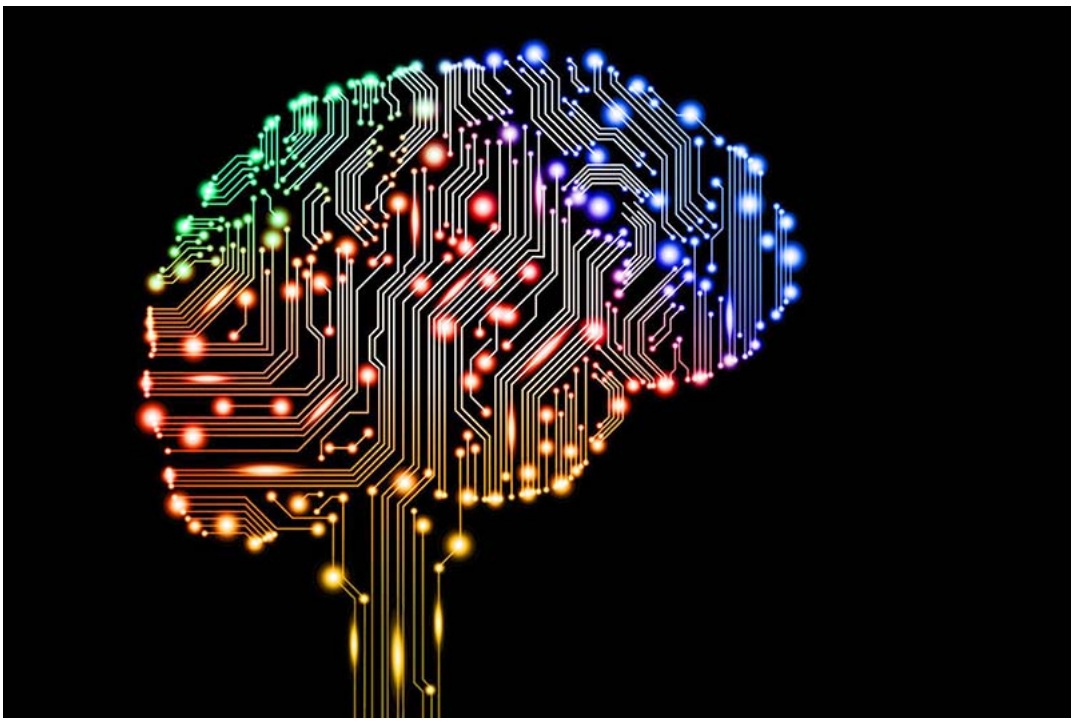
Bordeaux INP
ENSC

PROJET INFORMATIQUE INDIVIDUEL

Réseaux neuronaux en Python

Émilie ROGER – 27 avril 2017



PLAN DU RAPPORT

1. **Introduction** . . . 3
   - 1.1. Contexte . . . 3
   - 1.2. Objectifs . . . 3
2. **Gestion de projet** . . . 3
   - 2.1. Organisation . . . 3
     - 2.1.1. Fréquence de travail . . . 3
     - 2.1.2. Temps de travail . . . 4
   - 2.2. Communication . . . 4
   - 2.3. Planification . . . 4
3. **Présentation du projet** . . . 5
   - 3.1. Installation . . . 5
   - 3.2. Créer une couche . . . 6
     - 3.2.1. Couche d'entrée . . . 6
     - 3.2.2. Couche entièrement connectée . . . 6
     - 3.2.3. Couche de convolution . . . 7
     - 3.2.4. Couche de pooling . . . 7
   - 3.3. Créer un réseau de neurones . . . 8
   - 3.4. Entraîner et tester un réseau de neurones . . . 8
     - 3.4.1. Entraîner un réseau de neurones . . . 8
     - 3.4.2. Tester un réseau de neurones . . . 9
   - 3.5. Créer ou récupérer des données . . . 9
4. **Réalisation du projet** . . . 10
   - 4.1. Perceptron multi-couches . . . 10
   - 4.2. Réseau neuronal convolutif . . . 11
   - 4.3. Bibliothèque . . . 12
5. **Bilan du projet** . . . 12
   - 5.1. Points positifs & Apport pédagogique . . . 12
   - 5.2. Points négatifs & Difficultés . . . 13
6. **Conclusion** . . . 13
7. **Quelques informations pour comprendre les réseaux de neurones** . . . 14

1. Introduction

1.1. Contexte

Dans le cadre de ma formation à l'ENSC et afin de me préparer pour mon stage de 2^e année, j'ai effectué un projet informatique sur les réseaux neuronaux, notamment le perceptron multi-couches et le réseau neuronal convolutif, en Python.

En effet, lors de mon stage, je serai amener à utiliser le langage Python et à faire de l'intelligence artificielle avec la bibliothèque logicielle TensorFlow. Or celle-ci s'utilise avec Python et s'appuie en interne sur les réseaux neuronaux pour fonctionner.

Ce projet me permet donc d'une part de me familiariser avec Python, mais d'autre part de comprendre ce qui se passe lorsqu'on utilise TensorFlow.

1.2. Objectifs

Les objectifs de ce projet s'inscrivent dans 2 catégories : les objectifs en terme d'apport pédagogique et les objectifs en terme de résultat algorithmique.

Au niveau de l'apport pédagogique, comme précisé dans la partie précédente, l'objectif principal de ce projet est de me préparer pour mon stage de fin de 2^e année. Par ailleurs, ce projet a pour but de renforcer et d'approfondir mes connaissances en intelligence artificielle, un domaine dans lequel je me verrais bien travailler plus tard.

