UNIVERSITÉ DE MONTPELLIER

Département Informatique

HMIN201

Rapport du Projet

CNN pour la classification d'images

MASTER 1 DECOL

GOFOCUS

Étudiants :
Meriem AMERAOUI
Dounia BELABIOD
Jihene BOUHLEL
Bahaa Eddine NIL

Encadrant :
Pascal PONCELET

Année 2019/2020

Nous tenons à remercier notre encadrant, monsieur Pascal Poncelet pour son accompagnement et ses conseils qui nous ont été précieux tout au long du projet.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Problématique et solutions

Les réseaux de neurones sont utilisés depuis de très nombreuses années et font actuellement partie des algorithmes d'apprentissage automatique les plus populaires.

Bien que ces derniers donnent de bons résultats sur certains jeux de données en termes de précision et de vitesse, les réseaux de neurones convolutifs sont plus efficaces pour la tâche de reconnaissance d'image, grâce à leur architecture conçue pour traiter, corréler et comprendre efficacement la quantité de données dans des images à haute résolution.

Les réseaux de neurones convolutifs sont apparus progressivement suite à différents travaux : Les premiers travaux ont eu lieu en 1998, mais ils ont connu un énorme succès après 2012, lorsqu'un réseau de neurones convolutifs appelé AlexNet a réalisé une performance remarquable dans la classification des images de ImageNet en passant de 26% de taux d'erreur à 15.3%. L'évolution des CNN ne s'arrête pas en 2012, elle continue pour arriver en 2015 à un taux d'erreur de 3.6% réalisé par ResNet(152)[1][2].

Une application d'un réseau de neurones convolutifs sur la reconnaissance faciale de 5 600 images fixes de plus de 10 sujets, a rapporté un taux de reconnaissance de 97,6% selon l'article "Subject independent facial expression recognition with robust face detection using a convolutional neural network", ce dernier a enregistré une forte diminution du taux d'erreur. D'autres applications ont été effectuées pour l'évaluation de la qualité vidéo de manière objective après une formation manuelle, le système résultant avait une erreur quadratique moyenne très faible.[3]

1.2 Objectif du projet

Dans le cadre de notre TER de la première année du master informatique DECOL, nous avons eu pour projet « Comment les réseaux de neurones convolutifs obtiennent de si bons résultats en classification d'images ? ».

Notre travail comporte deux grandes parties importantes : compréhension du fonctionnement des réseaux de neurones convolutifs et réalisation de classes pour pouvoir constituer un tel réseau.

Dans un premier temps, nous avons compris comment fonctionnent réellement les réseaux de neurones, à quoi correspondent leurs étapes, leurs fonctions, leurs paramètres etc.., pour pouvoir ensuite faire des modifications afin de les adapter à nos besoins de classification, car souvent malheureusement, des applications sont réalisées en s'inspirant des lignes de code qui sont simplement recopiées sans savoir ce qu'il se passe réellement derrière. Les conséquences sont que quand le réseau ne fonctionne pas bien, les paramètres sont changés jusqu'à ce que cela marche mais rien n'assure que le modèle appris est généralisable.

1.3 Plan

Le reste du mémoire est organisé de la manière suivante :
Dans le chapitre 2, nous présentons brièvement le fonctionnement d'un réseau de neurones convolutifs. Ensuite, Nous présentons les outils qui nous ont permis de réaliser ce projet dans le chapitre 3. Au cours du chapitre 4, nous décrivons le travail réalisé pour concevoir des classes permettant de créer un CNN. Puis, le chapitre 5 décrira les expérimentations menées avec nos classes ainsi qu'une comparaison avec Keras (qui est une bibliothèque open source du deep learning écrite en python). La manière dont nous nous sommes organisés est précisée dans le chapitre 5. Enfin, nous concluons ce rapport en proposant des extensions possibles et précisons les difficultés rencontrées.

Nous avons mis à disposition un dépôt GitHub contenant tous les fichiers de notre projet ainsi qu'une vidéo de démonstration . Consultable via ce lien :
`https://github.com/nilbahaaeddine/M1_CNN.git`

Chapitre 2

Les réseaux de neurones à convolutions

Dans ce chapitre, nous allons présenter ce que c'est un réseau de neurones et un réseau de neurones à convolutions.

2.1 Les réseaux de neurones artificiels

Un réseau de neurones artificiels (désigné par l'acronyme *ANN*, de l'anglais *Artificial Neural Network*) est un système informatique qui s'inspire du fonctionnement du cerveau humain pour apprendre. [4]
L'objectif des réseaux de neurones artificiels est d'apprendre à reconnaître les schémas de données, une fois ce dernier formé sur des échantillons de données, il pourra faire des prédictions en détectant des modèles similaires dans les données futures.[5]

2.2 Limites des ANN

Les images utilisées pour les problèmes de visualisations sont grandes (volumineuses) et souvent de 224 x 224 ou plus. Pour traiter ces mêmes images en couleurs, avec les trois canaux de couleur (RVB), un réseau neuronal artificiel aura besoin de 224 x 224 x 3 ce qui nous donne 150 528 fonctions d'entrée. Une couche cachée typique dans un tel réseau peut avoir 1024 noeuds, nous devons donc former 150 528 x 1024 ce qui fait un total de plus de 150 millions de poids pour la première couche uniquement. Notre réseau serait énorme et presque impossible à former.

De plus, un réseau de neurones aura du mal à détecter des objets indépendamment de leurs positions dans une image.

Nous allons présenter dans la section suivante une solution pour atténuer ce problème que rencontrent les *ANN*.

2.3 Les réseaux de neurones à convolutions

Le réseau de neurones à convolutions (désigné par lacronyme *CNN*, de langlais *Convolutional Neural Network*) est l'un des modèles les plus performants pour effectuer de la reconnaissance et les classifications d'images, la détection d'objets et la reconnaissance des visages, etc[6].

Un être humain voit les images avec des formes et des couleurs, tandis que l'ordinateur, lui ne voit qu'un ensemble de pixels ou simplement des tableaux à deux dimensions de nombres.

Le *CNN* reçoit ainsi une image (vue comme une matrice), et applique deux sortes dopérations :

— Les filtres sont appliqués sur une image pour détecter les *Features*, ces derniers désignent les zones intéressantes de l'image numérique. Ces zones peuvent correspondre à des contours, les arêtes verticales, les variations de coloration, etc. Sachant que plus les filtres interviennent tard dans lenchaînement, plus ils permettent de détecter des formes complexes et plus abstraites.

— Les simplifications, pour alléger les calculs et dégager les informations les plus pertinentes.

Ensuite, le CNN donnera la probabilité de ce que cette image représente grâce à un réseau de neurones artificiels.

2.3.1 Les différentes couches du CNN

Il existe plusieurs types de couches pour un *CNN*.

2.3.1.1 La couche de convolution

La couche de convolution est la composante clé des réseaux de neurones à convolutions, et constitue toujours au moins leur première couche. Son but est de repérer la présence d'un ensemble de *features* dans les images reçues en entrée.

La couche de convolution reçoit donc en entrée une image, et calcule sa convolution en appliquant un nombre de filtres prédéfini. Nous obtenons pour chaque paire (image, filtre) une carte d'activation, ou *feature map*, qui nous indique où se situent les *features* dans l'image. Les *features* ne sont pas prédéfinis mais appris par le réseau lors de la phase d'entraînement. Les noyaux des filtres désignent les poids de la couche de convolution. Ils sont initialisés puis mis à jour par rétropropagation du gradient (expliquée dans la section 2.4.2).

Dans la figure 2.1, nous convoluons une image de taille 4x4 et un filtre de taille 3x3 pour produire une image en sortie de taille 2x2 qui est notre *feature map*. Cela se fait en quatre étapes :

1. Nous commençons par superposer notre filtre dans le coin supérieur gauche de l'image.
2. Nous effectuons une multiplication par élément entre les valeurs de l'image et les valeurs du filtre.
3. Nous résumons tous les résultats. Ce qui fait un calcul de :

   50*1+200*2+235*3+15*4+...=3447.

