Base de données MNIST

L'ensemble de données MNIST se compose de 60 000 images, chacune de dimensions 28 × 28

pixels. Quelques exemples de ces données sont présentés dans l'image à droite. Pour les humains, la reconnaissance de la plupart de ces exemples est une tâche triviale, et vous pouvez facilement lire les exemples de la première ligne comme 7, 5, 3, 9, 3, 0, et ainsi de suite. Cependant, dans certains cas, il est difficile même pour les humains de comprendre ce que représente l'image. Par exemple, la quatrième image de la cinquième ligne pourrait facilement être un 4 ou un 9.
Chaque image dans MNIST est annotée avec une étiquette, un chiffre de 0 à 9 représentant la valeur réelle représentée sur l'image.

Reformuler les valeurs de MNIST pour mieux les exploiter

Étant donné que les étiquettes dans cet ensemble de données sont des entiers de 0 à 9, nous allons utiliser une technique appelée encodage one-hot (one-hot encoding) pour transformer le chiffre 1 en un vecteur de longueur 10 avec des 0 partout, sauf à la position 1 où nous plaçons un 1. Cette représentation est utile et largement utilisée dans le contexte de l'apprentissage automatique. En réservant la première position dans un vecteur pour l'étiquette 1, nous permettons aux algorithmes tels que les réseaux de neurones de distinguer plus facilement entre les étiquettes. En utilisant l'encodage one-hot, par exemple, le chiffre 2 a la représentation suivante : [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]. Cela nous permettra de faire des neurones pour chaque chiffre en déplaçant le vecteur de chaque chiffre.

```
def encode_label(j):  # Définit une fonction appelée 'encode_label' qui prend
un argument 'j'
    e = np.zeros((10, 1))  # Crée un tableau numpy de dimensions (10, 1)
rempli de zéros
    e[j] = 1.0  # Attribue la valeur 1.0 à l'indice 'j' du tableau 'e'
    return e  # Renvoie le tableau 'e' qui contient l'encodage one-hot du
label

# Nous utilisons l'encodage one-hot pour les indices en vecteurs de longueur
10.
```

```
def shape_data(data):
    features = [np.reshape(x, (784, 1)) / 255.0 for x in data[0]]  # <1>

    labels = [encode_label(y) for y in data[1]]  # <2>

    return list(zip(features, labels))
  # <3>

# <1> Nous aplatissons les images d'entrée en vecteurs de caractéristiques de
longueur 784.
# <2> Tous les labels sont encodés en one-hot.
# <3> Ensuite, nous créons des paires de caractéristiques et de labels.

def load_data_impl():
    # Définit le chemin du fichier contenant les données MNIST
    path = 'C:\\Users\\Marin\\Documents\\Université\\Memoire\\IA go\\Fichier
mnist\\mnist.npz'

    # Charge le fichier en mémoire en utilisant numpy
```

```
    f = np.load(path)

    # Extrait les ensembles de données d'entraînement (x_train et y_train)
    x_train, y_train = f['x_train'], f['y_train']

    # Extrait les ensembles de données de test (x_test et y_test)
    x_test, y_test = f['x_test'], f['y_test']

    # Ferme le fichier
    f.close()

    # Retourne les données d'entraînement et de test sous forme de tuples
    return (x_train, y_train), (x_test, y_test)

def load_data():
    train_data, test_data = load_data_impl()
    return shape_data(train_data), shape_data(test_data)

# <4> Décompresser et charger les données MNIST génère trois ensembles de
données.
# <5> Nous ne conservons pas les données de validation ici et redimensionnons
les deux autres ensembles de données.
```

N'utilisant pas la même version de python que l'auteur du livre il était alors compliqué d'avoir exactement le même code. Les mises à jour ont rendu obsolètes sont codes, ce code trouvé sur un forum permet de palier ce problème. Le cœur du mémoire ne se joue pas sur l'implantation de la base de données dans notre algorithme mais le principal problème auquel j'ai pu faire face concernait ce manque de compatibilité entre le nouveau python et l'ancien ce qui rendait le code moins optimisé. Cela se remarquera sur certaines valeurs de précision et de test pour notre algorithme.

Exemple concret

Nous allons voir un exemple concret en faisant la moyenne de chaque pixel d'un chiffre précis. La fonction nous montrera alors les points les plus distinctifs d'un chiffre. On a alors ce code la :

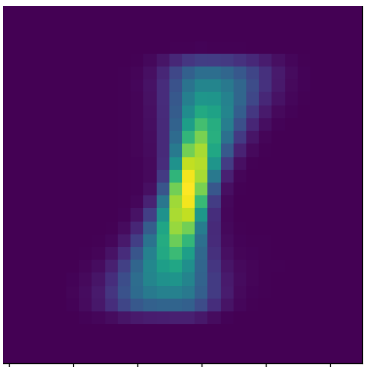
```
def average_digit(data, digit):
    # Filtre les données en ne gardant que celles correspondant au chiffre
spécifié (digit)
    filtered_data = [x[0] for x in data if np.argmax(x[1]) == digit]

    # Convertit la liste de données filtrées en un tableau numpy
    filtered_array = np.asarray(filtered_data)

    # Calcule la moyenne des données filtrées en fonction de l'axe 0 (chaque
pixel)
    return np.average(filtered_array, axis=0)
```

```
"""
Cette fonction prend en entrée un ensemble de données et un chiffre spécifique
(0-9).
 Elle filtre les données pour ne conserver que celles correspondant au chiffre
spécifié, puis calcule et retourne la valeur moyenne pour chaque pixel des
images filtrées.

"""
```

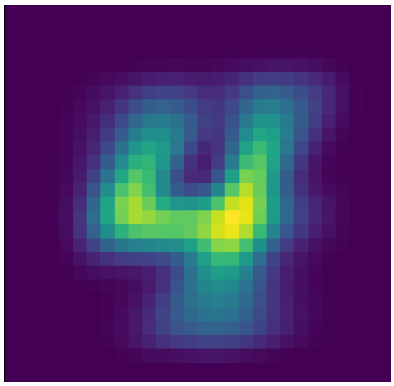
On va alors le code pour plusieurs chiffres avec ce code en remplaçant x par chaque chiffre :

```
train, test = load_data()
avg_eight = average_digit(train, x)  # <2>

# <1> Nous calculons la moyenne de tous les échantillons de nos données
représentant un chiffre donné.
# <2> Nous utilisons x comme paramètres pour un modèle simple de détection des
huit.


from matplotlib import pyplot as plt



img = (np.reshape(avg_eight, (28, 28)))
plt.imshow(img)
plt.show()
```

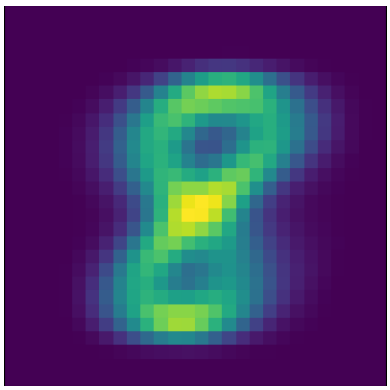
Chiffre 1: On voit alors que le 1 à une caractéristique principale ressemblant à un bâton et que le demi chapeau à sa tête n'est pas là.



Chiffre 4 : Les principales points chauf sont à distance du centre ce qui caractérise un chiffre asymétrique à la différence du 1 et du 8.



Chiffre 8 : Le chiffre 8 lui a plusieurs caractéristiques principales notamment des zones de vide entre chaque boucle de part et d'autre de l'axe des abscisses.



On voit que chacun de ses chiffres possèdent des caractéristiques propres à lui. Notre but sera alors de produire une IA qui sort différentes probabilités sur l'identification d'un chiffre lorsqu'on lui mettra des images en dehors de la base de données MNIST. On créera alors une fonction qui donne un résultat en fonction d'un seuil d'acceptabilité d'une probabilité.

Activation

Cette représentation moyenne d'un 8, avg_eight, dans notre ensemble d'entraînement MNIST, devrait contenir beaucoup d'informations sur ce que cela signifie d'avoir un 8 sur une image. Nous utiliserons avg_eight comme paramètres d'un modèle simple pour décider si un vecteur d'entrée donné x, représentant un chiffre, est un 8. Dans le contexte des réseaux neuronaux, nous parlons souvent de poids lorsqu'il s'agit de paramètres, et avg_eight servira de poids. Pour plus de commodité, nous utiliserons la transposition et définirons W = np.transpose(avg_eight). Nous pouvons ensuite calculer le produit scalaire de W et x, qui effectue une multiplication terme à terme des valeurs de W et x et additionne toutes les 784 valeurs résultantes. Si notre heuristique est correcte, si x est un 8, les pixels individuels devraient avoir une tonalité plus sombre à peu près aux mêmes endroits que W, et vice versa. Inversement, si x n'est pas un 8, il devrait y avoir moins de chevauchement. Testons cette hypothèse sur quelques exemples.

```
x_3 = train[2][0] #qui corréspond à un 4 dans la base de donnée
x_18 = train[17][0] #qui corréspond à un 8 dans la base de donnée
W = np.transpose(avg_eight)
print(np.dot(W, x_3)) # = 20.1
```

```
print(np.dot(W, x_18)) # = 54.2
```