Au niveau du résultat algorithmique, ce projet a pour objectif de programmer en Python une bibliothèque permettant de créer des réseaux de neurones selon les désirs de l'utilisateur : soit un perceptron multi-couches, soit un réseau neuronal convolutif. Ainsi, le but est que l'utilisateur puisse récupérer un réseau de neurones fonctionnel, c'est-à-dire qu'on puisse entraîner et tester avec des données numériques, rien qu'en précisant quelles sont les couches il souhaite avoir dans son réseau.

Ce projet n'a pas pour vocation de répondre à une demande ou un besoin de la part d'un quelconque public, mais est davantage un travail réalisé dans un cadre personnel. Cependant, le projet est disponible sur GithubCC et pourra être utilisé par toute personne désireuse de mieux comprendre le fonctionnement des réseaux de neurones d'un point de vue interne, à la condition qu'elle maîtrise le français !

2. Gestion de projet

Le projet s'est effectué en 2 phases. La première était surtout destinée à prendre en main les différents outils à ma disposition, alors que la seconde m'a permis de programmer la bibliothèque. Dans chacune de ses phases, j'ai été amenée à faire des recherches et à coder.

2.1. Organisation

2.1.1. Fréquence de travail

J'ai essayé de travailler régulièrement sur le projet, c'est-à-dire chaque mardi après-midi. Cela a été globalement respecté quoiqu'il y a eu une légère baisse de régime en milieu de projet (entre la première et la seconde phase) et de légères hausses en début et fin de projet.

En dehors des mardis après-midi, il m'est arrivée de travailler le week-end et la semaine surtout lorsqu'on n'avait pas d'autres projets à rendre. J'ai également mis à profit les vacances de Pâques pour bien avancer sur le projet.

2.1.2. Temps de travail

Chaque fois que je travaillais sur mon projet, je passais entre 2 et 5 heures dessus. Je pense pouvoir affirmer que le temps passé sur la première phase a était beaucoup plus courte en terme de durée que la seconde. Durant la première phase, je devais faire des séances de 2 à 3 heures maximum, alors que durant la seconde, je faisait minimum 3 heures.

Si durant la première phase, il s'agissait juste de suivre des tutoriels ou réfléchir à l'organisation et les spécifications de mon programme, ce qui étaient assez vite rébarbatifs et ne prenaient que peu de temps, durant la seconde phase, j'ai dû coder ce que j'avais imaginé, ce qui était bien plus stimulant et m'a pris beaucoup plus de temps.

2.2. Communication

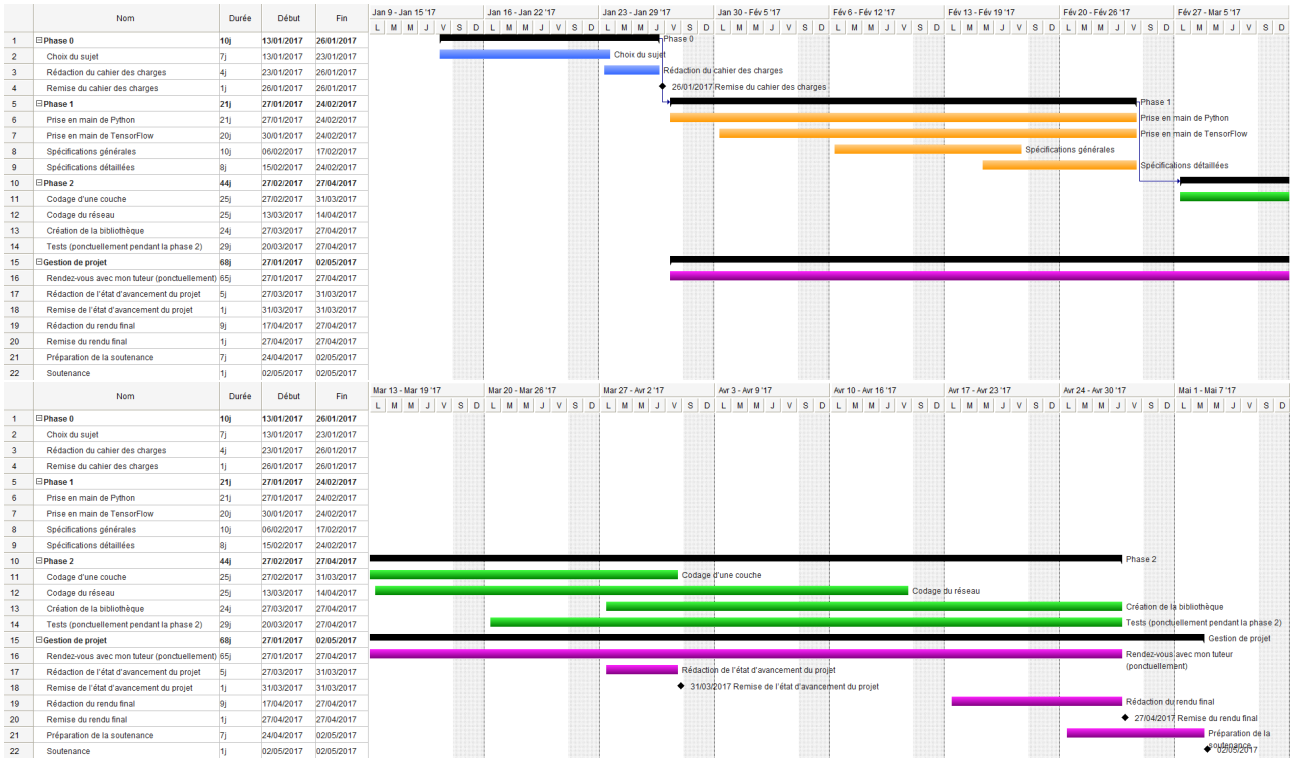
Durant mon projet, j'ai fait par de mon avancement à plusieurs reprises à mon tuteur, Serge Ariès, et à mon futur tuteur de stage, Olivier Bach, afin de leur dire où j'en étais et ce qu'il fallait encore que je fasse.