4. Nous plaçons notre résultat dans le pixel du *feature map*.
5. Nous glissons le filtre sur l'image par le pas indiqué par le paramètre *stride* de la convolution. Et nous revenons à la première étape jusqu'à ce que nous aurons parcouru toute l'image.

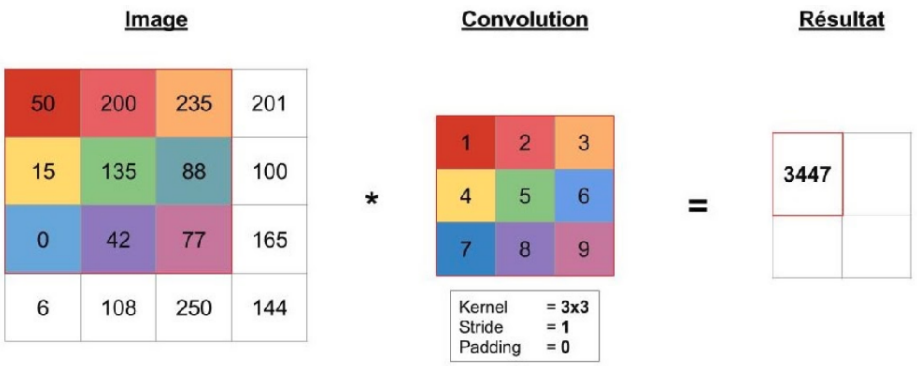


FIGURE 2.1 – Application d'un filtre 3x3 sur une image

Il faut noter que, plus nous faisons des convolutions sur l'image, plus nous serons capables de détecter des *patterns* (ensemble de *features*). Par exemple au début nous détecterons que les bordures, puis des formes plus précises. Les poids de chaque filtres vont être entraînés comme un réseau de neurones classique.[7]

2.3.1.2 La couche de correction

Dans ce projet nous allons nous intéresser à la couche ReLU, mais il existe d'autres couches pour faire de la correction.
Pour améliorer l'efficacité du traitement, il est préférable d'intercaler une couche qui va opérer une fonction d'activation sur les sorties de la couche précédente. Dans ce cadre nous trouvons *ReLU* (Rectified Linear Units) désigne la fonction réelle non-linéaire définie par **ReLU(x) = maxgu(0, x)**.
Comme nous pouvons le voir dans la figure 2.2, la couche de correction ReLU remplace donc toutes les valeurs négatives reçues en entrée par des zéros.

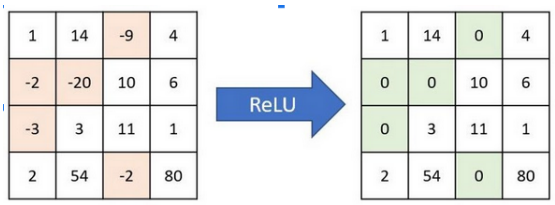


FIGURE 2.2 – Relu

2.3.1.3 La couche de pooling

Cette couche est souvent placée entre deux couches de convolution : elle reçoit en entrée plusieurs *feature maps*, et applique à chacune d'entre elles l'opération de pooling.

L'opération de pooling consiste à réduire la taille des images, tout en préservant leurs caractéristiques importantes.

Pour cela, nous découpons l'image en cellules régulières selon la taille du pool, puis nous gardons au sein de chaque cellule la valeur maximale (dans ce cas nous parlons de Max Pooling) ou nous calculons la moyenne (dans ce cas nous parlons d'average Pooling).
La figure 2.3 illustre ces deux phénomènes. Dans le cas d'un max pooling, la valeur

récupérée de la première cellule de taille 2x2 est la valeur maximale qui est égale à 20. Dans le cas d'un average pooling, la valeur récupérée de la première cellule est la moyenne des pixels de la cellule qui est égale à 13.
Nous faisons de même pour le reste des cellules.

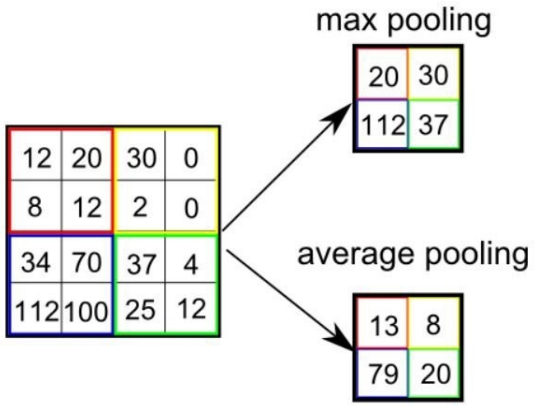


FIGURE 2.3 – Application du Max Pooling et de Average Pooling

En pratique, nous utilisons souvent des cellules carrées de petite taille pour ne pas perdre trop d'informations. Les choix les plus communs sont les cellules adjacentes de taille 2x2 pixels qui ne se chevauchent pas, ou des cellules de taille 3x3 pixels, dans ce cas il y a un chevauchement de 2 pixels entre les différentes cellules.

Nous obtenons en sortie le même nombre de *feature maps* qu'en entrée, mais celles-ci sont bien plus petites.

La couche de pooling permet de réduire le nombre de paramètres et de calculs dans le réseau. Nous améliorons ainsi l'efficacité du réseau et évitons le surapprentissage.

2.3.1.4 La couche flattening

Le flattening (ou mise à plat) consiste simplement à mettre bout à bout toutes les images que nous avons en sortie du pooling sous forme de matrices, pour en faire un seul vecteur. Les pixels sont récupérés ligne par ligne et ajoutés au vecteur final. La figure 2.4 modélise cette étape.

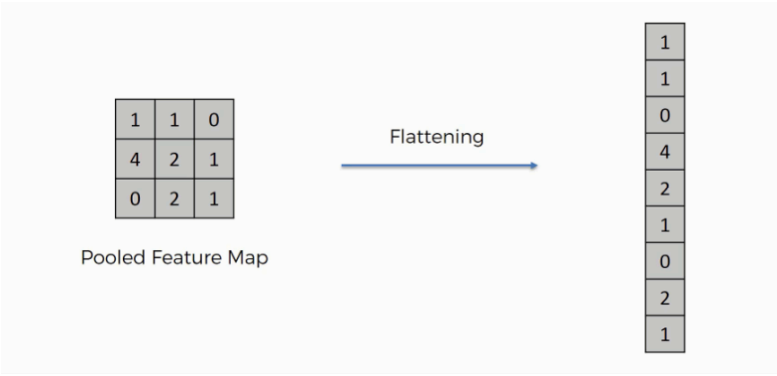


FIGURE 2.4 – Application du Flattening

L'intérêt de cette couche c'est que la couche fully connected (la couche suivante) prend simplement en entrée un vecteur, donc chaque neurone reçoit en entrée une valeur du vecteur.

2.3.1.5 La couche fully connected (Dense)

Il s'agit de la dernière couche d'un réseau de neurones. Notons que cette couche n'est donc pas caractéristique d'un CNN.

Elle permet de classifier l'image en renvoyant un vecteur de taille N, où N est le nombre de classes dans notre problème de classification d'images. Chaque élément du vecteur indique la probabilité pour l'image en entrée d'appartenir à une classe. Dans le cas d'une classification binaire, le vecteur final sera de taille 2.
La couche accorde à chaque valeur du tableau en entrée des poids qui dépendent de l'élément du tableau et de la classe.

Pour calculer les probabilités, la couche fully-connected multiplie donc chaque élément en entrée par un poids, fait la somme, puis applique soit la fonction d'activation Sigmoïde si N=2, ou la fonction d'activation Softmax si N>2.