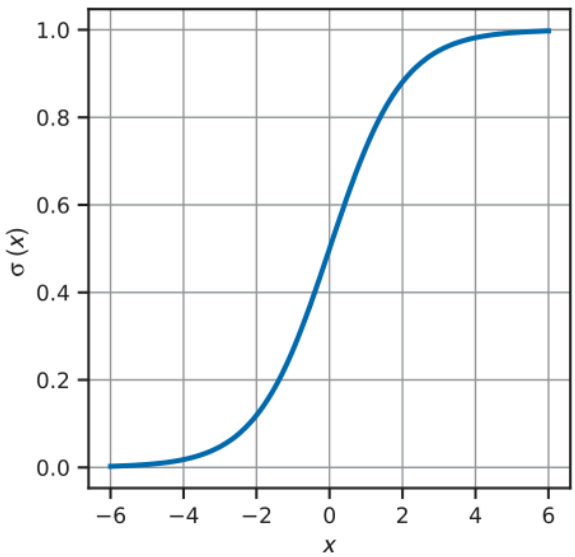
Nous calculons le produit scalaire de nos poids W avec deux échantillons MNIST, l'un représentant un 4 et l'autre représentant un 8. On peut constater que le résultat final de 54,2 pour le 8 est beaucoup plus élevé que le résultat de 20,1 pour le 4. Il faut maintenant revenir au à la deuxième partie de ce mémoire avec l'apparition d'une fonction d'activation. En principe, le produit scalaire de deux vecteurs peut donner n'importe quel nombre réel. Pour remédier à cela, on transforme la sortie du produit scalaire dans la plage [0, 1]. De cette manière, on peut, par exemple, essayer de définir une valeur seuil à 0,5 et déclarer tout ce qui est au-dessus de cette valeur comme étant un 8 même si ce n'est pas souvent vraie.

Sigmoid ou fonction d'activation

Pour ce faire on peut utiliser la fonction sigmoid que l'on reverra tout au long du mémoire.
La fonction sigmoid est définie par l'équation suivante :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

où σ(x) représente la valeur de sortie de la fonction sigmoid pour une valeur donnée x et e est l'exponentielle



Son graph est :

Les principes de la fonction sigmoid sont les suivants :

1. Plage de sortie limitée : La fonction sigmoid comprime les valeurs d'entrée dans une plage limitée entre 0 et 1. Cela signifie que quelles que soient les valeurs réelles de x, la sortie de la fonction sigmoid sera toujours comprise entre 0 et 1.

2. Propriété de monotonie : La fonction sigmoid est une fonction monotone croissante, ce qui signifie que si x1 < x2, alors σ(x1) < σ(x2). Cela garantit que l'ordre des valeurs est conservé après transformation par la fonction sigmoid.

3. Valeur médiane : La valeur médiane de la fonction sigmoid est de 0,5, ce qui signifie que σ(0) = 0,5. Cela signifie que lorsque la valeur d'entrée est proche de zéro, la sortie de la fonction sigmoid est proche de 0,5.

4. Asymptotes : La fonction sigmoid a deux asymptotes horizontales, c'est-à-dire que lorsque la valeur d'entrée x tend vers l'infini positif ou négatif, la sortie de la fonction sigmoid tend respectivement vers 1 et 0.

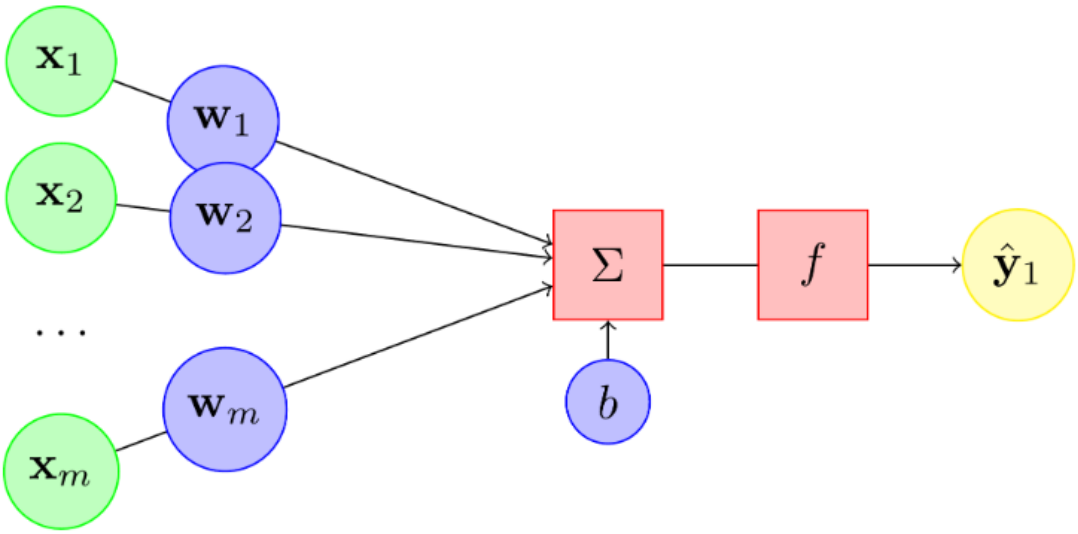
```
def sigmoid_double(x):
    return 1.0 / (1.0 + np.exp(-x))

# La fonction sigmoid_double prend en argument un nombre x et retourne sa
valeur transformée à l'aide de la fonction sigmoïde.

def sigmoid(z):
    return np.vectorize(sigmoid_double)(z)

# La fonction sigmoid prend en argument un tableau NumPy z et applique la
fonction sigmoïde à chaque élément de ce tableau à l'aide de la fonction
np.vectorize. Elle retourne le tableau transformé.
```

Notons que nous fournissons à la fois sigmoid_double, qui opère sur des valeurs de type double, ainsi qu'une version qui calcule la sigmoïde pour des vecteurs que nous utiliserons largement dans ce chapitre. Avant d'appliquer la sigmoïde à notre calcul précédent, notons que la sigmoïde de 2 est déjà proche de 1, donc pour nos deux échantillons précédemment calculés, sigmoid(54.2) et sigmoid(20.1) seront pratiquement indiscernables. Nous pouvons résoudre ce problème en décalant la sortie du produit scalaire vers 0. On appelle cela l'application d'un terme de biais, que nous appelons souvent b. Exactement le même principe que pour l'application de la deuxième partie. À partir des échantillons, nous pouvons estimer que la valeur de b peut être b = -45. En utilisant les poids et le terme de biais, nous pouvons maintenant calculer les prédictions de notre modèle de la manière suivante.

```
ef predict(x, W, b):  # <1>
    return sigmoid_double(np.dot(W, x) + b)


# <1> Une prédiction simple est définie en appliquant la fonction sigmoïde à
la sortie de np.dot(W, x) + b.

# La fonction predict prend en argument une entrée x (un tableau NumPy), une
matrice de poids W et un biais b.

b = -45  # <2>
```

```
# <2> Sur la base des exemples calculés jusqu'à présent, nous fixons le terme
de biais à -45. La moyenne entre 54.2 et 20.1 est d'environ 45 donc on réduit
de 45

# print(predict(x_3, W, b))   # <3>
# print(predict(x_18, W, b))  # <4>
# <3> La prédiction pour l'exemple avec un "4" est proche de zéro (1.52 10
puissance -11).
# <4> La prédiction pour un "8" est de 0,999 ici. Notre heuristique semble
donner des résultats intéressants.
```

On retrouve alors le même schéma que dans la deuxième partie :



Avec b=-45, W= np.transpose(avg_eight) et x= différences de pixels pour chaque image et f=sigmoid_double
On a alors une seule couche de neurone qui nous permet peut-être de déjà avoir un résultat satisfaisant.

Test

On doit maintenant voir si ce simple modèle fonctionne pour reconnaître tous les chiffres donc d'avoir un seul neurone pour reconnaitre les chiffres, juste avec la moyenne additionnée au biais transformé avec la fonction sigmoid_double.

```
Treshold signifie le seuil auquel on accepte la prédiction
def evaluate(data, digit, threshold, W, b):  # fonction qui évalue les
performances du modèle
    total_samples = 1.0 * len(data)  # calcule le nombre total d'échantillons
    correct_predictions = 0  # initialise le nombre de prédictions correctes à
0
    for x in data:  # boucle sur chaque échantillon dans les données d'entrée
        if predict(x[0], W, b) > threshold and np.argmax(x[1]) == digit:  # si
la prédiction est correcte
            correct_predictions += 1  # incrémente le nombre de prédictions
correctes
```

```
        if predict(x[0], W, b) <= threshold and np.argmax(x[1]) != digit:  #
si la prédiction est incorrecte
            correct_predictions += 1  # incrémente le nombre de prédictions
correctes (cela compte comme une erreur car le modèle a prédit que c'était la
bonne classe)
    return correct_predictions / total_samples  # retourne le taux de
prédictions correctes sur l'ensemble des échantillons

# <1> Comme métrique d'évaluation, nous choisissons la précision, le rapport
des prédictions correctes parmi toutes.
# <2> Prédire une instance de huit comme "8" est une prédiction correcte.
# <3> Si la prédiction est inférieure à notre seuil et que l'échantillon n'est
pas un "8", nous avons également prédit correctement.


evaluate(data=train, digit=8, threshold=0.5, W=W, b=b)  # <1>

# Cette fonction évalue les performances du modèle entraîné sur l'ensemble de
données d'entraînement (train) pour reconnaître le chiffre 8.
# digit=8 spécifie que la fonction recherche des images contenant le chiffre
8.
# threshold=0.5 spécifie la valeur de seuil à utiliser pour la classification
binaire.
# W et b sont les poids et les biais du modèle qui ont été entraînés
précédemment.

evaluate(data=test, digit=8, threshold=0.5, W=W, b=b)   # <2>

# Cette fonction évalue les performances du modèle entraîné sur l'ensemble de
données de test (test) pour reconnaître le chiffre 8.
# digit=8 spécifie que la fonction recherche des images contenant le chiffre
8.
# threshold=0.5 spécifie la valeur de seuil à utiliser pour la classification
binaire.
# W et b sont les poids et les biais du modèle qui ont été entraînés
précédemment.

eight_test = [x for x in test if np.argmax(x[1]) == 8]
evaluate(data=eight_test, digit=8, threshold=0.5, W=W, b=b)  # <3>

# Cette fonction évalue les performances du modèle entraîné sur un
sous-ensemble des données de test (eight_test) pour reconnaître le chiffre 8.
# digit=8 spécifie que la fonction recherche des images contenant le chiffre
8.
# threshold=0.5 spécifie la valeur de seuil à utiliser pour la classification
binaire.
# W et b sont les poids et les biais du modèle qui ont été entraînés
précédemment.
# eight_test est une liste de tous les exemples de test contenant le chiffre
8.
```

On a alors :

`evaluate(data=train, digit=8, threshold=0.5, W=W, b=b) = 0.682` la précision sur l'ensemble d'entraînement est d'environ 68%. Ce qui est peut pour un modèle que l'on a entraîné spécifiquement avec les données. Cependant, il n'est pas judicieux d'évaluer sur l'ensemble d'entraînement, car cela ne vous indique pas à quel point votre algorithme généralise (à quel point il se comporte bien sur des données jamais vues auparavant).