J'ai également demandé plusieurs fois conseils à Olivier Bach pour savoir si je partais dans la bonne direction et avoir des précisions lorsque j'étais bloquée par quelque chose au niveau algorithmique et organisation du code.

2.3. Planification

Dans l'ensemble, j'ai suivi le planning que je m'étais imposée au niveau de la durée de chaque phase. Au niveau de la phase 1, la durée de chaque sous-phase a également été respectée. Cependant, quelques modifications sont survenues dans la seconde phase. Les temps de travail dédiés pour la gestion de projet ont aussi été tenus à quelques jours près.

Voici ci-dessous le planning initial et le planning final :



	Nom	Durée	Début	Fin
1	**Phase 0**	**10j**	**13/01/2017**	**26/01/2017**
2	Choix du sujet	7j	13/01/2017	23/01/2017
3	Rédaction du cahier des charges	4j	23/01/2017	26/01/2017
4	Remise du cahier des charges	1j	26/01/2017	26/01/2017
5	**Phase 1**	**21j**	**27/01/2017**	**24/02/2017**
6	Prise en main de Python	21j	27/01/2017	24/02/2017
7	Prise en main de TensorFlow	20j	30/01/2017	24/02/2017
8	Spécifications générales	10j	06/02/2017	17/02/2017
9	Spécifications détaillées	8j	15/02/2017	24/02/2017
10	**Phase 2**	**44j**	**27/02/2017**	**27/04/2017**
11	Codage d'une couche	25j	27/02/2017	31/03/2017
12	Codage du réseau	25j	13/03/2017	14/04/2017
13	Création de la bibliothèque	24j	27/03/2017	27/04/2017
14	Tests (ponctuellement pendant la phase 2)	29j	20/03/2017	27/04/2017
15	**Gestion de projet**	**68j**	**27/01/2017**	**02/05/2017**
16	Rendez-vous avec mon tuteur (ponctuellement)	65j	27/01/2017	27/04/2017
17	Rédaction de l'état d'avancement du projet	5j	27/03/2017	31/03/2017
18	Remise de l'état d'avancement du projet	1j	31/03/2017	31/03/2017
19	Rédaction du rendu final	9j	17/04/2017	27/04/2017
20	Remise du rendu final	1j	27/04/2017	27/04/2017
21	Préparation de la soutenance	7j	24/04/2017	02/05/2017
22	Soutenance	1j	02/05/2017	02/05/2017

	Nom	Durée	Début	Fin
1	**Phase 0**	**10j**	**13/01/2017**	**26/01/2017**
2	Choix du sujet	7j	13/01/2017	23/01/2017
3	Rédaction du cahier des charges	4j	23/01/2017	26/01/2017
4	Remise du cahier des charges	1j	26/01/2017	26/01/2017
5	**Phase 1**	**21j**	**27/01/2017**	**24/02/2017**
6	Prise en main de Python	21j	27/01/2017	24/02/2017
7	Prise en main de TensorFlow	20j	30/01/2017	24/02/2017
8	Spécifications générales	10j	06/02/2017	17/02/2017
9	Spécifications détaillées	8j	15/02/2017	24/02/2017
10	**Phase 2**	**44j**	**27/02/2017**	**27/04/2017**
11	Codage d'une couche	25j	27/02/2017	31/03/2017
12	Codage du réseau	25j	13/03/2017	14/04/2017
13	Création de la bibliothèque	24j	27/03/2017	27/04/2017
14	Tests (ponctuellement pendant la phase 2)	29j	20/03/2017	27/04/2017
15	**Gestion de projet**	**68j**	**27/01/2017**	**02/05/2017**
16	Rendez-vous avec mon tuteur (ponctuellement)	65j	27/01/2017	27/04/2017
17	Rédaction de l'état d'avancement du projet	5j	27/03/2017	31/03/2017
18	Remise de l'état d'avancement du projet	1j	31/03/2017	31/03/2017
19	Rédaction du rendu final	9j	17/04/2017	27/04/2017
20	Remise du rendu final	1j	27/04/2017	27/04/2017
21	Préparation de la soutenance	7j	24/04/2017	02/05/2017
22	Soutenance	1j	02/05/2017	02/05/2017