Ce traitement revient à multiplier le vecteur en entrée par la matrice contenant les poids. Le fait que chaque valeur en entrée soit connectée avec toutes les valeurs en sortie explique le terme fully-connected.

La figure 2.5 montre un exemple de réseau composé de nombreuses couches à convolutions[8].

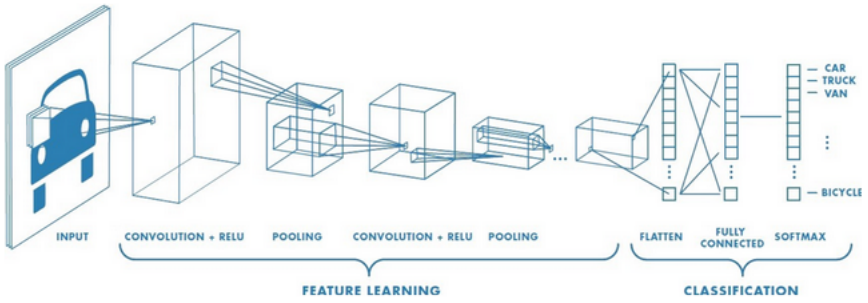


FIGURE 2.5 – Les différentes couches du CNN

2.4 Forward et Backward Propagation

Tout réseau de neurones comprend généralement deux phases importantes qui sont le forward et le backward propagation.

2.4.1 Forward Propagation

Le forward propagation : la phase où l'entrée est passée complètement à travers le réseau, le forward propagation fait référence au calcul et au stockage des variables intermédiaires, y compris les sorties pour le réseau neuronal dans l'ordre de la couche d'entrée à la couche de sortie.

Le but ici est de transmettre la propagation pour obtenir la sortie et la comparer à la valeur réelle pour obtenir l'erreur.

2.4.2 Backward Propagation

Le backward propagation : dans cette phase les gradients sont rétropropagiés, c'est-à-dire que les gradients d'erreur sont calculés grâce à une méthode pour chaque neurone, de la dernière couche vers la première. C'est dans cette phase aussi que se déroule la mise à jour des poids.

Toute phase du backward doit être précédée d'une phase forward correspondante, car lors du forward, chaque couche met en cache toutes les données dont elle aura besoin pour le backpropagation.
Le but de la backpropagation est de minimiser l'erreur, pour faire une propagation en arrière, il faut alors trouver la dérivée de l'erreur. Dans cette phase, chaque couche recevra un gradient de perte par rapport à ses sorties et renverra également un gradient de perte par rapport à ses entrées.

Chapitre 3

Outils utilisés

3.1 Langage de programmation

Dans le cadre de ce projet, notre encadrant nous a demandé d'utiliser le langage de programmation Python, un des langages les plus utilisés en sciences des données.

Python est un langage de programmation interprété, multi-paradigme et aussi multiplateformes. C'est un langage avec un typage dynamique fort, un système de gestion d'exceptions et une gestion automatique de la mémoire grâce au ramasse-miettes. Il est utilisé pour la programmation impérative structurée, fonctionnelle ainsi que orientée objet. [9]

La version avec laquelle nous avons réalisé ce projet est python3.

3.2 Bibliothèques utilisées

Dans notre projet nous avons utilisé les bibliothèques suivantes :

— NumPy : Cette bibliothèque est utilisée pour manipuler des matrices ou tableaux multidimensionnels, ainsi que des fonctions mathématiques opérant sur ces tableaux.

— Matplotlib : Cette bibliothèque est utilisée pour tracer et visualiser des données sous formes de graphiques. Sa version stable actuelle (la 2.0.1 en 2017) est compatible avec la version 3 de Python.

— TensorFlow : Cette bibliothèque permet de créer des modèles du deep learning. Elle simplifie le développement de logiciels machine learning essentiellement pour les applications mobiles et les services principaux.

— Keras : Cette bibliothèque permet d'interagir avec les algorithmes de réseaux de neurones profonds et de machine learning. Elle a été conçue pour permettre une expérimentation rapide avec les réseaux de neurones profonds.

3.3 Environnement

Jupyter est une application web utilisée pour programmer dans plus de 40 langages de programmation, dont Python. Elle permet de créer des notebooks, qui sont des documents interactifs contenant du texte en markdown, du code exécutable et des visualisations graphiques et textuelles. Son interface est similaire à celle des bloc-notes d'autres programmes tels que Maple, Mathematica et SageMath.[10]

3.4 Outils de partage et de communication

Afin d'assurer une meilleure gestion du projet (travail parallèle, gestion de versions, gestion de bugs, etc.) nous avons décidé d'utiliser l'outil GitHub, qui en plus d'offrir l'hébergement de projets avec Git, offre un logiciel de suivi de problèmes. GitHub propose aussi l'intégration d'un grand nombre de services externes, tels que l'intégration continue et la gestion de versions. Nous avons également utilisé Trello pour la gestion des tâches, cela nous a permis de mieux s'organiser et voir notre progression. De plus, nous avons eu recours à un service de stockage et de partage de fichiers Google Drive.
A cause de la mise en quarantaine dû à la propagation du virus COVID-19 pendant la réalisation de notre projet, les réunions en présentiel n'étaient donc plus possibles, ce qui nous a mené à organiser des réunions régulières à distance en utilisant l'outil Discord, que ce soit avec notre encadrant ou juste entre nous pour travailler ensemble.

Chapitre 4

Travail réalisé

Tout d'abord, nous nous sommes inspirés de la classe **MyNeuralNetwork**, qui est une classe dont nous avons le code source et qui permet de créer un réseau de neurones artificiels. Notre but est de créer une classe **ConvolutionalNeuralNetwork** plus ou moins similaire à celle que notre encadrant a mis à notre disposition afin de créer des réseaux de neurones convolutifs.

4.1 Compréhension

Dans un premier temps, nous avons principalement essayé de comprendre la descente du gradient, le fonctionnement du CNN (Forward et Backward) et la classe **MyNeuralNetwork** (Forward et Backward). Nous avons fait une présentation en présence de notre encadrant pour expliquer le déroulement du CNN, couche par couche. Cette présentation nous a permis de savoir que nous étions prêts à commencer l'implémentation.

4.2 Implémentation

Nous avons exploité plusieurs pistes au début. Nous avons commencé par essayer d'intégrer les couches CNN mis à part la couche softmax dans la classe **MyNeuralNetwork**, en passant en paramètres à cette dernière le résultat de la couche flatten du CNN pendant la phase du forward, mais nous nous sommes rendus compte que ce n'était pas possible puisque la classe CNN traite les données image par image alors que la classe **MyNeuralNetwork** de base prenait en entrée un jeu de données entier (plusieurs images).

Dans cette partie nous allons vous expliquer la solution alternative que nous avons implémenté.

4.2.1 Implémentation des différentes couches

4.2.1.1 La classe Conv3x3

La classe Conv3x3 est la classe qui implémente la couche de convolution expliquée précédemment.

Son constructeur prend qu'un seul argument : le nombre de filtres. Nous stockons ce dernier et initialisons un tableau de filtres aléatoire.

Nous avons créé une couche convolution de taille 3 x 3, il est également possible de créer une couche convolution de taille différente.