`evaluate(data=test, digit=8, threshold=0.5, W=W, b=b) = 0.6694` Les performances sur les données de test sont proches de celles de l'entraînement, d'environ 67% ce qui reste faible

```
eight_test = [x for x in test if np.argmax(x[1]) == 8]
evaluate(data=eight_test, digit=8, threshold=0.5, W=W, b=b)  = 0.8193018480492813
```

Ce qui est relativement correcte pour un premier modèle avec une seule couche ce qui signifie qu'on a raison 8 fois sur 10 sur les cas de 8 non vue auparavant ce qui ne reste pas assez fiable pour être exploité.

On va alors devoir changer les paramètres de biais et du treshold et rajouter un certain nombre de couche pour augmenter significativement la précision.

Changement de dimension

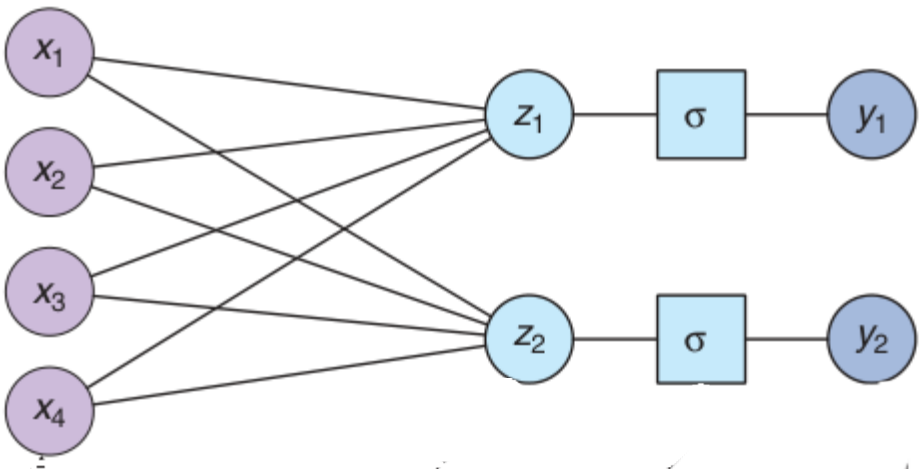
Introduction

Dans le premier cas, nous avions simplifié le problème de reconnaissance des chiffres manuscrits en un problème de classification binaire ; c'est-à-dire distinguer un 8 de tous les autres chiffres. Mais nous souhaitons prédire 10 classes, une pour chaque chiffre. Formellement, on peut y parvenir assez facilement en modifiant y, W et b ; en modifiant la sortie, le poids et le biais de notre modèle.

Tout d'abord, on prend y un vecteur de longueur 10 ; y aura une valeur représentant la probabilité de chaque chiffre parmi les 10 :

$$y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_8 \\ y_9 \end{bmatrix} \qquad \sigma(z) = \begin{bmatrix} \sigma(z_0) \\ \sigma(z_1) \\ \vdots \\ \sigma(z_8) \\ \sigma(z_9) \end{bmatrix}$$

Ensuite, adaptons les poids et le biais en conséquence. Jusqu'à présent, W est un vecteur de longueur 784. Au lieu de cela, nous allons faire de W une matrice avec des dimensions (10, 784). De cette manière, nous pouvons effectuer une multiplication matricielle entre W et un vecteur d'entrée x, c'est-à-dire Wx, dont le résultat sera un vecteur de longueur 10. En poursuivant, si nous faisons du terme de biais un vecteur de longueur 10, nous pouvons l'ajouter à Wx. Enfin, notez que nous pouvons calculer la sigmoïde d'un vecteur z en l'appliquant à chacune de ses composantes :

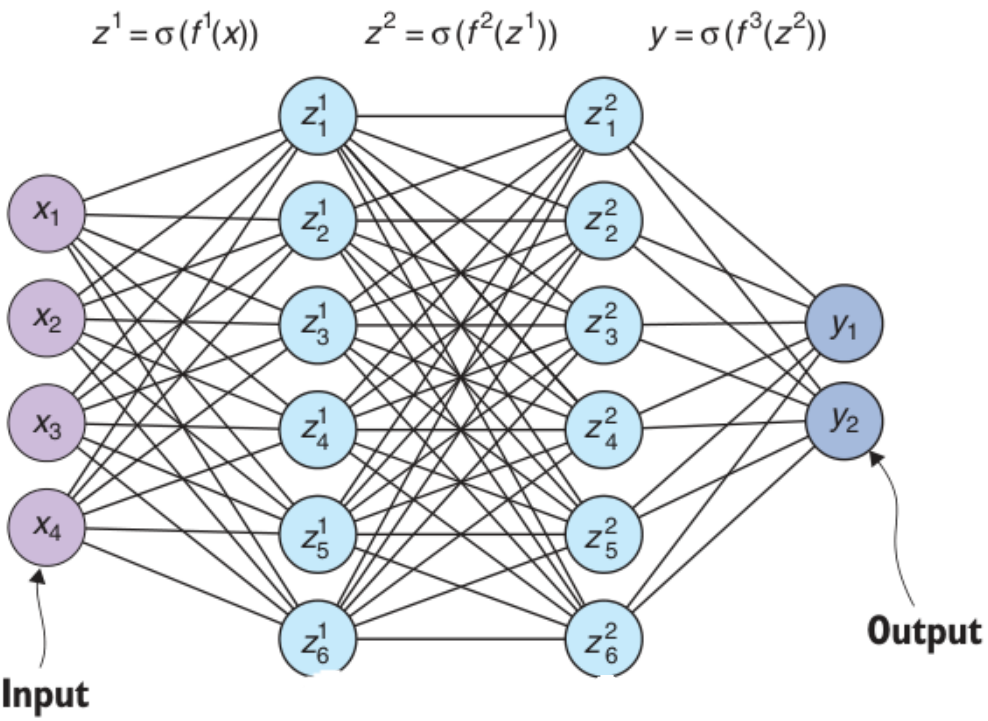


X= vecteur de dimension 4
Z= matrice W qui transforme x en un vecteur z de dimension 2
σ= Application de la fonction sigmoïde pour chaque élément de z
y= Sortie d'un vecteur de dimension 2

Cette manipulation nous permet de désormais transformer un vecteur d'entrée x vers un vecteur de sortie y, alors qu'auparavant, y était simplement une seule valeur. L'avantage de cela est que rien ne nous empêche de réaliser cette transformation de vecteur à vecteur plusieurs fois, construisant ainsi ce que nous appelons un réseau feed-forward.

Le réseau de neurones feed-forward

A l'inverse du réseau back forward que nous allons voir par la suite le réseau feed-forward ne fait que d'aller vers l'avant de couche en couche. Nous venons de voir que grâce à la mise en place de matrice nous pouvions réitérer une fonction à l'aide du résultat précédent. Ainsi rien nous empêche de créer de nouvelles fonctions pour améliorer la qualité de notre modèle.
Le réseau feed-forward se présenterai comme ça :



Avec une avancée vers la droite et z qui est une fonction d'activation.
Cependant malgré une multitude de couche on ne sait toujours pas évaluer la qualité de nos prédictions. Bien que la fonction que l'on avait fait avant marchait pour un seul chiffre, la dimension est toute autre avec plusieurs vecteurs à la fois et donc de matrice.
D'un point de vue mathématique la propagation se traduit par :
forward pass) pour la i-ème couche de notre réseau neuronal peut maintenant s'écrire comme suit :

$$y^{i+1} = \sigma\left(W^i y^i + b^i\right) = f^i \circ y^i$$

De manière récursive on peut maintenant écrire :

$$y = f^n \circ \cdots \circ f^1(x)$$

Étant donné que vous calculez votre fonction de perte (Loss) avec Loss

$$\sum_i \text{Loss}\left(W, b, X_i, \hat{y}_i\right) = \sum_i \text{Loss}\left(y_i, \hat{y}_i\right)$$

à partir des prédictions y et des étiquettes ŷ, on peut diviser la fonction de perte de manière similaire :

$$\text{Loss}\,(y, \hat{y}) = \text{Loss} \circ f^n \circ \cdots \circ f^1(x)$$

$$\frac{d\text{Loss}}{dx} = \frac{d\text{Loss}}{df^n} \cdot \frac{df^n}{df^{n-1}} \cdots \frac{df^2}{df^1} \cdot \frac{df^1}{dx}$$