Figure 1 – Planning initial

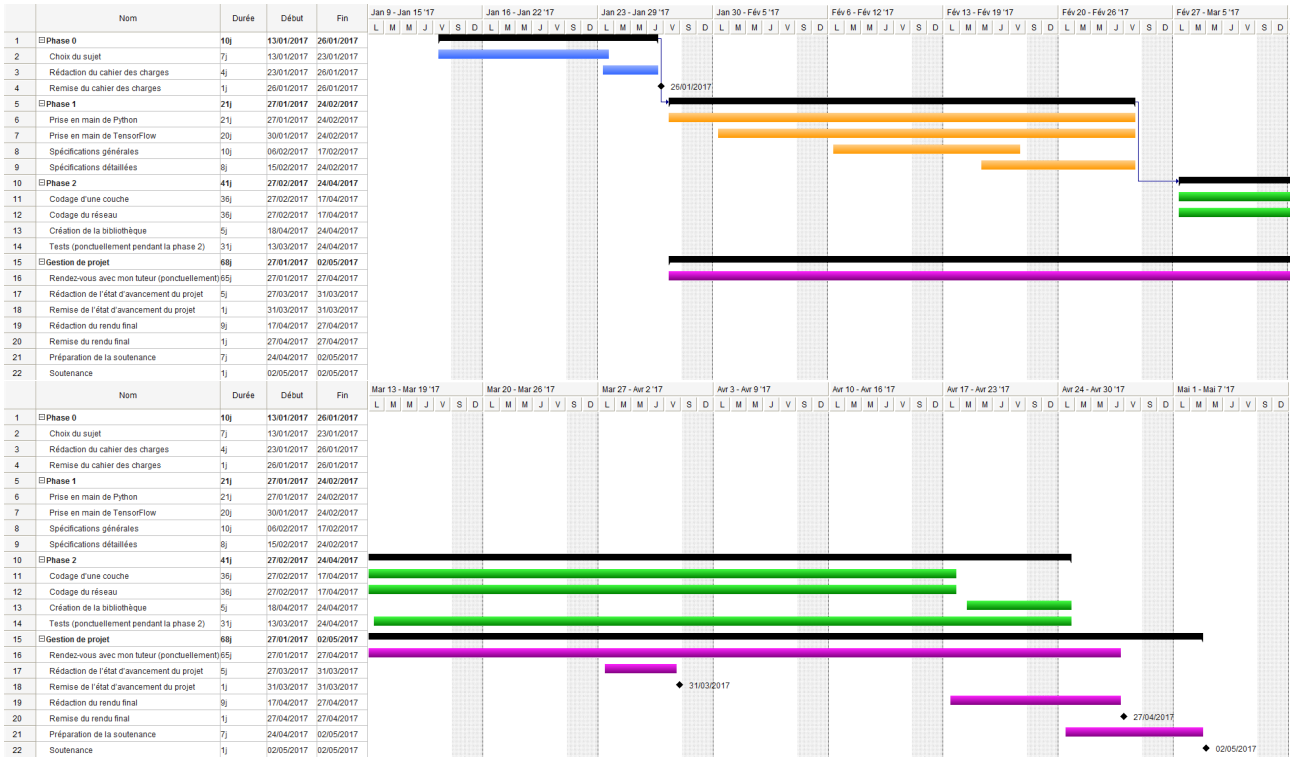


FIGURE 2 – Planning final

On peut donc voir que le codage de la couche et du réseau ce sont fait en parallèle, ce qui a permis d'avancer le début des tests. La création de la bibliothèque ne s'est faite qu'à l'issue du codage de la couche et du réseau et a duré moins de temps que prévu.

3. Présentation du projet

3.1. Installation

Avant d'installer le projet, il est indispensable d'avoir installé sur votre ordinateur Python et la bibliothèque Numpy au préalable. Ensuite, voici la procédure à suivre pour installer le projet :

1. récupérez sur votre ordinateur le projet `pii-neural-network`
2. ouvrez une invite de commandes à l'emplacement où vous avez rangé le projet
3. tapez `python setup.py install` dans l'invite de commandes
4. appuyez sur la touche `Entrée` de votre clavier

Vous pouvez désormais taper `python` dans l'invite de commandes et utiliser la bibliothèque `pii_neural_network` à votre envie ! Cette bibliothèque est composé de 4 packages :

- un package `couche` permettant de créer une couche de neurones
- un package `reseau` permettant de générer un réseau de neurones, de l'entraîner et le tester sur des données numériques
- un package `fonction` contenant les fonctions d'activation disponibles pour les couches de neurones
- un package`donnees` qui permet de récupérer les fichiers de la base de données MNIST

Ci-dessous, sera expliqué comment se servir de chaque package afin de montre les possibilités qu'ils offrent et comment s'en servir pour générer des réseaux de neurones.

3.2. Créer une couche

Pour créer une couche, on utilise le package `couche`. Il faut donc commencer par l'importer en tapant la commande `from pii_neural_network import couche` par exemple. On peut alors créer 4 types de couches :

- des couches d'entrées, autrement dit la première couche d'un réseau de neurones
- des couches totalement connectées, dont on se sert à la fois dans le perceptron multi-couches et le réseau neuronal convolutif, la dernière couche d'un réseau de neurones étant une couche totalement connectée
- des couches de convolution, dont on se sert exclusivement pour le réseau neuronal convolutif
- des couches de pooling (de regroupement), dont on se sert également exclusivement pour le réseau neuronal convolutif et sont toujours précédés d'une couche de convolution

3.2.1. Couche d'entrée

Pour créer une couche d'entrée, il suffit de connaître ses dimensions (hauteur et largeur). Ensuite, il suffit d'utiliser le constructeur dédié de la façon suivante : `CoucheEntree(hauteur, largeur)`.

Si vous voulez faire apprendre à votre réseau de neurones l'opérateur logique `XOR`, cela nécessite 2 valeurs d'entrée et donc une hauteur de 2 pour une largeur de 1. Si vous voulez lui faire apprendre la fonction `sinus`, cela nécessite 1 valeur d'entrée et donc une hauteur de 1 pour une largeur de 1. Si vous voulez lui faire apprendre une image de 28 pixels $\times$ 28 pixels, cela nécessite une couche d'entrée de hauteur et de largeur 28.

Voici ci-dessous les codes à entrer dans l'invite de commandes pour créer les 3 couches d'entrées que je viens de donnée à titre d'exemple dans le paragraphe précédent :

```
from pii_neural_network import couche

couche_xor = couche.CoucheEntree(hauteur = 2, largeur = 1)
couche_sinus = couche.CoucheEntree(2, 1)
couche_image = couche.CoucheEntree(28, 28)
```

3.2.2. Couche entièrement connectée

Pour créer une couche entièrement connectée, il faut préciser la hauteur de la couche qui correspond en fait au nombre de neurones sur la couchecar pour une couche entièrement connectée, il n'y a ni largeur, ni profondeur. Il faut également indiquer dans quel intervalle on initialisera les poids synaptiques de la couche, et dire quelle sera la fonction d'activation de la coucher. Ensuite, il suffit d'utiliser le constructeur dédié de la façon suivante : `CoucheConnectee(hauteur, initialisation, activation)`.