1. **Forward :**
   La couche convolution peut se situer au début comme elle peut aussi être utiliser au milieu de notre réseau (dans le cas où nous appliquons plusieurs convolutions). Dans ce dernier cas le forward de cette couche prend en entrée la sortie du forward de la couche qui la précède, sinon il prendra l'image initiale.
   La méthode iterate_regions_forward que nous avons implémentée nous permet de générer toutes les parties de l'image originale ayant la même taille que le filtre afin d'appliquer dessus le filtre, chacun des filtres de cette couche produira une image d'une taille réduite pour avoir au final un ensemble d'images.

```
def forward_propagation(input):
    if(input is 2D):
        foreach region_image:
            output = sum(region_image * filtres)
    else:
        foreach input:
            foreach region_image:
                output = sum(region_image * filters)
    return output

```

2. **Backward :**
   Comme expliqué dans le Forward la couche convolution peut se situer au milieu de notre réseau et pas qu'au début.
   Si la convolution est la première couche de notre réseau, le backward n'a rien à retourner, nous allons juste modifier le filtre par rapport au learning rate.
   Dans le cas où cette couche n'est pas au début de notre réseau, son backward ne fait rien de compliquer à part renvoyer la perte du gradient pour les entrées de cette couche.

```
def backward_propagation(d_L_d_out, learn_rate):
    if(input is 2D):
        foreach region_image:
            foreach filters:
                d_L_d_filters += d_L_d_out * region_image
        filters -= learn_rate * d_L_d_filters
        return None
    else:
        foreach region_image:
            foreach image_filter:
                d_L_d_filters += d_L_d_out * region_image
                d_L_d_input = d_L_d_out*learn_rate
        return d_L_d_input
```

4.2.1.2 La classe ReLU

La classe Relu est la classe qui implémente la couche de Relu expliquée précédemment.

1. **Forward :**
   Le forward de la classe relu parcourt la matrice (i, j) de l'image, si la valeur $C_{i,j}$ est supérieure à 0, nous gardons la même valeur sinon nous remplaçons cette valeur négative par 0.

```
def forward_propagation(input):
    output = maximum(0, input)
    return output
```

2. **Backward :**
   Le backward de la couche ReLU remplace les valeurs inférieures ou égales à 0 par un zéro et remplace les autres valeurs par un un.

```
27 def backward_propagation(d_out):
28     dOut[dOut <= 0] = 0
29     dOut[dOut > 0] = 1
```

4.2.1.3 La classe MaxPool

La classe MaxPool est la classe qui implémente la couche de MaxPool expliquée précédemment.

1. **Forward :** Le fonctionnement du forward de cette classe est similaire à celui de la classe conv3x3, sauf qu'au lieu du produit matriciel, nous récupérons le maximum de chaque partie d'image générée par la méthode iterate_regions _forward sur l'ensemble des images de la sortie de la couche qui la précède.

```
30 def forward_propagation(input):
31     foreach region_image:
32         output = amax(region_image)
33     return output
```

2. **Backward :**
   Le backward de la couche MaxPool ne fait rien de plus à part passer le résultat de sa couche précédente à sa couche suivante en doublant la largeur et la hauteur de la perte du gradient, en affectant chaque valeur de gradient à l'endroit où se trouvait la valeur maximale d'origine dans son bloc 2 x 2 correspondant, et en remplaçant les autres valeurs par des zéros.

```
34 def backward_propagation(d_L_d_out):
35     foreach region_image:
36         foreach image_filter:
37             if region_image[index] == amax[image_filter] :
38                 d_L_d_input = d_L_d_out
39     return d_L_d_input
```

4.2.1.4 La classe MyFlatten

La classe Myflatten est la classe qui implémente la couche de Flatten expliquée précédemment.

1. **Forward :**
   Le forward de cette classe nous permet de transformer les images représentées par des matrices en vecteur pour pouvoir le passer au forward de la couche Dense.

```
40 def forward_propagation(input):
41     last_input_shape = input.shape
42     input = inpt.flatten()
43     return input
```

2. **Backward :**
   Le backward de cette classe nous permet de transformer le vecteur reçu du backprop de la couche Dense en matrice.

```
44 def backward_propagation(d_L_d_out):
45     return d_L_d_out.reshape(last_input_shape)
```

4.2.1.5 La classe Dense

La classe Dense est la classe qui implémente la couche de Dense expliquée précédemment. Elle peut prendre différentes fonctions d'activation.

1. **Forward :**
   Nous faisons appel à la fonction forward de la fonction d'activation correspondante passée en paramètre pendant la création de notre réseau.

```
46 def forward_propagation(input):
47     if(activation_func == 'softmax'):
48         output = softmax_forward(input)
49     return output
```

2. **Backward :**
   Nous faisons appel à la fonction backward de la fonction d'activation correspondante passée en paramètre pendant la création de notre réseau.

```
50 def backward_propagation(d_L_d_out):
51     if(activation_func == 'softmax'):
52         output = softmax_backprop(d_L_d_out)
53     return output
```

4.2.2 Implémentation de la classe Dropout

C'est possible que les neurones de la couche fully connected développent une co-dépendance entre eux au cours de leur formation, ce qui limite la puissance individuelle de chaque neurone et qui provoque un sur-apprentissag des données de formation.

Nous parlons d'un sur-apprentissage lorsque le jeu d'entraînement s'adapte un peu trop à la fonction de prédiction. Ce qui empêche notre réseau de prédire efficacement les nouvelles données.

Pour réduire le risque de sur-apprentissage et améliorer les performances du temps de test, nous avons ajouté une couche permettant une régularisation très efficace : il s'agit de la couche de Dropout.

Le principe général de cette couche est de désactiver aléatoirement à chaque passage une partie des neurones du réseau pour diminuer artificiellement le nombre de paramètres.

1. **Forward :**
   Tout d'abord, nous échantillonnons un tableau de distribution de Bernoulli indépendant, qui joue le rôle d'un masque de zéro (pour désactiver le neurone) ou de un (pour laisser le neurone actif) selon une certaine probabilité p.
   Ensuite, nous multiplions la couche précédente par le masque construit.

```
54 def forward(...):
55     mask = np.random.binomial(1,probabilité,size=input.shape)
56                                     / probabilité
57             ...
58     out = input * mask
59
60     return out
```

2. **Backward :**
   Pendant le backward propagation, les neurones inactifs ne seront pas considérés, donc nous n'appliquerons pas de descente de gradient dessus.
   Nous n'appliquons la descende de gradient que sur les neurones provenant de la couche qui suit.

4.2.3 Implémentation de la classe ConvolutianalNeuralNetwork

Cette classe nous permet de créer un réseau de neurone à convolutions. Elle a les méthodes suivantes :

1. **info :**
   Cette méthode permet d'afficher quelques informations sur le réseau créé.
2. **addLayer :**
   Cette méthode permet d'ajouter une couche a notre réseau.
3. **forward_propagation :**
   Cette méthode fait appel à tous les forwards de toutes les couches de notre réseau dans l'ordre.
4. **cost_function :**
   Cette méthode calcule le coût de la descente du gradient.
5. **backward_propagation :**
   Cette méthode fait appel à tous les backwards de toutes les couches de notre réseau dans l'ordre inverse.
6. **accuracy :**
   Cette méthode renvoie 1 si la valeur prédite correspond à sa vrai valeur sinon elle renvoie 0.
7. **predict :**
   Cette méthode calcule la prédiction de chaque image.
8. **fit :**
   C'est la méthode coeur de notre classe, elle permet d'entraîner notre modèle en lançant les différentes phases en fonction du nombre d'epochs. Elle retourne l'historique du coût et de l'accuracy et les affiche.

4.2.4 Construction du réseau

Pour la construction de notre réseau, nous avons créé une instance de la classe **ConvolutionalNeuralNetwork** qui nous a permis d'utiliser les méthodes ci-dessus.

D'abord, nous avons utilisé la méthode **addLayer()** qui prend en paramètre le constructeur de la classe d'une des couches. Voici dans l'ordre l'enchaînement des couches qui constituent notre réseau pour les données de MNIST :

1. La couche conv3x3 avec 8 filtres
2. La couche Relu
3. La couche MaxPool2
4. La couche Dropout
5. La couche MyFlatten
6. La couche Dense qui prend la taille du résultat du Flatten et une fonction d'activation, dans notre réseau c'est la fonction softmax.