Delta donc le gradient que nous allons voir juste après est alors :

$$\Delta^i = \frac{d\text{Loss}}{df^n} \cdots \frac{df^{i+1}}{df^i}$$

Fonction de perte

Pour quantifier dans quelle mesure nous nous sommes éloignés de notre cible avec notre prédiction, nous introduisons le concept de fonctions de perte.Une fonction de perte évalue l'adéquation des paramètres de votre algorithme, compte tenu des données fournis. L'objectif d'entraînement est de minimiser la perte en trouvant de bonnes stratégies pour adapter les paramètres.
La fonction de perte de l'erreur quadratique moyen. On mesure à quel point notre prédiction était proche de l'étiquette réelle en mesurant la distance au carré et en faisant la moyenne sur tous les exemples observés. On a

$$\text{MSE}\,(y, \hat{y}) = \frac{1}{2} \sum_{i=1}^{k} (y_i - \hat{y}_i)^2$$

Avec yi les résultats que l'on trouve et ŷ les résultats attendus (si on trouve 7 alors que l'étiquette était 8 on calcule l'écart sur le plan entre les probabilités d'avoir un 7 ou 8, par exemple l'écart entre un 1 et un 8 est important mais entre un 4 et un 9 est faible car il y a plus de similitude entre un 4 et un 9 qu'entre un 1 et un 8)
Sur Python cela donne :

```
class MSE:
    def __init__(self):
        pass

    @staticmethod
    def loss_function(predictions, labels):
        """
```

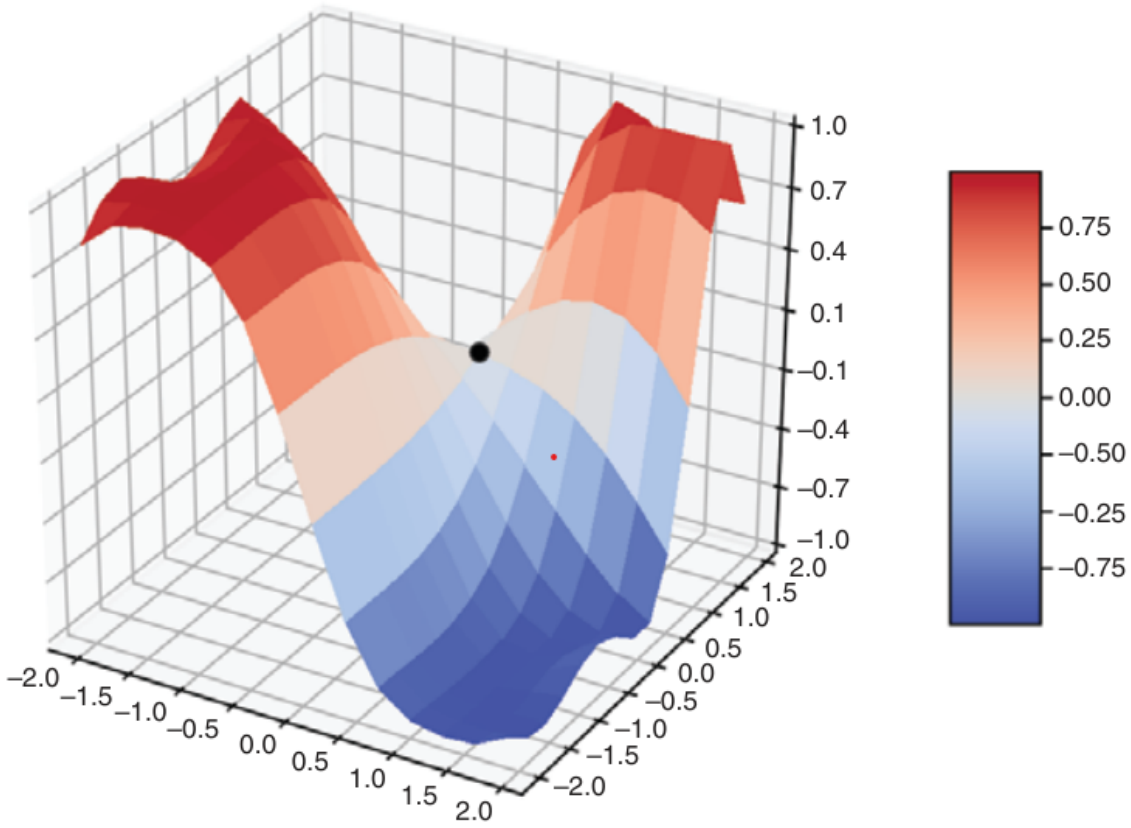
```
        Cette méthode calcule la fonction de perte MSE (Mean Squared Error)
entre les prédictions et les étiquettes
        """
        diff = predictions - labels  # Calcul de la différence entre les
prédictions et les étiquettes
        return 0.5 * sum(diff * diff)[0]  # Calcul de la fonction de perte MSE

    @staticmethod
    def loss_derivative(predictions, labels):
        """
        Cette méthode calcule la dérivée de la fonction de perte MSE par
rapport aux prédictions
        return predictions - labels  # Calcul de la dérivée de la fonction de
perte MSE par rapport aux prédictions

# Cette classe définit deux méthodes statiques : loss_function() et
loss_derivative(). La méthode loss_function() calcule la fonction de perte MSE
(Mean Squared Error) entre les prédictions et les étiquettes.
# La méthode loss_derivative() calcule la dérivée de la fonction de perte MSE
par rapport aux prédictions. Les deux méthodes prennent en entrée deux
tableaux 2D :
# predictions et labels, qui représentent respectivement les prédictions et
les étiquettes réelles.
```

On écrit la dérivée pour l'utiliser par la suite pour la méthode back-forward

La fonction de perte pour un ensemble de prédictions et d'étiquettes nous donne des informations sur la précision de nos paramètres de modèle. Plus la perte est petite, meilleures sont nos prédictions, et vice versa. La fonction de perte elle-même est une fonction de nos paramètres de réseau. Dans notre implémentation de l'erreur quadratique moyenne (MSE), nous ne fournissons pas directement les poids, mais ils sont implicitement donnés à travers les prédictions, car nous les utilisons pour les calculer.

En théorie, nous savons, grâce au calcul différentiel, que pour minimiser la perte, nous devons calculer sa dérivée et la mettre égale à zéro. Nous appelons l'ensemble des paramètres à ce point une solution. Calculer la dérivée d'une fonction et l'évaluer à un point spécifique s'appelle le calcul du gradient. Nous avons effectué la première étape du calcul de la dérivée dans notre implémentation de l'erreur quadratique moyenne, mais il y a encore plus à faire. Notre objectif est de calculer explicitement les gradients pour tous les termes de poids et de biais de notre réseau.

La figure ci-dessous montre une surface dans l'espace tridimensionnel. Cette surface peut être interprétée comme une fonction de perte pour une entrée bidimensionnelle. Les deux premiers axes représentent les poids, et le troisième axe orienté vers le haut indique la valeur de perte.



On voit qu'il y a un minimum dans la partie bleu. Notre but est de se rapprocher de ce minimum.
La descente de gradient pour trouver nos minima

De manière intuitive, lorsque nous calculons le gradient d'une fonction pour un point donné, ce gradient pointe dans la direction de la plus forte augmentation. En partant d'une fonction de perte, Loss, et d'un ensemble de paramètres W, l'algorithme de descente de gradient pour trouver un minimum pour cette fonction se déroule comme suit :

1. Calculons le gradient Δ de Loss à pour l'ensemble actuel de paramètres W (calculons la dérivée de Loss par rapport à chaque poids W).
En dérivant on obtient alors :

$$\Delta W^i = \frac{d\text{Loss}}{dW^i} = \Delta^i \cdot \left(y^i\right)^\top$$

2. Mettons à jour W en lui soustrayant Δ. Nous appelons cette étape suivre le gradient. Comme Δ pointe dans la direction de la plus forte augmentation, le soustraire nous mène dans la direction de la plus forte descente.

3. Répétons jusqu'à ce que Δ soit égal à 0 ou proche de 0.

Étant donné que notre fonction de perte est non négative, nous savons en particulier qu'elle possède un minimum. Elle pourrait avoir de nombreux minima, voire une infinité de minima. Par exemple, si nous pensons à une surface plane, chaque point de celle-ci est un minimum.

Cependant, ce minimum n'est pas forcément un extremum il peut être local comme on le voit avec le point noir sur la figure. Dans ce cas là la fonction MSE ne marche plus il faut alors avoir d'autres fonction de pertes pour augmenter sa précision mais nous n'allons pas le voir dans ce projet.

Pour calculer les gradients et appliquer la descente de gradient pour un réseau neuronal, il faudrait évaluer la fonction de perte et sa dérivée par rapport aux paramètres du réseau à chaque point de l'ensemble d'entraînement, ce qui est trop coûteux dans la plupart des cas. À la place, on utilisera une technique appelée descente de gradient stochastique (SGD). Pour exécuter le SGD, il faut d'abord sélectionner quelques échantillons de notre ensemble d'entraînement, qu'on appelle mini-lot (ou mini-batch). Pour les OCR la taille du mini-lot doit être égal à au nombre de grandeur de l'étiquette pour s'assurer que chaque étiquette soit dans le mini lot

Pour un réseau neuronal feed-forward donné avec n couches et un mini-lot de données d'entrée x1, ..., xk de taille k, on peut calculer le forward pass de notre réseau neuronal et calculer la perte pour ce mini-lot. Pour chaque échantillon xj de ce lot, on peut ensuite évaluer le gradient de notre fonction de perte par rapport à n'importe quel paramètre de notre réseau. Les gradients des poids et des biais dans la couche i sont appelés respectivement ΔjW i et Δjbi.