Par exemple, pour une couche entièrement connectée présentant 6 neurones, dont les poids synaptiques sont initialisées dans l'intervalle $[-0,5; 0,5]$ et dont la fonction d'activation est la fonction sigmoïde, on écrira :

```
from pii_neural_network import couche, fonction

couche = couche.CoucheConnectee(6, 0.5, fonction.sigmoide)
```

Notons que les fonctions d'activation ne peuvent pour l'instant être que `sigmoide` pour la fonction sigmoïde, `relu` pour la fonction ReLU ou `tanh` pour la fonction tangente hyperbolique.

3.2.3. Couche de convolution

Pour créer une couche de convolution, il faut préciser la profondeur de la couche, l'intervalle dans lequel les poids synaptiques seront initialisés, la fonction d'activation de la couche, la taille du filtre utilisé pour la convolution et éventuellement le pas (par défaut à 1). Ensuite, il suffit d'utiliser le constructeur dédie de la façon suivante : `CoucheConvoluee(profondeur, initialisation, activation, taille_filtre [, pas])`.

Par exemple, pour une couche de convolution ayant une profondeur de 2, dont les poids synaptiques sont initialisées dans l'intervalle $[-0,2; 0,2]$, dont la fonction d'activation est la fonction ReLU, qui prend ses valeurs après une convolution d'un filtre de taille 3×3 et dont le pas est de 1, on écrira par exemple :

```
from pii_neural_network import couche, fonction

couche = couche.CoucheConvoluee(2, 0.2, fonction.relu, 3)
```

Pour une couche de convolution, la hauteur et la largeur du filtre sont déduits en fonction de la hauteur et la largeur de la couche précédente dans le réseau, de la taille du filtre et du pas (qui représente le décalage du filtre entre chaque calcul).

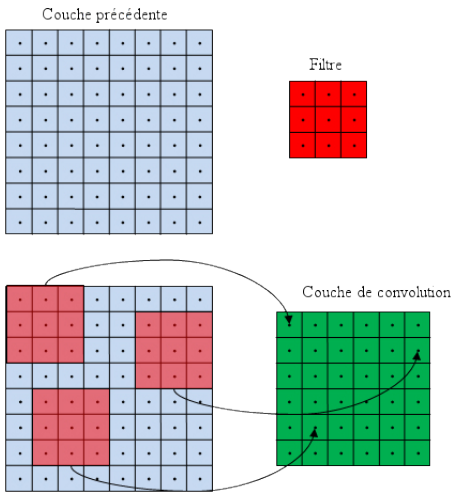


FIGURE 3 – Illustration de la convolution en 2 dimensions avec un pas de 1

3.2.4. Couche de pooling

Pour créer une couche de pooling, il faut uniquement donner la taille du regroupement à effectuer, c'est-à-dire combien on va « compresser » de neurones. Ensuite, il suffit d'utiliser le constructeur dédie de la façon suivante : `CouchePooling(taille_pooling)`.

Par exemple, pour une couche de pooling effectuant une compression de 2×2 neurones (le plus courant), il faudra écrire :

```
from pii_neural_network import couche

couche = couche.CouchePooling(2)
```

Lorsqu'on regroupe des neurones, le plus courant est de garder le neurone ayant la valeur maximale et d'utiliser une compression de 2×2. On appelle alors ce regroupement « Max-Pool 2×2 ».

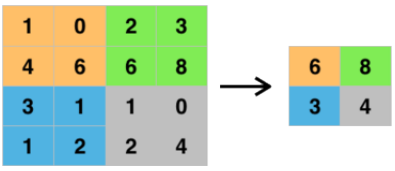


FIGURE 4 – Illustration du pooling « Max-Pool 2×2 »

On rappelle également qu'une couche de pooling doit obligatoirement être précédée d'une couche de convolution.

3.3. Créer un réseau de neurones

Une fois que vous avez créé l'ensemble des couches de votre réseau, comme indiqué dans la partie précédente, il est très simple de générer votre réseau de neurones. En effet, vous n'avez qu'à appeler le constructeur en lui donnant en argument les couches que vous désirez dans l'ordre, c'est-à-dire de la première à la dernière couche.

Par exemple, si on reprend l'exemple de l'image de 28 pixels par 28 pixels et qu'on souhaite générer un réseau neuronal convolutif avec une seule sortie, on peut le créer de la manière suivante :

```
from pii_neural_network import couche, fonction, reseau

couche_image = couche.CoucheEntree(28, 28)
couche_convoluee = couche.CoucheConvoluee(2, 0.3, fonction.relu, 5)
couche_pooling = couche.CouchePooling(2)
couche_connectee = couche.CoucheConnectee(1, 0.5, fonction.sigmoide)

reseau_image = reseau.Reseau(couche_image, couche_convoluee, couche_pooling,
couche_connectee)
```

Pour un réseau neuronal convolutif, voici quelques architectures assez courantes en ce qui concerne l'enchaînement des différents types de couches :

- `CoucheEntree` > `CoucheConvoluee` > `CoucheConnectee`
- `CoucheEntree` > `CoucheConvoluee` > `Pooling` > `CoucheConnectee`
- `CoucheEntree` > (`CoucheConvoluee` > `Pooling`) $\times 2$ > (`CoucheConnectee`) $\times 2$

3.4. Entraîner et tester un réseau de neurones

3.4.1. Entraîner un réseau de neurones

En supposant que vous avez déjà récupéré ou construit les données d'entraînement dans une variable `donnees` et que vous avez déjà créé un réseau de neurones dans une variable `reseau`, entraîner ce réseau est très simple.