Une fois cet enchaînement établi nous avons fait appel avec la même instance de la classe **ConvolutionalNeuralNetwork** à la méthode fit qui va entraîner notre modèle en appliquant le forward et le backprop de chaque couche.

```
61 print(f'Creating the network...')
62 network = MyCNN()
63 network.addLayer(Conv3x3(8)) # 28x28 -> 26x26x8
64 network.addLayer(Relu())  # 26x26x8 -> 26x26x8
65 network.addLayer(MaxPool2()) # 26x26x8 -> 13x13x8
```

```
66 network.addLayer(Dropout(0.1))
67 network.addLayer(MyFlatten()) # 13x13x8 -> 1352
68 network.addLayer(Dense(1352, 10, activation = "softmax"))
```

4.2.5 Data generator

Data generator permet d'avoir encore plus de données en plus de notre jeu de données, afin d'apprendre à notre modèle à reconnaître les mêmes images avec de légers changements (zoom, rotation, déplacement de l'objet en question, etc).[11]

Chapitre 5

Test et comparaison

5.1 Outils d'expérimentation

5.1.1 Les jeux de données

Pour réaliser notre projet, nous avons utilisé plusieurs jeux de données.

MNIST

MNIST est un jeu de données qui contient des images de chiffres manuscrits en noir et blanc. Il contient 60.000 images d'entraînement et 10.000 images de test. Chaque image a une résolution de 28 x 28 pixels, représentant un chiffre.[12]

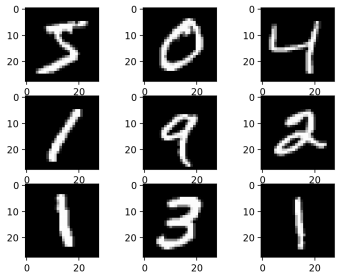


FIGURE 5.1 – Exemple du jeu de données MNIST

Fashion-MNIST

C'est un jeu de données où les images représentent des vêtements de résolution 28 x 28 pixels. Il regroupe 70 000 images réparties en 10 catégories, dont 60.000 images d'entraînement et 10.000 images de test.[13]

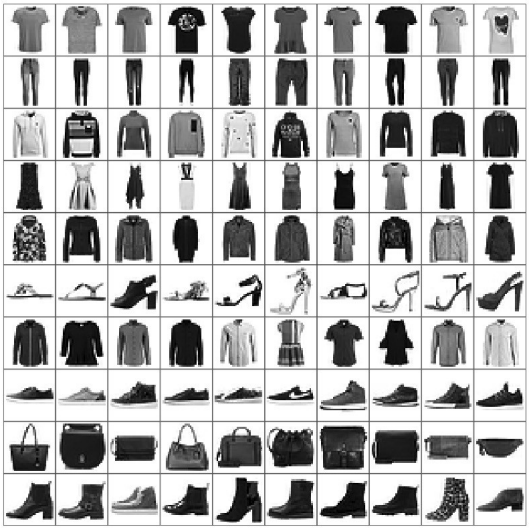


FIGURE 5.2 – Exemple du jeu de données Fashion-MNIST

Cifar-10

Le jeu de données CIFAR-10 contient 60.000 images en couleur de 32 x 32 regroupées en 10 classes, avec 6.000 images par classe. Il y a 50.000 images d'entraînement et 10.000 images de test.[14]

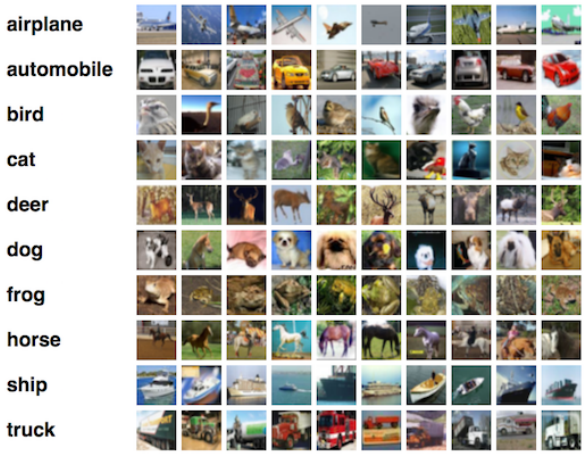


FIGURE 5.3 – Exemple du jeu de données CIFAR-10

5.1.2 Outils matériels

Les expérimentations ont été menées sur une machine sous Windows 10 de 8GB RAM et de processeur intel CORE i7 8th Gen.

5.2 Analyse des résultats

Au cours de ces expérimentations, nous avons testé deux modèles différents sur les trois jeux de données et nous avons fini par comparer les résultats obtenus par nos modèles avec ceux obtenus par les modèles conçus par la bibliothèque Keras.

Pour ce fait, nous avons fixé la taille des jeux de données à 2000 images au total, 1200 images pour l'entraînement et 800 images pour le test, avec 6 epochs et eta = 0.01. Concernant le Dropout, nous avons fixé la probabilité à 0.5. Puis nous avons calculé la moyenne pour avoir le résultat final.

5.2.1 Comparaison des résultats de notre modèle

La figure 5.4 représente le résultat obtenu lorsque nous lançons le programme. Le modèle commence par apprendre sur le jeu dapprentissage en passant par différents epochs, en faisant un forward qui parcourt, toutes les couches de notre modèle (de la première à la dernière). Lorsqu'il arrive à la dernière couche (la couche Dense), cette dernière donnera une probabilité pour chaque classe, pour que le backward puisse à son tour modifier les valeurs dans le sens inverse. Plus nous avançons dans les epochs plus la précision augmente, contrairement à la perte qui baisse. Cest grâce à la descente du gradient qui actualise les paramètres en faisant une marche arrière sur nos couches.

```
Learning...
        Running epoch 1...
                (Past 100 steps) Step 100 : Average Loss : 2.27 | Accuracy : 16% | Time : 2.19
                (Past 100 steps) Step 200 : Average Loss : 2.21 | Accuracy : 32% | Time : 1.93
                (Past 100 steps) Step 300 : Average Loss : 2.13 | Accuracy : 46% | Time : 2.00
                (Past 100 steps) Step 400 : Average Loss : 2.04 | Accuracy : 60% | Time : 2.03
                (Past 100 steps) Step 500 : Average Loss : 2.00 | Accuracy : 52% | Time : 1.99
                (Past 100 steps) Step 600 : Average Loss : 2.00 | Accuracy : 46% | Time : 1.92
                (Past 100 steps) Step 700 : Average Loss : 1.96 | Accuracy : 50% | Time : 1.87
                (Past 100 steps) Step 800 : Average Loss : 1.86 | Accuracy : 63% | Time : 1.94
                (Past 100 steps) Step 900 : Average Loss : 1.81 | Accuracy : 69% | Time : 2.01
                (Past 100 steps) Step 1000 : Average Loss : 1.76 | Accuracy : 68% | Time : 1.98
                (Past 100 steps) Step 1100 : Average Loss : 1.77 | Accuracy : 56% | Time : 1.96
                (Past 100 steps) Step 1200 : Average Loss : 1.74 | Accuracy : 59% | Time : 1.87
        Running epoch 2...
                (Past 100 steps) Step 100 : Average Loss : 1.55 | Accuracy : 74% | Time : 2.04
                (Past 100 steps) Step 200 : Average Loss : 1.59 | Accuracy : 68% | Time : 1.97
                (Past 100 steps) Step 300 : Average Loss : 1.50 | Accuracy : 75% | Time : 1.90
```

FIGURE 5.4 – Résultat de l'exécution

Analyse en fonction du Dropout

La table 5.1 représente les résultats obtenus après avoir tester un modèle qui n'applique pas la classe Dropout sur les trois jeux de données MNIST, Fashin-MNIST et CIFAR-10. La table 5.2 représente les résultats obtenus après avoir tester un modèle qui applique la classe Dropout.
En observant les valeurs obtenues, nous pouvons constater qu'un modèle avec une classe Dropout est moins performant comparé à un modèle sans Dropout.