Pour chaque couche et chaque échantillon dans le lot, vous calculez les gradients respectifs et utilisez les règles de mise à jour suivantes pour les paramètres :

$$W^i \leftarrow W^i - \alpha \sum_{j=1}^{k} \Delta_j W^i$$

$$b^i \leftarrow b^i - \alpha \sum_{j=1}^{k} \Delta_j b^i$$

où Wi est la matrice de poids de la couche i, bi est le vecteur de biais de la couche i, ΔjWi et Δjbi sont les gradients calculés pour l'échantillon j et la couche i du mini-lot, et α est le taux d'apprentissage qui contrôle la taille des mises à jour de paramètres.

En résumé, le SGD est une méthode d'optimisation qui permet de calculer les gradients et de mettre à jour les paramètres du réseau de manière itérative en utilisant des mini-lots d'échantillons sélectionnés à partir de l'ensemble d'entraînement.

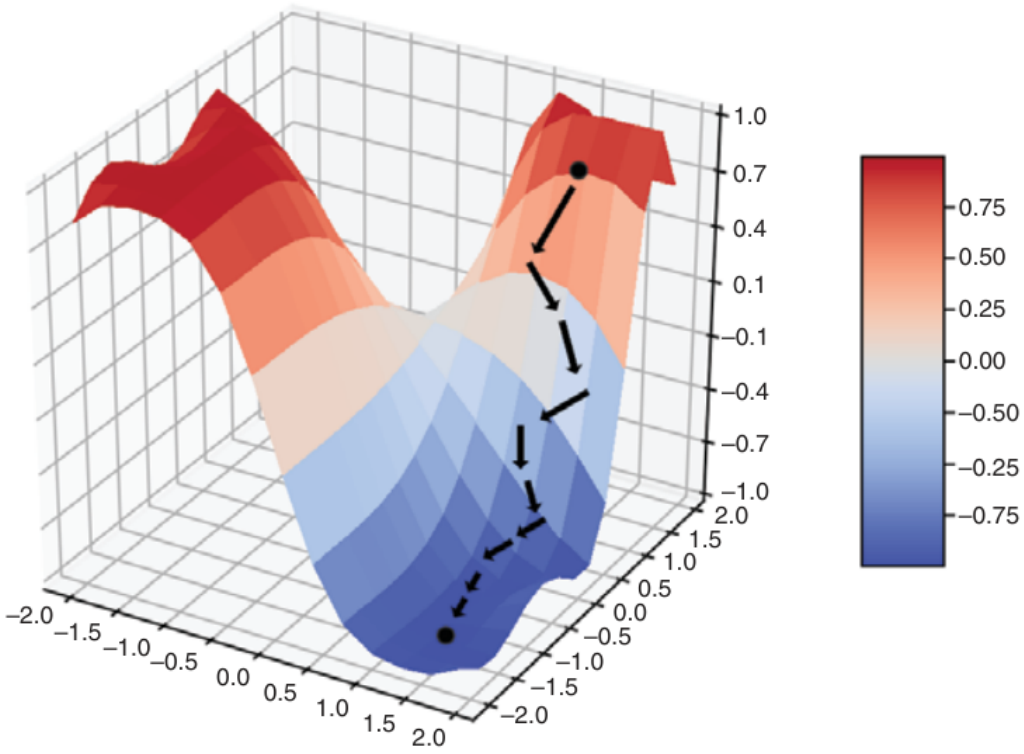
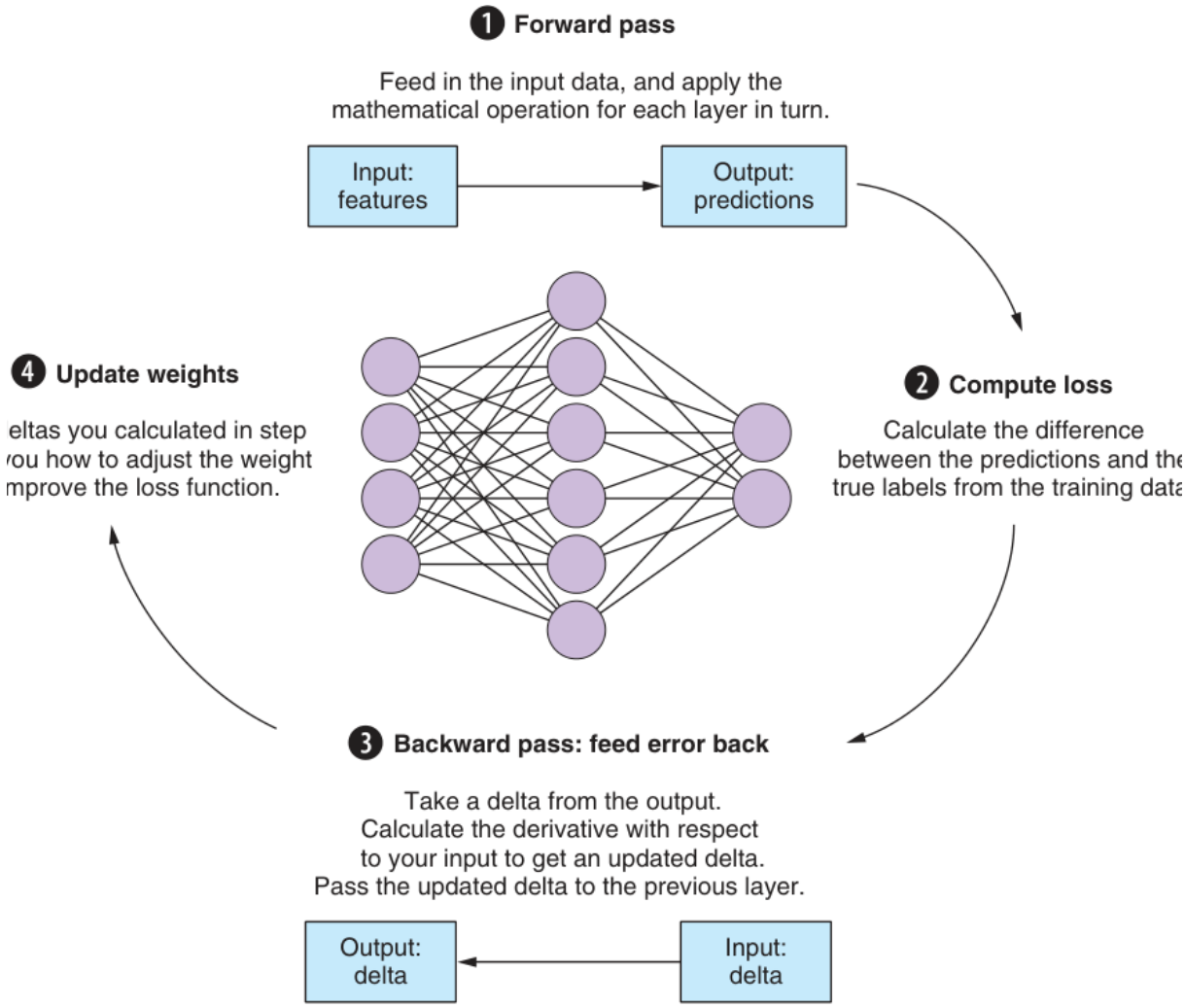


Figure représentant la méthode SGD.

Propagation des gradients à travers le réseau de neurones

Tout comme on peut faire passer les données d'entrée de couche en couche vers l'avant, on peut aussi faire passer les dérivées de couche en couche en sens inverse. On propage alors les dérivées à travers notre réseau, d'où le nom de rétropropagation (backpropagation).



Pour nous guider à travers la figure, prenons cela étape par étape :

1)Passe avant sur nos données d'entraînement. Dans cette étape, nous prenons un échantillon de données d'entrée x et le faisons passer à travers notre réseau pour obtenir une prédiction, comme suit :
a. Nous calculons la partie affine-linéaire : Wx + b.
b. Nous appliquons la fonction sigmoïde σ(x) au résultat. Notons que nous abusons légèrement de la notation en utilisant x dans une étape de calcul pour représenter la sortie du résultat précédent.
c. Nous répétons ces deux étapes jusqu'à arriver à la couche de sortie. Dans notre exemple, nous avons choisi deux couches, mais le nombre de couches n'a pas d'importance.

2)Évaluation de notre fonction de perte. Dans cette étape, nous prenons les étiquettes $\hat{y}$ correspondant à notre échantillon x et les comparons à nos prédictions y en calculant une valeur de perte. Dans notre exemple, nous choisissons l'erreur quadratique moyenne comme fonction de perte.

3)Rétropropagation des termes d'erreur. Dans cette étape, nous prenons notre valeur de perte et la faisons remonter à travers notre réseau. Nous le faisons en calculant les dérivées couche par couche,

ce qui est possible grâce à la règle de la chaîne. Cette règle permet d'expliciter la dérivée d'une fonction composée pour deux fonction dérivable. Alors que la forward pass fait passer les données d'entrée à travers notre réseau dans une direction, la backward pass fait remonter les termes d'erreur dans la direction opposée.
a. Nous propageons les termes d'erreur, ou deltas, notés par Δ, dans l'ordre inverse de la passe avant.
b. Pour commencer, nous calculons la dérivée de notre fonction de perte, qui sera notre Δ initial. Encore une fois, comme dans la forward pass, nous abusons de la notation et appelons le terme d'erreur propagé Δ à chaque étape du processus.
c. Nous calculons la dérivée de la sigmoïde par rapport à son entrée, qui est simplement σ·(1 - σ). Pour transmettre Δ à la couche suivante, nous pouvons effectuer une multiplication composante par composante : σ(1 - σ) · Δ.
d. La dérivée de notre transformation affine-linéaire Wx + b par rapport à x est simplement W. Pour transmettre Δ, nous calculons W^T · Δ.
e. Nous répétons ces deux étapes jusqu'à atteindre notre première couche du réseau.