Pour l'entraîner, il faut d'abord réfléchir au coefficient d'apprentissage (dans l'intervalle $]0, 1[$) et au nombre d'entraînements sur les données. Pour un nombre d'entraînement de 100 pour 100 données, on va donc 100 fois apprendre les 100 données, soit 10 000 apprentissages.

Ensuite, il suffit de faire appel à la méthode `entrainement` de la classe `Reseau` en lui passant en argument les données, le coefficient d'apprentissage choisi et le nombre d'entraînements à effectuer sur les données.

Par exemple, pour un coefficient d'apprentissage de 0,5 et et pour 500 itérations, il faudra écrire le code suivant :

```
reseau.entrainement(donnees, 0.5, 500)
```

Le coefficient d'apprentissage influe l'apprentissage. Un coefficient faible nécessite un apprentissage plus long (un nombre d'entraînement plus important), mais un coefficient d'apprentissage trop fort, risque de faire apprendre au réseau des choses fausses.

Le nombre d'entraînements influe sur le temps d'apprentissage. Plus le nombre d'entraînements est important, plus le temps d'apprentissage sera long. Cependant, si le nombre d'entraînements est trop faible, le réseau n'aura pas eu le temps de bien apprendre et il risque de faire trop d'erreurs.

Par ailleurs, dans le cas où le nombre de données est important, je recommande de ne prendre qu'un échantillon aléatoire de données, car plus il y a de données à traiter, plus le temps d'apprentissage sera long.

Durant l'apprentissage, par 10 fois (s'il y a plus de 10 entraînements), on affiche l'erreur quadratique que commet le réseau. Cela permet ainsi de voir sa progression dans l'apprentissage. Plus l'erreur est faible, meilleur sera le réseau.

3.4.2. Tester un réseau de neurones

En supposant que vous avez déjà récupéré ou construit les données de tests (qui peuvent être les mêmes que celles d'entraînement) dans une variable `donnees` et que vous avez déjà créé un réseau de neurones dans une variable `reseau`, tester celui-ci est également très simple.

Il suffit en effet de faire appel à la méthode `test` de la classe `Reseau` en lui passant en argument les données à tester.

```
reseau.test(donnees)
```

On va alors afficher pour chaque donnée, les valeurs d'entrée, les valeurs de sorties obtenues / prédites par le réseau et les valeurs de sortie attendues / désirées.

Dans le cas où les données de test sont les mêmes que les données d'entraînement, il est préférable de tester le réseau que sur un échantillon réduit de données. Il n'est *a priori* pas nécessaire de tester le réseau sur toutes les données d'entraînement.

3.5. Créer ou récupérer des données

Comme on vient de le voir, pour entraîner et tester un réseau de neurones, il est nécessaire d'avoir des données. Pour cela, on peut soit les créer, soit les récupérer.

La création de données est recommandée pour des choses simples telles que les opérateurs logiques (`AND`, `XOR`, etc.) ou les fonctions (`sinus`, fonction carré, etc.). On peut évidemment créer des données plus compliquées et plus longue, cependant s'il faut tout écrire à la main, cela risque de prendre du temps.

Actuellement, on ne peut récupérer facilement que la base de données MNIST disponible sur le site de Yann LeCun, à l'adresse suivante : `http://yann.lecun.com/exdb/mnist`. Cependant, si vous possédez d'autres fichiers contenants des données et que les récupérés dans un bon format (un liste contenant des couples `(valeurs_entrees, valeurs_sorties_desirees)`, il n'y a aucun problème.

Voici ci-dessous 2 exemples de création de données. Le premier permet de créer les données nécessaires à l'apprentissage de l'opérateur logique `OR` et le second à l'apprentissage de la fonction `cosinus` dans l'intervalle $[0, \pi]$.

```
import numpy as np

donnees_or = [
                [np.array([0.0, 0.0]), np.array([0.0])],
                [np.array([0.0, 1.0]), np.array([1.0])],
                [np.array([1.0, 0.0]), np.array([1.0])],
                [np.array([1.0, 1.0]), np.array([1.0])]
              ]

x = np.linspace(0, np.pi, 1000)
y = np.cos(x)
donnees_cos = [(i, j) for i, j in zip(x, y)]
```

Voici maintenant comment récupérer les données de la base de données MNIST en utilisant le package `donnees` et la fonction `recuperation`.

```
from pii_neural_network import donnees

(e_images, e_labels), (t_images, t_labels) = donnees.recuperation()
donnees_entrainement = [(x, y) for x, y in zip(e_images, e_labels)]
donnees_test = [(x, y) for x, y in zip(t_images, t_labels)]
```

4. Réalisation du projet

Le codage s'est effectué en 3 temps. Dans un premier temps, j'ai commencé par coder la classe `Reseau` et la classe `Couche` mais en ne pensant qu'au perceptron multi-couches. Dans un second temps, j'ai modifié le code de ces classes afin de pouvoir gérer un réseau neuronal convolutif. Enfin, dans un dernier temps, j'ai ajouté le code pour avoir une bibliothèque.

4.1. Perceptron multi-couches

J'ai commencé par comprendre comment fonctionner un perceptron multi-couches. Cela a donné lieu à la production d'un document qui explique comment on trouve les formules utilisées pour le codage du perceptron.

Initialement, j'ai commencé par créer 2 classes : la classe `Network` et la classe `Layer`, que j'ai par la suite renommé `Reseau` et `Couche`.

La première classe permettait, lorsqu'on lui passait en argument les couches qu'on souhaitait avoir, de construire le réseau, de l'entraîner et le tester. Elle avait donc un constructeur et 2 méthodes (une pour entraîner le réseau et une pour tester le réseau).

La seconde classe permettait, lorsqu'on lui donnait un nom (que j'ai supprimé vu qu'il ne servait à rien), le nombre de neurones, la fonction d'activation et l'intervalle dans lequel les poids synaptiques sont initialisés, de créer la couche associée. Elle n'avait donc qu'un constructeur qui mettait les informations dans un dictionnaire.