Dataset	MNIST	Fashion-MNIST	CIFAR-10
Accuracy (%)	77	76	10
Train time (s)	115.55	112.06	309.38
Test time (s)	6.83	7.93	29.59

TABLE 5.1 – Résultat sans le Dropout

Dataset	MNIST	Fashion-MNIST	CIFAR-10
Accuracy (%)	70	72	9
Train time (s)	123.83	121.40	336.41
Test time (s)	8.02	8.21	31.34

TABLE 5.2 – Résultat avec le Dropout

La figure 5.5 représente la courbe obtenue après avoir tester le modèle sans Dropout sur le jeu de données Mnist et la figure 5.6 représente celle du modèle avec Dropout. Cette courbe étudie l'évolution de la perte et de la précision en fonction du temps. Nous pouvons remarquer que la perte du modèle avec Dropout est plus importante que la perte du modèle sans Dropout, et que la précision sans Dropout est légèrement meilleure.

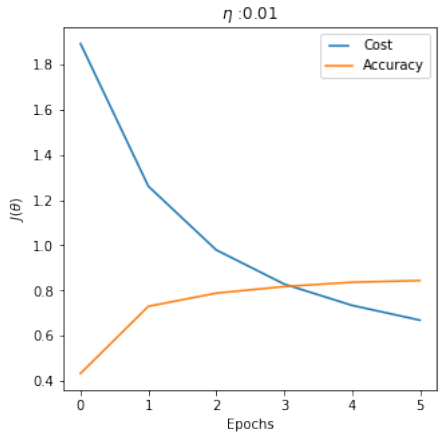


FIGURE 5.5 – MNIST sans Dropout

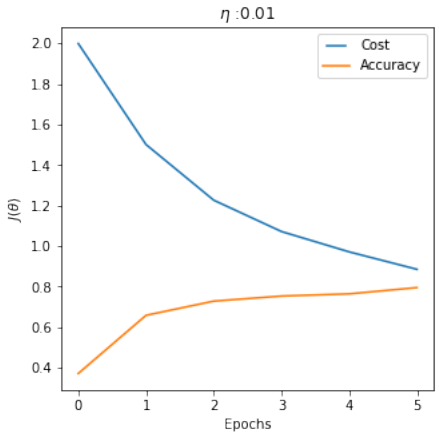


FIGURE 5.6 – MNIST avec Dropout

Nous pouvons clairement voir que notre modèle a une précision beaucoup plus élevée sur les données de MNIST et Fashion-MNIST que sur les données CIFAR-10, ceci est dû à la complexité des images de ce dernier, en plus d'être en couleurs. D'après nos observations nous pouvons constater que sans doute, en appliquant le Dropout nous avons des résultats moins performants, mais nous évitons le risque du sur-apprentissage.

5.2.2 Comparaison des résultats de Keras

Nous avons effectué les mêmes tests que précédemment, mais cette fois-ci avec la bibliothèque Keras.

Analyse en fonction du Dropout

La table 5.3 représente les résultats obtenus après avoir appliquer un modèle Keras sans le Dropout sur nos trois jeux de données.
La table 5.4 représente les résultats obtenus après avoir tester un modèle Keras qui utilise la couche Dropout.
En observant les valeurs obtenues, nous pouvons constater qu'un modèle Keras avec une classe Dropout est moins performant qu'un modèle sans le Dropout.

Dataset	MNIST	Fashion-MNIST	CIFAR-10
Accuracy (%)	86	77	33
Train time (s)	2.82	2.48	2.91
Test time (s)	0.08	0.10	0.09

TABLE 5.3 – Résultat sans Dropout

Dataset	MNIST	Fashion-MNIST	CIFAR-10
Accuracy (%)	83	77	31
Train time (s)	2.66	2.94	3.26
Test time (s)	0.07	0.08	0.11

TABLE 5.4 – Résultat avec dropout

Nous remarquons que même pour Keras, les jeux de données MNIST et Fashion-MNIST ont des meilleures précisions que CIFAR-10, et qu'un modèle avec Dropout est moins performant.

5.2.3 Comparaison de nos résultats avec ceux de Keras

Nous remarquons que les résultats obtenus par Keras sont plus performants. Néanmoins, avec notre modèle nous obtenons des résultats raisonnables proches des résultats de Keras, par exemple nous avons 76% de précision sur le jeu de données Fashion-MNIST contre 77% de précision avec Keras. En revanche, nous remarquons que Keras est plus rapide lors de l'exécution, cela est expliqué par le fait que ce dernier est bien optimisé.

Chapitre 6

Conduite de projet

Dans ce chapitre nous allons parler de la gestion de notre projet, en appliquant les méthodes vues en cours de l'UE HMIN204 intitulée conduite de projet.

6.1 Présentation du projet

Notre projet est un travail d'étude et de recherche intitulé "Comment les réseaux de neurones convolutifs obtiennent de si bons résultats en classification ?", ce projet porte sur le domaine du deep learning. Le TER a commencé le jeudi 23 janvier 2020 et il finit le mercredi 20 mai 2020.

Nous sommes quatre étudiants du même cursus en première année du Master DECOL au sein de la faculté des sciences, et le nom de notre groupe est "GoFocus".

Nous avions plus ou moins les mêmes connaissances dans le domaine du deep learning, et ces dernières n'étaient pas très avancées. Mais nos diverses expériences personnelles au cours de ces années d'études et notre motivation, nous ont permis d'aiguiser notre curiosité et de nous ouvrir à ce domaine et de vouloir en apprendre plus sur les réseaux de neurones.

6.2 Découpage du projet

Nous avons découpé notre projet en différentes parties, en se basant sur le principe du découpage temporel vu que les tâches réalisées sont dépendantes. Nous avons plusieurs cycles de développement composés de plusieurs phases :

1. La première phase correspond à l'étude de la problématique de la tâche à réaliser.
2. La deuxième phase correspond à l'élaboration d'un plan d'action, en discutant la stratégie à mettre en place.
3. La troisième phase correspond à la réalisation (implémentation) de la tâche.
4. La quatrième phase qui est la phase de test.
5. La dernière phase correspond à l'intégration du travail réalisé dans l'ensemble du projet.

6.3 L'estimation des risques

Le tableau 6.1 représente quelques risques que nous estimions possibles

Nature du risque	Degré du risque pour le projet (0 à 5)					
	0	1	2	3	4	5
Taille du projet/Temps				•		
Panne technique		•				
Degré d'intégration			•			
Instabilité de l'équipe de projet		•				
Épidémie				•		
Confinement			•			

TABLE 6.1 – Tableau d'estimation des risques

En fonction de ces risques, nous pensons que le meilleur modèle à adopter est le modèle en V.

6.4 Les techniques de planification

Tout d'abord, nous avons commencé par l'élaboration d'une liste de tâches avec la durée estimée de chacune d'elle. Nous pouvons voir cette décomposition hiérarchique des travaux nécessaires pour réaliser les objectifs de notre projet grâce au diagramme WBS (Figure Annexe A).

Dans un second temps, nous avons établi un calendrier de travail. Pour cela, nous avons utilisé le diagramme de Gantt (Figure Annexe B) qui illustre la répartition du temps attribué à chacune des grandes étapes du projet. C'est une conclusion graphique résultante de notre gestion de projet et qui nous permet d'identifier au premier coup d'oeil les travaux chronophages et l'homogénéité entre les tâches.

Associé au diagramme GANTT, le diagramme PERT (Figure Annexe C) permet de mettre en évidence les liaisons entre les différentes tâches afin den déduire un chemin dit ń critique ż.

6.5 Organisation du travail

Étant donné que les membres de notre groupe sont polyvalents, nous n'avons pas affecté des tâches précises pour chacun d'entre nous. Nous avons préféré réaliser des tâches diverses pour développer le plus de connaissances possible.