4)Mise à jour de nos poids avec les informations de gradient. Dans la dernière étape, nous utilisons les deltas que nous avons calculés tout au long du processus pour mettre à jour les paramètres de notre réseau (poids et termes de biais).
a. La fonction sigmoïde n'a pas de paramètres, donc il n'y a rien à faire.
b. La mise à jour Δb que le terme de biais dans chaque couche reçoit est simplement Δ.
c. La mise à jour ΔW pour les poids dans une couche est donnée par Δ · x^T (nous devons transposer x avant de le multiplier par le delta).
d. Notez que nous avons commencé en disant que x est un seul échantillon. Cependant, tout ce que nous avons discuté s'applique également aux mini-lots. Si x représente un mini-lot d'échantillons (x est une matrice dans laquelle chaque colonne est un vecteur d'entrée), les calculs de forward and backward pass sont exactement les mêmes.

5) On recommence l'opération avec la mise à jour des poids et des biais. Jusqu'à avoir une bonne accuracy.

PYTHON

SequentialNetwork

Nous allons maintenant voire comment coder sur python toutes les choses théoriques que l'on a vu juste avant.

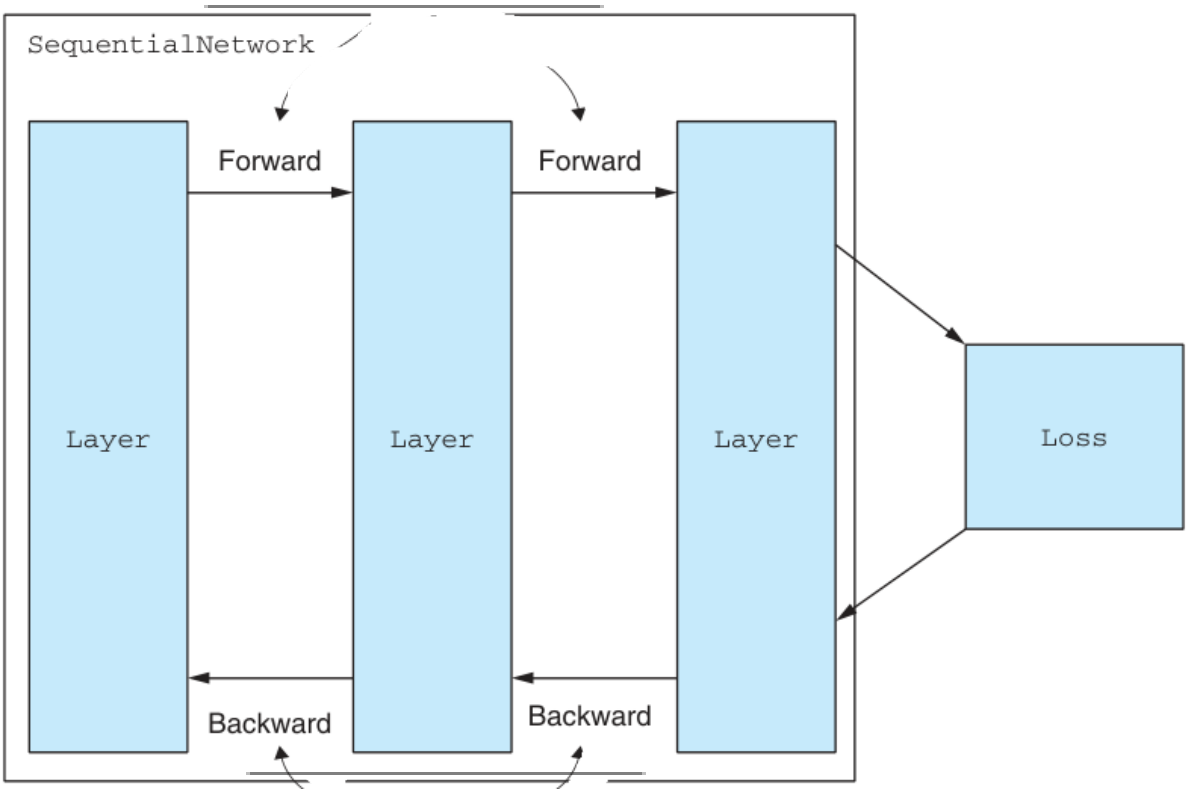


Diagramme de classes pour votre implémentation Python d'un réseau feed-forward. Un SequentialNetwork contient plusieurs instances de Layer. Chaque Layer implémente une fonction mathématique et sa dérivée. Les méthodes forward et backward implémentent respectivement les passes avant et arrière. Une instance de Loss calcule notre fonction de perte, l'erreur entre nos prédictions et nos données d'entraînement.