Le problème c'est que tous les calculs se faisaient dans la classe `Reseau`. Ainsi, il y avait beaucoup de codes dans `Reseau` et quasiment pas dans `Couche`.

J'ai donc décidé de « décentraliser » le code. Ainsi, j'ai ajouté 3 méthodes dans la classe `Couche` : une pour connecter une couche à la couche prédécesseur dans le réseau, une pour calculer les valeurs des neurones de la couche et une pour calculer les variations des matrices de poids et de biais de la couche ainsi que l'erreur à propager.

De cette façon, au lieu de créer les matrices des poids et des biais dans le constructeur du réseau, ce qui en faisaient des attributs de la classe `Reseau`, je les construis lorsque je connecte mes couches les unes avec les autres, ce qui m'a permis d'en faire des attributs de la classe `Couche`, ce que je trouve plus logique.

Code avant :

```
                                                                    Reseau

liste_poids = list()
for couche_precedente, couche in liste_couches:
    poids = np.random.uniform(-couche.initialisation, couche.initialisation,
        (couche.nb_neurones, couche_precedente.nb_neurones))
    liste_poids.append(poids)
```

Code après :

```
                                                                    Reseau

for couche_precedente, couche in liste_couches:
    couche.connexion(couche_precedente)
```

```
                                                                    Couche

def connexion(self, couche_precedente)
    self.poids = np.random.uniform(-self.initialisation, self.initialisation,
        (self.nb_neurones, couche_precedente.nb_neurones))
```

Par ailleurs, j'ai séparé chacune des méthodes de la classe `Reseau` en 2 méthodes. D'abord, j'ai séparé la méthode `entrainement`. Cette séparation est intervenue assez rapidement, avant même que je ne décide de rajouter des méthodes dans la classe `Couche`. Puis, j'ai séparé la méthode `test`. Cette séparation est, elle, intervenue juste après l'ajoute des 3 méthodes dans la classe `Couche`.

Par ailleurs, avant la séparation, la méthode `test` ne prenait qu'une seule donnée à la fois. Elle prend désormais une liste de données, ce qui permet d'afficher plusieurs résultats en un seul coup.

Pour finir, le constructeur d'une couche nécessitant une fonction d'activation, j'ai créé un fichier `fonction.py` contenant ces fonctions d'activation. Pour le moment, seulement 3 fonctions sont disponibles, mais il est possible d'en ajouter !

4.2. Réseau neuronal convolutif

Pour coder le réseau neuronal convolutif, j'ai repris la même architecture que précédemment.

Le fait d'avoir « décentraliser » les calculs dans la classe `Couche`, c'est que du coup, je n'ai pas eu besoin de modifier la classe `Reseau` lorsque j'ai ajouté les codes pour le réseau neuronal convolutif.

Au départ, je comptais faire une classe abstraite `Reseau` qui aurait deux héritiers : un pour le perceptron multi-couches et un pour le réseau neuronal convolutif.

Mais, compte tenu de l'évolution de l'architecture de mon code et qu'un réseau neuronal convolutif possède, parmi ses couches, des couches entièrement connectées, il m'a paru plus judicieux de créer une classe abstraite `Couche` qui aurait deux héritiers : un pour les couches entièrement connectées et un pour les couches de convolution.

Ainsi ma classe `Couche` s'est vue renommée `CoucheConnectee`, car la classe `Couche` est devenue une classe abstraite ayant un constructeur et 4 méthodes abstraites.

J'ai également ajouté une classe `CoucheConvoluee`, ce qui m'a obligé à créer une classe `CoucheEntree`. En effet, initialement, avec le perceptron multi-couches, toutes mes couches étaient entièrement connectées et se construisaient donc de la même façon. Or, une couche de convolution ne se construit pas pareil. La première couche peut notamment avoir 2 dimensions alors qu'une couche entièrement connectée n'a qu'une seule dimension.

Pour finir, j'ai ajouté une classe `CouchePooling` qui n'était pas prévue. Mais comme elle est importante dans un réseau neuronal convolutif, cela m'a fortement incitée à la prendre en compte.

La difficulté avec les couches de convolution et de pooling, c'est qu'elles peuvent avoir 3 dimensions, ce qui n'était pas le cas avec les couches entièrement connectées. La profondeur est donc un attribut de la classe `Couche` que j'ai dû rajouter après coup.

Enfin, pour pouvoir tester le réseau neuronal convolutif, j'ai décidé d'utiliser la base de données MNIST, ce qui a nécessité l'ajout d'une fonction permettant de récupérer ces données. J'ai donc créer un nouveau fichier (`donnees.py`) contenant une unique fonction. Cette fonction permet de récupérer la base de données MNIST. Il est tout à fait envisage de rajouter des fonctions à ce fichier qui permettent de récupérer d'autres types de données.

4.3. Bibliothèque

Au niveau de la construction d'une bibliothèque, j'ai suivi un tutoriel très bien fait sur Internet. Cela m'a amené à revoir un peu le rangement de mon code dans le dépôt Github, mais ça n'a pas été bien compliqué à faire vu le peu de fichiers qu'il y a dans mon dépôt...

Il a fallu que je rajoute un fichier `__init__.py` au niveau des fichiers contenant mes classes. J'ai également dû créer un fichier `setup.py` qui permet d'installer la bibliothèque sur un ordinateur et le fichier `MANIFEST.in` qui permet d'indiquer les fichiers à récupérer lor de l'installation.

5. Bilan du projet

5.1. Points positifs & Apport pédagogique

Ce projet m'a permis d'effectuer seule un projet ce qui a été très formateur, surtout au niveau de la gestion de projet. J'ai ainsi appris qu'il était bien plus facile de se motiver lorsqu'on travaille en groupe, mais qu'on était généralement plus performant tout seul. De plus, j'ai beaucoup apprécié la grande liberté que permet un travail individuel.