Nous avons notamment préféré de travailler ensemble sur la plus part des parties, et nous nous sommes affectés au fil du projet des micro-tâches fréquemment mises à jour, de sorte que la charge soit équitable. Dans le cas où nous ne travaillons pas tous ensemble, nous répartissons le travail en deux et nous travaillons en binôme.

6.6 Métriques quantitatives

Nous avons choisi comme outil de gestion de projet, le gestionnaire GitHub afin d'assurer une meilleure gestion et un bon suivi de problèmes qui peuvent survenir. Nous avons envoyé des commits pour chaque tâche fonctionnelle. Nous sommes ainsi arrivés en fin de projet avec un cumul de 72 commits, globalement bien répartis entre nous quatre. (Avec environ 3500 modifications).
Nous avons implémenté trois fichiers principaux et sept classes, ce qui nous fait un

total de 1169 lignes de code.
Durant les deux premiers mois, nous travaillions en moyenne deux jours par semaine, ce qui fait 16h de travail par semaine par chaque membre. Ensuite, nous avons accéléré un peu le rythme pour arriver à une moyenne de 36h par semaine durant le mois de mai.

6.7 Métriques qualitatives

Pendant la réalisation de ce projet, nous avons été confrontés à une situation de confinement suite à la propagation du virus du COVID-19, ce qui a impacté l'avancement de celui-ci. De plus, à cause de cette situation, la date du rendu du projet a été avancée au 20 mai 2020, par conséquent nous avons dû apporter quelques changements à notre diagramme de gantt initial.

La figure dans l'annexe D illustre le nouveau Gantt, nous pouvons voir la différence par rapport au Gantt initial qui a été sauvegardé avant la modification du diagramme.

Chapitre 7

Conclusion

7.1 Résumé de la contribution

Dans ce projet, nous avons présenté le fonctionnement des réseaux de neurones. Nous avons aussi expliqué la réalisation d'un réseau de neurones convolutifs qui peut s'étendre pour ajouter des nouvelles couches qui permettent d'avoir de meilleurs résultats de prédiction, nous arrivons bien à éviter le sur-apprentissage, par exemple grâce à l'ajout de la couche de dropout. Nous avons également comparé le résultat obtenu avec Keras. Nous avons remarqué que même si l'approche est beaucoup moins rapide, elle a confirmé que le travail réalisé était juste.

Nous avons consacré beaucoup de temps à la compréhension des réseaux de neurones, notamment le fonctionnement du backward et la descente du gradient, que nous pensions commode en début du projet. De plus, trouver la meilleure approche pour traiter ce sujet ne semblait pas évidente au début, pour cela nous avons exploré plusieurs stratégies jusqu'à en trouver celle qui répondait à nos contraintes. Après avoir bien cerné le sujet, nous ne regrettons pas de lavoir choisi. Le deep learning en général et les CNNs en particulier sont de plus en plus utilisés pour le traitement des données, et grâce à cette riche expérience, nous avons désormais, de très bonnes bases dans ce domaine.

Nous avons mis à disposition un dépôt GitHub contenant tous les fichiers de notre projet ainsi qu'une vidéo de démonstration . Consultable via ce lien :
https://github.com/nilbahaaeddine/M1_CNN.git

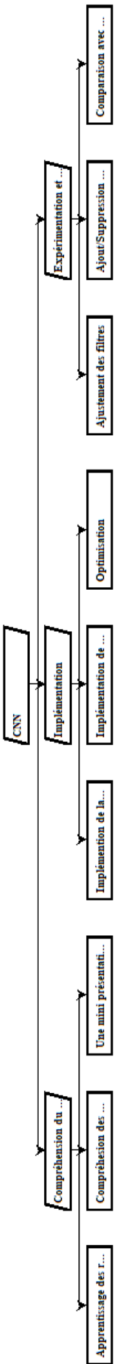
7.2 Perspectives

Le travail que nous avons réalisé pourra être repris par la suite pour l'améliorer en optimisation et pour ajouter de nouvelles fonctionnalités, comme l'ajout d'autres fonctions d'activation hormis la fonction softmax, tel que la fonction sigmoïde, nous pourrions aussi tester notre modèle sur d'autres jeux de données sur des machines qui supportent un gros volume de données.

Notre programme peut être étendu en ajoutant différentes couches pour l'obtention de résultats plus précis.