Voici le code complet détaillé pour comprendre chaque étape :
Base du principe de l'implémentation Layer

```
class Layer:
    def __init__(self):
        self.params = []    # Les paramètres de la couche, initialisés à une
liste vide

        self.previous = None   # Référence à la couche précédente
        self.next = None   # Référence à la couche suivante

        self.input_data = None   # Données en entrée de la couche. Chaque
couche peut conserver les données entrant et sortant dans le forward pass
        self.output_data = None  # Données en sortie de la couche

        self.input_delta = None   # Delta en entrée de la couche
        self.output_delta = None  # Delta en sortie de la couche

#Connection layer entre la prochaine couche et l'ancienne


    def connect(self, layer):
```

```
        self.previous = layer  # Connexion à la couche précédente
        layer.next = self      # Mise à jour de la référence à la couche
suivante

#Implémentation du principe de forward et backward passes



    def forward(self):
        raise NotImplementedError   # La méthode pour calculer la propagation
directe n'est pas implémentée. Chaque implémentation de couche doit fournir une
fonction pour faire avancer les données d'entrée.

    def get_forward_input(self):
        if self.previous is not None:
            return self.previous.output_data  # Récupération des données en
sortie de la couche précédente.
        else:
            return self.input_data   # Si c'est la première couche, on utilise
les données en entrée

    def backward(self):
        raise NotImplementedError  # La méthode pour calculer la
rétropropagation n'est pas implémentée
Les couches doivent mettre en œuvre la rétropropagation des termes d'erreur,
une méthode pour faire remonter les erreurs d'entrée à travers le réseau.

    def get_backward_input(self):
        if self.next is not None:
            return self.next.output_delta   # Récupération du delta en sortie
de la couche suivante
        else:
            return self.input_delta   # Si c'est la dernière couche, on
utilise le delta en entrée

    def clear_deltas(self):
        pass   # Effacer les deltas ne fait rien pour cette couche

    def update_params(self, learning_rate):
        pass   # La mise à jour des paramètres ne fait rien pour cette couche

    def describe(self):
        raise NotImplementedError  # La méthode pour décrire la couche n'est
pas implémentée


def sigmoid_prime_double(x):
    return sigmoid_double(x) * (1 - sigmoid_double(x))  # La fonction de
dérivée de la fonction sigmoid_double

def sigmoid_prime(z):
```

```
    return np.vectorize(sigmoid_prime_double)(z)  # Application vectorisée de
la fonction sigmoid_prime_double




# Ce code définit la classe Layer qui sera utilisée pour créer les différentes
couches du réseau de neurones.
# Elle est accompagnée de deux fonctions auxiliaires sigmoid_prime_double et
sigmoid_prime pour calculer la dérivée de la fonction sigmoïde.
# La classe Layer possède plusieurs attributs pour stocker les données, les
paramètres et les deltas,
# ainsi que des méthodes pour gérer les connexions entre les couches,
# effectuer la propagation directe et la rétropropagation, effacer les deltas
et mettre à jour les paramètres.
# Les méthodes forward, backward et describe sont des méthodes abstraites et
doivent être implémentées dans les classes dérivées.


class ActivationLayer(Layer):
    def __init__(self, input_dim):
        super(ActivationLayer, self).__init__()

        # Initialisation des dimensions d'entrée et de sortie de la couche
        self.input_dim = input_dim
        self.output_dim = input_dim

    def forward(self):
        # Récupération des données d'entrée
        data = self.get_forward_input()
        # Application de la fonction d'activation sigmoid à ces données
        self.output_data = sigmoid(data)

    def backward(self):
        # Récupération du delta de sortie de la couche suivante
        delta = self.get_backward_input()
        # Récupération des données d'entrée
        data = self.get_forward_input()
        # Calcul du delta de la couche actuelle à partir du delta de la couche
suivante et de la dérivée de la fonction d'activation sigmoid appliquée aux
données d'entrée
        self.output_delta = delta * sigmoid_prime(data)

    def describe(self):
        # Affichage des informations de la couche
        print("|-- " + self.__class__.__name__)
        print(" |-- dimensions: ({},{})"
            .format(self.input_dim, self.output_dim))
```

```
# La classe ActivationLayer hérite de la classe Layer et représente une couche
d'activation appliquant la fonction sigmoid à ses entrées.

# La méthode __init__ initialise les dimensions d'entrée et de sortie de la
couche.

# La méthode forward calcule la sortie de la couche en appliquant la fonction
sigmoid aux données d'entrée récupérées grâce à la méthode get_forward_input.

# La méthode backward calcule le delta de la couche en utilisant le delta de
la couche suivante et la dérivée de la fonction d'activation sigmoid appliquée
aux données d'entrée récupérées grâce à la méthode get_forward_input.

# La méthode describe affiche les informations de la couche.

class DenseLayer(Layer):
    def __init__(self, input_dim, output_dim, weight=None, bias=None):
        super(DenseLayer, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim

        if weight is not None:
            self.weight = weight
        else:
            self.weight = np.random.randn(output_dim, input_dim) /
np.sqrt(input_dim)

        if bias is not None:
            self.bias = bias
        else:
            self.bias = np.random.randn(output_dim, 1)

        # Stockage des poids et des biais dans une liste pour pouvoir les
mettre à jour lors de l'apprentissage
        self.params = [self.weight, self.bias]

        # Initialisation des deltas pour les poids et les biais
        self.delta_w = np.zeros(self.weight.shape)
        self.delta_b = np.zeros(self.bias.shape)

    def forward(self):
        # Récupération des données en entrée
        data = self.get_forward_input()

        # Calcul de la sortie en multipliant les poids par les données en
entrée et en ajoutant les biais
        self.output_data = np.dot(self.weight, data) + self.bias

    def backward(self):
        # Récupération des données en entrée et des deltas en sortie
        data = self.get_forward_input()
```

```
        delta = self.get_backward_input()

        # Calcul du delta pour les biais en ajoutant les deltas en sortie
        self.delta_b += delta

        # Calcul du delta pour les poids en multipliant les deltas en sortie
par les données en entrée transposées
        self.delta_w += np.dot(delta, data.transpose())

        # Calcul du delta en entrée pour la couche précédente en multipliant
les poids transposés par les deltas en sortie
        self.output_delta = np.dot(self.weight.transpose(), delta)

    def update_params(self, rate):
        # Mise à jour des poids et des biais en soustrayant les deltas
multipliés par le taux d'apprentissage
        self.weight -= rate * self.delta_w
        self.bias -= rate * self.delta_b

    def clear_deltas(self):
        # Réinitialisation des deltas pour les poids et les biais
        self.delta_w = np.zeros(self.weight.shape)
        self.delta_b = np.zeros(self.bias.shape)

    def describe(self):
        # Affichage du nom de la couche et des dimensions des données en
entrée et en sortie
        print("|--- " + self.__class__.__name__)
        print(" |-- dimensions: ({},{})"
            .format(self.input_dim, self.output_dim))

# La classe DenseLayer est une couche dense, c'est-à-dire qu'elle connecte
chaque neurone de la couche précédente à chaque neurone de la couche suivante.
Les poids et les biais sont initialisés avec des valeurs aléatoires, et les
deltas sont initialisés à zéro.

# La méthode forward calcule la sortie de la couche en multipliant les poids
par les données en entrée et en ajoutant les biais.

# La méthode backward calcule les deltas pour les poids et les biais en
utilisant les deltas en sortie et les données en entrée. Elle calcule
également le delta en entrée pour la couche précédente.

# La méthode update_params met à jour les poids et les biais en soustrayant
les deltas multipliés par le taux d'apprentissage.

# La méthode clear_deltas(self) réinitialise les deltas accumulés pendant la
phase de rétropropagation, pour éviter qu'ils ne s'accumulent à chaque
itération de l'entraînement.

# Enfin, la méthode describe(self) permet d'afficher les informations de base
sur le layer, telles que ses dimensions.
```

```
class SequentialNetwork:
    def __init__(self, loss=None):
        print("Initialize Network...")
        self.layers = []
        if loss is None:
            self.loss = MSE()  # initialisation de la fonction de coût à MSE
si elle n'est pas fournie

    def add(self, layer):
        self.layers.append(layer)  # ajout d'une couche à la liste des couches
        layer.describe()  # affichage de la description de la couche
        if len(self.layers) > 1:
            self.layers[-1].connect(self.layers[-2])  # connexion de la couche
courante à la couche précédente

    def train(self, training_data, epochs, mini_batch_size, learning_rate,
test_data=None):
        n = len(training_data)
        for epoch in range(epochs):
            random.shuffle(training_data)  # mélange des données
            mini_batches = [training_data[k:k + mini_batch_size] for k in
range(0, n, mini_batch_size)]
            # création de mini-lots pour l'entraînement
            for mini_batch in tqdm(mini_batches, desc=f"Epoch
{epoch+1}/{epochs}", unit="batch"):
                self.train_batch(mini_batch, learning_rate)  # entraînement
pour chaque mini-lot
            if test_data:
                n_test = len(test_data)
                print("Epoch {0}: {1} / {2}".format(epoch,
self.evaluate(test_data), n_test))
                # évaluation sur les données de test à chaque fin d'époque
            else:
                print("Epoch {0} complete".format(epoch))

    def train_batch(self, mini_batch, learning_rate):
        self.forward_backward(mini_batch)  # propagation et rétropropagation
pour chaque mini-lot

        self.update(mini_batch, learning_rate)  # mise à jour des paramètres
pour chaque mini-lot

    def update(self, mini_batch, learning_rate):
        learning_rate = learning_rate / len(mini_batch)
        for layer in self.layers:
            layer.update_params(learning_rate)  # mise à jour des paramètres
pour chaque couche
        for layer in self.layers:
            layer.clear_deltas()  # réinitialisation des deltas pour chaque
couche
```

```
    def forward_backward(self, mini_batch):
        for x, y in mini_batch:
            self.layers[0].input_data = x
            for layer in self.layers:
                layer.forward()  # propagation avant pour chaque couche
            self.layers[-1].input_delta =
self.loss.loss_derivative(self.layers[-1].output_data, y)
            # calcul de la dérivée de la fonction de coût pour la dernière
couche
            for layer in reversed(self.layers):
                layer.backward()  # rétropropagation pour chaque couche en
partant de la dernière

    def single_forward(self, x):
        self.layers[0].input_data = x
        for layer in self.layers:
            layer.forward()  # propagation avant pour chaque couche
        return self.layers[-1].output_data

    def evaluate(self, test_data):
        test_results = [(np.argmax(self.single_forward(x)), np.argmax(y)) for
(x, y) in test_data]
        # prédiction pour chaque exemple de test et comparaison avec la
réponse attendue
        return sum(int(x == y) for (x, y) in test_results)  # somme des
prédictions correctes

    def save_model(self, file_path):
        model_data = []
        for layer in self.layers:
            if isinstance(layer, DenseLayer):
                model_data.append({
                    'type': 'DenseLayer',
                    'input_dim': layer.input_dim,
                    'output_dim': layer.output_dim,
                    'weight': layer.weight,
                    'bias': layer.bias
                })
            elif isinstance(layer, ActivationLayer):
                model_data.append({
                    'type': 'ActivationLayer',
                    'input_dim': layer.input_dim
                })
        with open(file_path, 'wb') as f:
            pickle.dump(model_data, f)

def load_data():
    training_data, test_data = load_data_impl()
    return shape_data(training_data), shape_data(test_data)

training_data, test_data = load_data()
```

```
# Le code suivant crée une instance de SequentialNetwork et y ajoute plusieurs couches de neurones à l'aide des classes DenseLayer et ActivationLayer.
# Il entraîne ensuite le réseau de neurones sur les données training_data pendant un certain nombre d'epochs,
# en utilisant une taille de mini-batch de 10, un taux d'apprentissage de 3.0 et les données test_data pour évaluer les performances.

# net = SequentialNetwork()
# net.add(DenseLayer(784, 392))  # Ajout d'une couche DenseLayer avec 784 entrées et 392 sorties
# net.add(ActivationLayer(392)) # Ajout d'une couche d'activation pour la couche précédente
# net.add(DenseLayer(392, 196)) # Ajout d'une couche DenseLayer avec 392 entrées et 196 sorties
# net.add(ActivationLayer(196)) # Ajout d'une couche d'activation pour la couche précédente
# net.add(DenseLayer(196, 10))  # Ajout d'une couche DenseLayer avec 196 entrées et 10 sorties
# net.add(ActivationLayer(10))  # Ajout d'une couche d'activation pour la couche précédente

# # Entraînement du réseau de neurones

# net.train(training_data, epochs=10, mini_batch_size=10, learning_rate=3.0, test_data=test_data)


# Explications :

# La première ligne crée une instance de la classe SequentialNetwork qui représente un réseau de neurones séquentiel.
# Les six lignes suivantes ajoutent des couches de neurones au réseau à l'aide des classes DenseLayer et ActivationLayer.
# Chaque couche DenseLayer est suivie d'une couche ActivationLayer qui applique une fonction d'activation à la sortie de la couche précédente.
# Ces couches sont créées avec des dimensions spécifiques, qui dépendent des dimensions de l'entrée et de la sortie de chaque couche.
# La première couche a 784 entrées (la taille des images MNIST) et 392 sorties, la deuxième couche a 392 entrées et 196 sorties, et la dernière couche a 196 entrées et 10 sorties (le nombre de classes pour la classification MNIST).
# La dernière ligne entraîne le réseau de neurones sur les données training_data pendant cinq epochs en utilisant une taille de mini-batch de 10, un taux d'apprentissage de 3.0 et les données test_data pour évaluer les performances.
# Pendant l'entraînement, les données d'entraînement sont mélangées et divisées en mini-batchs.
# Pour chaque mini-batch, le réseau de neurones est entraîné en appelant la méthode train_batch(), qui effectue une propagation avant et une propagation arrière à travers le réseau pour calculer les gradients des paramètres du réseau.
```

```
# Les paramètres sont ensuite mis à jour en appelant la méthode
update_params() pour chaque couche du réseau. La méthode clear_deltas() est
également appelée pour chaque couche pour effacer les gradients des paramètres
calculés pendant le mini-batch.
# En fin d'époch, les performances du réseau sont évaluées en appelant la
méthode evaluate() avec les données de test et en comparant les sorties du
réseau avec les étiquettes de classe réelles


# Définition d'une fonction nommée compute_model_accuracy qui prend deux
arguments : un objet de réseau neuronal et un ensemble de données de test.
def compute_model_accuracy(network, test_data):
    # Initialisation des compteurs pour le nombre de prédictions correctes et
le nombre total d'échantillons de test.
    correct_predictions = 0
    total_samples = len(test_data)

    # Boucle pour chaque échantillon dans l'ensemble de données de test. x est
l'entrée du réseau neuronal et y est la sortie attendue.
    for x, y in test_data:
        # Utilisation du réseau neuronal pour prédire l'étiquette de x et
extraction de l'étiquette véritable de y.
        predicted_label = np.argmax(network.single_forward(x))
        true_label = np.argmax(y)

        # Vérification si la prédiction du réseau neuronal est correcte. Si
c'est le cas, le compteur des prédictions correctes est incrémenté.
        if predicted_label == true_label:
            correct_predictions += 1

    # Calcul de l'exactitude du modèle en divisant le nombre de prédictions
correctes par le nombre total d'échantillons de test. Affichage du résultat à
l'écran avec deux décimales.
    accuracy = correct_predictions / total_samples
    print(f"Model accuracy: {accuracy * 100:.2f}%")


# Définition d'une fonction nommée load_model qui prend un seul argument : le
chemin du fichier de modèle.
def load_model(file_path):
    # Ouverture du fichier de modèle en mode binaire pour la lecture et
chargement des données de modèle à partir du fichier en utilisant le module
pickle.
    with open(file_path, 'rb') as f:
        model_data = pickle.load(f)

    # Création d'un nouvel objet SequentialNetwork qui sera rempli avec les
couches du modèle chargé.
    loaded_model = SequentialNetwork()

    # Boucle pour chaque couche dans les données de modèle.
    for layer_data in model_data:
```

```
        # Ajout d'une nouvelle couche à l'objet loaded_model selon le type de
couche stocké dans les données de modèle.
        if layer_data['type'] == 'DenseLayer':
            loaded_model.add(DenseLayer(layer_data['input_dim'],
layer_data['output_dim'], layer_data['weight'], layer_data['bias']))
        elif layer_data['type'] == 'ActivationLayer':
            loaded_model.add(ActivationLayer(layer_data['input_dim']))

    # Renvoi de l'objet loaded_model rempli avec les couches du modèle chargé.
    return loaded_model


# Chargement du modèle stocké dans le fichier Model.pkl
loaded_model = load_model('Model.pkl')

# Calcul de l'exactitude du modèle chargé sur les données de test
compute_model_accuracy(loaded_model, test_data)



# Continuation de l'entraînement du modèle chargé en utilisant la méthode
train() de l'objet loaded_model.
loaded_model.train(training_data, epochs=5, mini_batch_size=10,
learning_rate=3.0, test_data=test_data)

# Calcul de l'exactitude du modèle chargé et entraîné sur les données de test.
compute_model_accuracy(loaded_model, test_data)

# Sauvegarde du modèle mis à jour dans un fichier Model_updated.pkl.
loaded_model.save_model('Model_updated.pkl')
```