Ce projet m'a également permis de travailler sur un sujet que j'aime, à savoir l'intelligence artificielle. J'ai ainsi pu apprendre un certain nombre de nouvelles choses sur les réseaux de neurones, leur fonctionnement, leur utilité, ce qu'ils permettent de faire, etc. ce qui a été à la fois intéressant et enrichissant.

Durant ce projet, j'ai été amenée à faire quelques recherches, que ça soit sur les réseaux de neurones ou sur Python, et comme la documentation est souvent en anglais, je pense avoir un peu amélioré mon anglais au niveau du vocabulaire, que ça soit pour formuler mes requêtes ou pour comprendre les résultats.

Ce projet m'a également appris à me remettre en question, car il m'est arrivée faire du code et des recherches qui ne débouchaient nul part. Enfin, cela m'a permis de prendre conscience de l'importance de la phase de spécifications du code et à bien se projeter.

5.2. Points négatifs & Difficultés

Ce projet visait surtout à me faire appréhender le fonctionnement d'un réseau de neurones en en codant un moi-même. Ainsi, il n'y a eu aucun rendu visuel à faire ce qui a parfois été assez difficile à gérer. En effet, parfois j'avais l'impression de passer des heures sur mon code pour juste avoir l'affichage 4 lignes de texte dans une console, ce qui n'était pas très stimulant...

Je n'ai pas eu de gros problèmes au niveau de la gestion de projet, si ce n'est quelque passage à vide où il était assez difficile de me motiver. Cependant, je pense que le cahier des charges a été respecté et les délais aussi.

Au niveau du code, j'ai eu quelques hésitation quant à la gestion de la profondeur pour les couches de convolution et de pooling, et j'ai eu également des difficultés à bien programmer la rétro-propagation du gradient. J'avais en effet tendance à me mélanger les pinceaux au niveau des transposées, des produits matriciels ou des produits « un à un ».

J'ai également rencontré quelques problèmes lors du codage pour la récupération de la base de données MNIST. Finalement, c'est juste qu'il fallait décompresser les fichiers... Mais, j'ai mis du temps à m'en rendre compte ce qui m'a fait perdre pas mal de temps

6. Conclusion

S'il est possible de créer un réseau de neurones, que ça soit un perceptron multi-couches ou un réseau neuronal convolutif, ce qui était l'objectif du projet, de nombreuses choses restent à faire.

Une amélioration que j'aurais bien aimer mettre en place est qu'on puisse tester les réseaux sur autres choses que la base de données MNIST ou des données simples (opérateurs logiques et fonctions).

Il faudrait également que l'utilisateur ait plus de souplesse quant à la connexion des couches les unes aux autres. En effet, avec le présent projet, les matrices des poids sont initialisées aléatoirement dans un intervalle et les matrices des biais sont initialisées à 1. Par ailleurs, pour le pooling, on utilise forcément un « Max-Pool ».

Enfin, il aurait été intéressant de mettre en place une couche de perte pour le réseau neuronal convolutif.

7. Quelques informations pour comprendre les réseaux de neurones

Voici le principe de connexion entre 2 couches de neurones x et y :

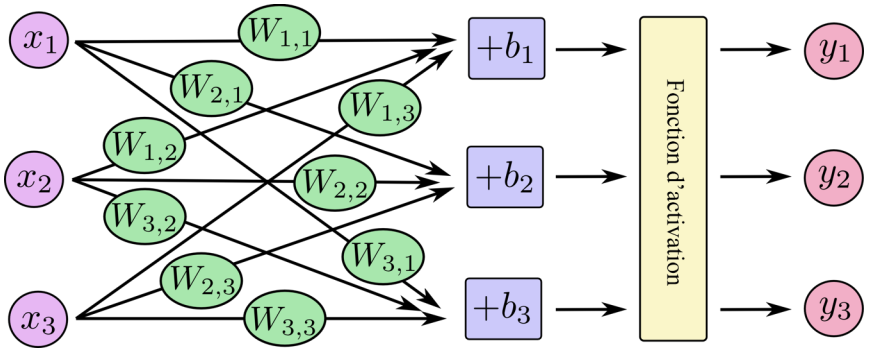


FIGURE 5 – Connexion entre 2 couches de neurones

On peut donc voir que les valeurs x_i des neurones de la couche x sont liées aux valeurs y_j des neurones de la couche y par des poids synaptiques $W_{j,i}$ auquel on ajoute un biais b_j et qu'on fait passer à travers une fonction d'activation.

Pour ajuster les poids, on utilise la méthode de rétropropagation du gradient. Cette méthode consiste à comparer les sorties désirées aux sorties calculées par le réseau et de minimiser cette écart. Pour cela, il suffit de calculer la dérivée de l'écart par rapport aux poids synaptiques et aux biais. Pour en savoir plus, consultez le document `theorie.pdf` joint avec ce rapport.

Pour calculer les valeurs des neurones d'une couche, en fonction des valeurs de la couche précédente, on peut voir d'après la figure précédente qu'il suffit de multiplier prendre toutes les valeurs des neurones qui sont connectés avec le neurone dont on veut connaître la valeur, de les multiplier par leur poids synaptiques, de sommer les résultats obtenus, d'ajouter le biais associé au neurone dont on recherche la valeur et ensuite de prendre l'image du résultat par la fonction d'activation.

Par exemple, dans l'exemple précédent, pour obtenir y_1, il suffit de faire $f\left(x_1 W_{1,1} + x_2 W_{1,2} + x_3 W_{1,3} + b_1\right)$ où f est la fonction d'activation.