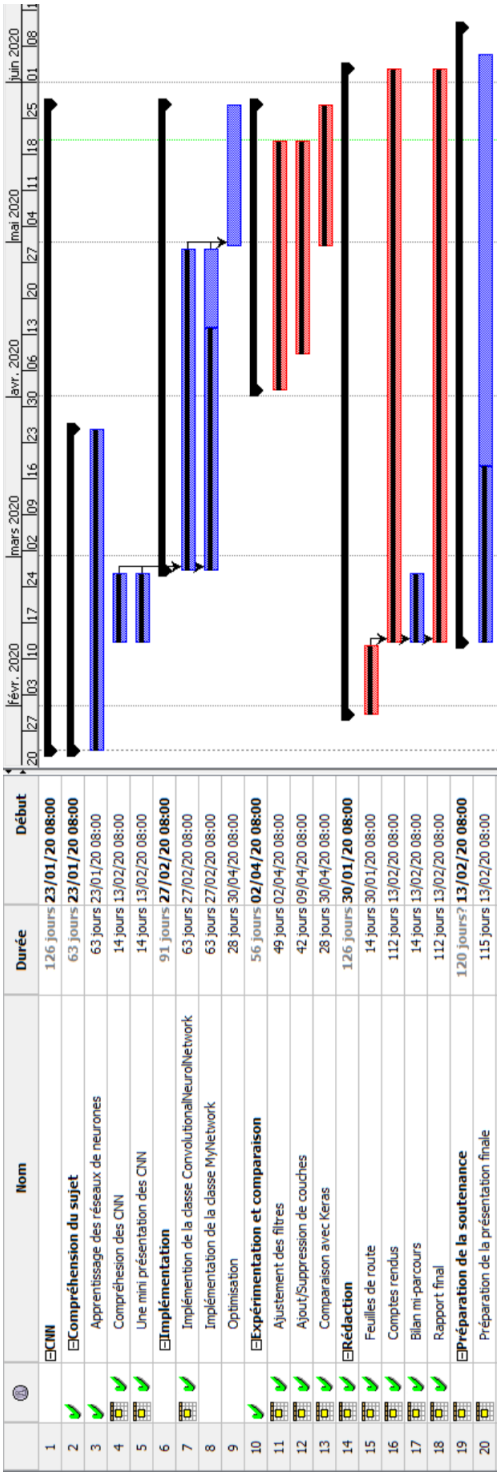
Annexe A

WBS du projet



Annexe B

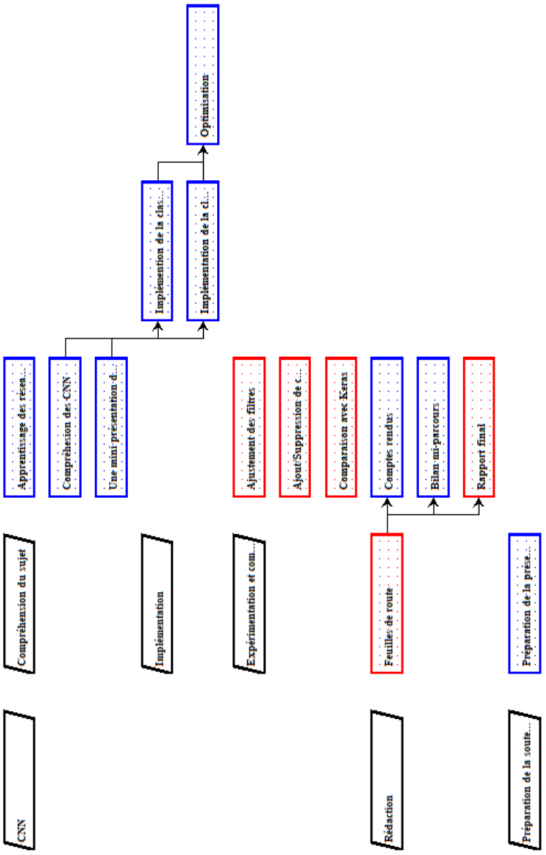
Diagramme de Gantt initial



	Nom	Durée	Début
1	CNN	126 jours	23/01/20 08:00
2	Compréhension du sujet	63 jours	23/01/20 08:00
3	Apprentissage des réseaux de neurones	63 jours	23/01/20 08:00
4	Compréhesion des CNN	14 jours	13/02/20 08:00
5	Une mini présentation des CNN	14 jours	13/02/20 08:00
6	Implémentation	91 jours	27/02/20 08:00
7	Implémention de la classe ConvolutionalNeuralNetwork	63 jours	27/02/20 08:00
8	Implémentation de la classe MyNetwork	63 jours	27/02/20 08:00
9	Optimisation	28 jours	30/04/20 08:00
10	Expérimentation et comparaison	56 jours	02/04/20 08:00
11	Ajustement des filtres	49 jours	02/04/20 08:00
12	Ajout/Suppression de couches	42 jours	09/04/20 08:00
13	Comparaison avec Keras	28 jours	30/04/20 08:00
14	Rédaction	126 jours	30/01/20 08:00
15	Feuilles de route	14 jours	30/01/20 08:00
16	Comptes rendus	112 jours	13/02/20 08:00
17	Bilan mi-parcours	14 jours	13/02/20 08:00
18	Rapport final	112 jours	13/02/20 08:00
19	Préparation de la soutenance	120 jours?	13/02/20 08:00
20	Préparation de la présentation finale	115 jours	13/02/20 08:00

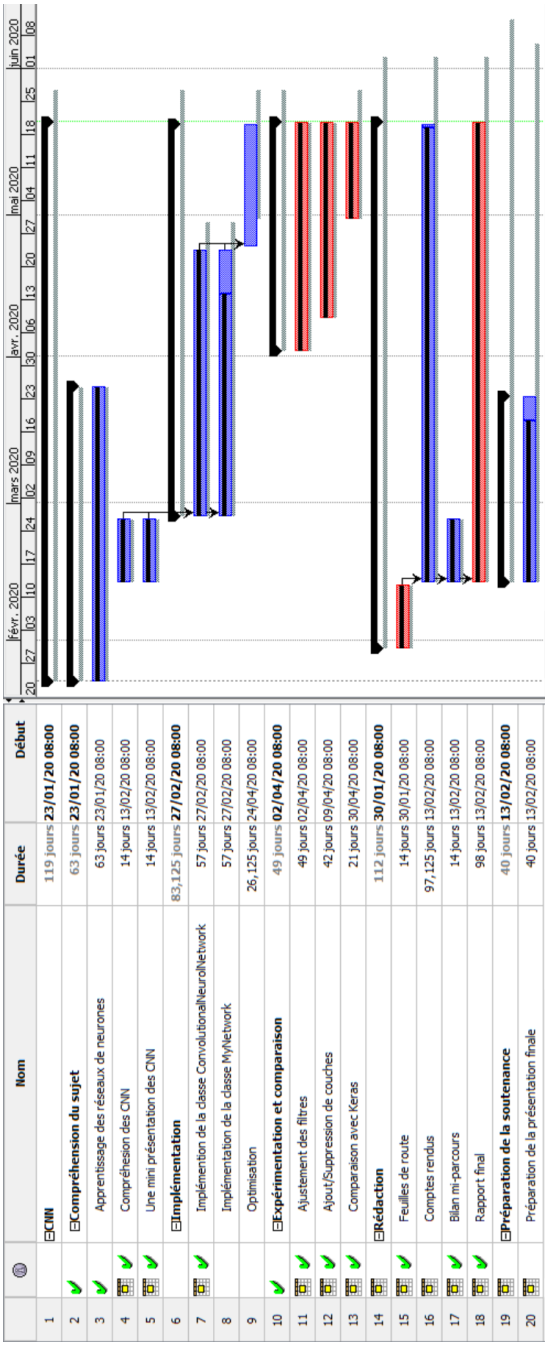
Annexe C

Diagramme PERT final



Annexe D

Diagramme de Gantt final



		Nom	Durée	Début
1		**CNN**	119 jours	**23/01/20 08:00**
2	✓	**Compréhension du sujet**	63 jours	**23/01/20 08:00**
3	✓	Apprentissage des réseaux de neurones	63 jours	23/01/20 08:00
4	✓	Compréhesion des CNN	14 jours	13/02/20 08:00
5	✓	Une mini présentation des CNN	14 jours	13/02/20 08:00
6		**Implémentation**	83,125 jours	**27/02/20 08:00**
7	✓	Implémention de la classe ConvolutionalNeuralNetwork	57 jours	27/02/20 08:00
8		Implémentation de la classe MyNetwork	57 jours	27/02/20 08:00
9		Optimisation	26,125 jours	24/04/20 08:00
10	✓	**Expérimentation et comparaison**	49 jours	**02/04/20 08:00**
11	✓	Ajustement des filtres	49 jours	02/04/20 08:00
12	✓	Ajout/Suppression de couches	42 jours	09/04/20 08:00
13	✓	Comparaison avec Keras	21 jours	30/04/20 08:00
14		**Rédaction**	112 jours	**30/01/20 08:00**
15	✓	Feuilles de route	14 jours	30/01/20 08:00
16		Comptes rendus	97,125 jours	13/02/20 08:00
17	✓	Bilan mi-parcours	14 jours	13/02/20 08:00
18	✓	Rapport final	98 jours	13/02/20 08:00
19		**Préparation de la soutenance**	40 jours	**13/02/20 08:00**
20		Préparation de la présentation finale	40 jours	13/02/20 08:00

Bibliographie

[1] Siddharth Das. CNN Architectures : LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. Library Catalog : medium.com, consulté le 20/05/2020.

[2] Zahangir Alom, Tarek M Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, and Mst Shamima Nasrin. The History Began from AlexNet : A Comprehensive Survey on Deep Learning Approaches. page 39.

[3] Convolutional neural network. Page Version ID : 957262810, consulté le 20/05/2020.

[4] vibhor nigam. Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning. Library Catalog : towardsdatascience.com, consulté le 20/05/2020.

[5] Amar Budhiraja. Learning Less to Learn Better Dropout in (Deep) Machine learning. Library Catalog : medium.com, consulté le 20/05/2020.

[6] Neural Networks for Image Recognition : Methods, Best Practices, Applications. Library Catalog : missinglink.ai, consulté le 20/05/2020.

[7] Lambert R. Focus : Le Réseau de Neurones Convolutifs. Library Catalog : penseeartificielle.fr Section : Focus, consulté le 20/05/2020.

[8] A Comprehensive Guide to Convolutional Neural Networks the ELI5 way. Library Catalog : penseeartificielle.fr Section : Focus, consulté le 29/05/2020.

[9] Python (langage). Page Version ID : 170859166, consulté le 20/05/2020.

[10] Jupyter. Page Version ID : 170409027, consulté le 20/05/2020.

[11] Image classification with CNNs and small augmented datasets. Library Catalog : www.novatec-gmbh.de, consulté le 20/05/2020.

[12] Base de données MNIST. Page Version ID : 168498873, consulté le 20/05/2020.

[13] Fashion MNIST. Page Version ID : 160653649, consulté le 20/05/2020.

[14] CIFAR-10. Page Version ID : 931952784, consulté le 20/05/2020.