Le résultat du modèle est assez faible pour l'avoir fait tourner 5h j'ai eu à mon maximum 92% de précision ce qui est correct mais assez faible. Cela peut s'expliquer par le fait que python n'était pas sur la même version et que donc les problèmes que j'ai pu avoir avec ont pu fausser le réseau ou la base de données. Il peut y avoir aussi une malchance avec l'apparition constamment d'un minimum local.

Le résumé du chapitre nous donne alors les principales notions que l'on a apprises avec le livre :
Résumé :
- Un réseau neuronal séquentiel est un réseau neuronal artificiel simple construit à partir d'une pile linéaire de couches. Les réseaux neuronaux peuvent être appliqués à une grande variété de problèmes d'apprentissage automatique, y compris la reconnaissance d'images.
- Un réseau feed-forward est un réseau séquentiel composé de couches denses avec une fonction d'activation.
- Les fonctions de perte évaluent la qualité de nos prédictions. L'erreur quadratique moyenne est l'une des fonctions de perte les plus couramment utilisées en pratique. Une fonction de perte vous donne une manière rigoureuse de quantifier la précision de votre modèle.

- La descente de gradient est un algorithme pour minimiser une fonction. La descente de gradient consiste à suivre la pente la plus raide d'une fonction. En apprentissage automatique, vous utilisez la descente de gradient pour trouver les poids du modèle qui donnent la plus petite perte.
- La descente de gradient stochastique est une variation de l'algorithme de descente de gradient. Dans la descente de gradient stochastique, vous calculez le gradient sur un petit sous-ensemble de votre ensemble d'entraînement appelé mini-batch, puis mettez à jour les poids du réseau en fonction de chaque mini-batch. La descente de gradient stochastique est généralement beaucoup plus rapide que la descente de gradient régulière sur de grands ensembles d'entraînement.
- Avec un réseau neuronal séquentiel, vous pouvez utiliser l'algorithme de rétropropagation pour calculer le gradient efficacement. La combinaison de la rétropropagation et des mini-batches rend l'entraînement suffisamment rapide pour être utilisable sur des ensembles de données massifs.

Conclusion

Il est évident que ce livre a encore beaucoup à apprendre sur les intelligences artificielles n'ayant traité qu'un chapitre sur 14. Cependant, le livre mets en perspective une bonne approche pour commencer à se former dans le monde de l'IA. Dans l'ensemble ce mémoire même s'il était plus un mémoire pour apprendre des connaissances poussées et des capacités restent très intéressant pour comprendre les enjeux et l'envers du décor des intelligences artificielles que l'on voit apparaître en ce moment. Le but recherché était de créer une IA sachant reconnaître l'écriture manuscrite des chiffres et des lettres en combinant deux IA d'OCR afin de traduire des notes manuscrites sur le format TEX. Même si le but n'a pas été atteint les bases pour commencer un nouveau projet cette fois ci avec plus de production n'ont été que renforcer voir battit.
Néanmoins, malgré l'importance d'avoir des logiciels indépendants qui ne collectent aucunes données, il est toutefois difficile de ne pas voir que le monde de l'IA artificielle se développe exponentiellement actuellement. On peut alors se demander si dans l'espace de mois voir de semaines mon projet ne sera pas déjà produit par une IA spécialisée dans les OCR ou bien par une intelligence artificielle généralisée type Open AI avec ChatGPT.
J'ai d'ailleurs demandé à GPT4 de faire une IA qui utilise MNIST pour reconnaitre des chiffres manuscrits. Voila son code fait en 5 min qui avait une accuracy de 99.99% à partir de 10 époques :

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
from keras.utils import to_categorical

# Charger les données MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Redimensionner les données d'entrée
X_train = X_train.reshape(-1, 784).astype('float32') / 255
X_test = X_test.reshape(-1, 784).astype('float32') / 255

# Convertir les étiquettes en catégories one-hot
y_train = to_categorical(y_train, num_classes=10)
```

```
y_test = to_categorical(y_test, num_classes=10)

# Créer le modèle
model = Sequential()
model.add(Dense(392, input_shape=(784,), activation='relu'))
model.add(Dense(196, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compiler le modèle
model.compile(loss='categorical_crossentropy',
optimizer=SGD(learning_rate=0.01), metrics=['accuracy'])

# Entraîner le modèle
model.fit(X_train, y_train, batch_size=10, epochs=10, validation_data=(X_test,
y_test))

# Sauvegarder le modèle
model.save('Model_keras.pkl')

# from keras.models import load_model

# # Charger le modèle sauvegardé
# loaded_model = load_model("Model_keras.pkl")

# # Vérifier l'accuracy du modèle chargé sur les données de test
# loaded_model_acc = loaded_model.evaluate(X_test, y_test, verbose=0)[1]
# print(f"Précision du modèle chargé sur l'ensemble de test :
{loaded_model_acc * 100:.2f}%")
```

Il est alors toujours intéressant de savoir utiliser ces outils en encadrant leur utilisation pour ne pas dévoiler des données sensibles à une entreprise qui ne se gène pas pour en collecter.
On peut alors se demander si dans un futur plus ou moins proche le monde numérique sera dominé par des oligopoles d'entreprise d'IA comme l'a été le monde avec les GAFAM.

Bibliographie:

Comme on l'a vue précédemment le mémoire se base principalement sur le livre « Python au lycée 2, livre/cours d'Arnaud Baudin » et le livre « Deep Learning and the Game of Go, Max Pumperla ».
L'utilisation de l'intelligence artificielle chatGPT a aussi été notable notamment pour permettre le changement de code entre les versions non compatible de Python, aucun forum sur internet ne faisait référrence à ce problème précisément.
On a aussi utilisé les définitions d'open classroom pour les neurones:
https://openclassrooms.com/fr/courses/5801891-initiez-vous-au-deep-learning/5801898-decouvrez-le-neurone-formel
Wikipédia a aussi été utilisé: https://fr.m.wikipedia.org/wiki/Apprentissage_profond
Le forum github m'a permis de mieux exploiter la base de données MNIST:
https://github.com/topics/mnist-dataset