Introduction à l’apprentissage profond

**Partie 1 : Perceptron**

**Une image contenant capture d’écran, carré, Rectangle, Police

Description générée automatiquement**

Figure : Base de données MNist

Les tenseurs sont des structures de données utilisées pour représenter et stocker des données multidimensionnelles. Ces tenseurs sont définis en fonction de leur rang (nombre de dimensions), leur forme (taille) et le type de données qu’ils contiennent. Les tenseurs dans le fichier *perceptron pytorch.py* sont les suivants :

• représente les données d’entraînement, pour chaque image on nous indique 28\*28=784 pixels/entrées, donc une taille de (63 000, 784)

• représente les labels des données d’entraînement, stockée sous la forme d’un vecteur d’un codage one-hot de taille 10, soit une taille de (63 000, 10)

• représente les données de test, de taille, de fois 7 000 cette fois et toujours 784 entrées (7 000, 784)

• représente les labels des données de test, de taille ici aussi 10 pour le codage one-hot (7 000, 10)

• représentant les poids du modèle. Il est précisé dans l’énoncé que l’entrée de chaque réseau a 784 entrées (28x28 pixels) et 10 sorties. Ainsi le tenseur représentant les poids est de dimension 2 et sa taille est telle que (784,10).

• qui représente les biais du modèle. Il est également de dimension 2 et initialisé à 1 pour sa première dimension et comme pour les poids, à 10 (nombre de classes de sorties). Il est donc de taille (1,10)

• est un tenseur permettant de récupérer l’entrée. Il est défini en fonction du batch size et le nombre de pixels par image. Il est donc de taille (5, 10).

• est un tenseur permettant de calculer la sortie du modèle. Sa taille est définie par le nombre de sorties et son nombre de classes soit (5, 10)

• représente les vrais labels, il est de même taille que soit (5, 10).

• est un tenseur permettant de mettre à jour les poids en stockant le gradient calculé. Il est de même taille que et car il résulte de leur différence, donc de taille (5,10).

**Partie 2 : Shallow network**

*(Voir le fichier Shallow\_network.py)*

Hyperparamètres  
Avec les paramètres que j’ai fixé de base :

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.00001 # taux d'apprentissage

hidden\_size = 128 # nb neurones de la couche cachée

# poids initiaux

w\_min = - 0.0001

w\_max = 0.0001

Nous avons un taux de bonnes réponses qui commence à *0.0870* pour monter à *0.3700* après les 10 lectures. Nous allons maintenant jouer avec les hyperparamètres en modifiant à chaque fois un seul de ces paramètres et en maintenant les autres aux valeurs ci-dessus.   
  
• En jouant sur le taux d’apprentissage η, voici les résultats obtenus :

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| η | 0.00001 | 0.00002 | 0.00005 | 0.0001 | 0.0002 | 0.0005 | 0.001 | 0.002 | 0.005 | 0.01 |
| Taux initial bonnes réponses | **0.1094** | **0.1399** | **0.2221** | **0.4187** | **0.5851** | **0.7731** | **0.6780** | **0.8216** | **0.6786** | **0.5621** |
| Taux final bonnes réponses | **0.3730** | **0.4887** | **0.5069** | **0.7456** | **0.6646** | **0.8910** | **0.6499** | **0.9007** | **0.8353** | **0.5621** |

Le taux d’apprentissage est un paramètre influant sur la vitesse à laquelle le modèle apprend à partir des données d’entrée. Nous voyons ici que le taux d’apprentissage à 0.002 est le plus intéressant car cette valeur donne la convergence vers une valeur élevée pour le taux de bonne réponse et ce, avec une petite oscillation seulement. Une valeur de η de 0,002 indique que le modèle effectue des mises à jour de poids relativement petites à chaque itération d'entraînement. Cela montre également une convergence rapide, ce qui indique que le modèle a une efficacité d'apprentissage élevée, car il atteint de bonnes performances avec relativement peu d'époques d'entraînement. Nous remarquons aussi qu’il n’est pas judicieux de prendre une valeur du taux d’apprentissage trop élevé. En effet, alors que le taux d’apprentissage influe sur l’apprentissage en fonction de l’entrée courante, s’il est trop élevé les mises à jour des poids du modèles seront trop élevées également, ce qui entraîne ce que l’on appelle l’overshooting, le modèle ne converge plus correctement ce qui aboutit à une baisse de performance.

Pour la suite des tests, nous gardons les hyperparamètres de base et ne prenons pas les valeurs « optimales » afin qu’ils n’influent pas trop les résultats de nos tests.

• Pour les poids initiaux, on peut essayer de creuser l’écart de poids minimal et maximal afin d’observer si une divergence pourrait apparaître. Ainsi voici les résultats avec les différences de poids ( et sont opposés) suivantes : 0.0002, 0.002, 0.02, 0.2, 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0.0002 | 0.002 | 0.02 | 0.2 | 2 |
| Taux initial de bonnes réponses | **0.0979** | **0.1094** | **0.1126** | **0.2391** | **0.1763** |
| Taux final de bonnes réponses | **0.2226** | **0.4131** | **0.6650** | **0.6090** | **0.2391** |

Ainsi les meilleurs résultats en termes de bonnes réponses sont pour un poids de ± 0.01 même si on remarque quelques oscillations pour un tel poids. Prendre un écart plus ou moins important entre les poids initiaux influera sur la capacité du modèle à partir de points différents dans l’espace des poids à chaque exécution de l’entraînement. Ainsi elle peut permettre d’explorer plus efficacement l’espace des poids pour trouver un minimum global. Des valeurs trop faibles (trop proche de 0) sont cependant déconseillées car le modèle pourrait avoir du mal à sortir des minimums locaux. Ainsi cette valeur permet de maintenir un équilibre entre sous-apprentissage et sur-apprentissage.

• Nombre de neurones de la couche cachée : De même nous allons essayer pour différents nombres de neurones puis nous analyserons les résultats (1, 10, 50, 200, 500 et 1 000)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Nb neurones | 1 | 10 | 50 | 200 | 500 | 1000 |
| Taux initial de bonnes réponses | **0.1026** | **0.1371** | **0.2093** | **0.1881** | **0.1350** | **0.1400** |
| Taux final de bonnes réponses | **0.1091** | **0.2647** | **0.4101** | **0.6593** | **0.6911** | **0.5960** |

On remarque qu’un nombre assez conséquent de neurones dans la couche cachée est préférable (environ 500 ici). En effet, un grand nombre de neurones dans la couche cachée permet d’avoir une meilleure capacité de modélisation (le modèle apprend des représentations plus complexes des données). Dans notre cas, cela signifie que notre modèle s’apparente à un réseau de neurones profond. Cependant un nombre de neurones trop grand peur augmenter le risque de surajustement, également cela nécessite plus de temps pour l’entraînement car plus de paramètres à ajuster. Pour un modèle plus complexe, il sera alors préférable de choisir un nombre bien moins important, c’est pourquoi nous choisirons 256 (entre 200 et 500) neurones.

Nous pourrions donc essayer de combiner nos résultats sur notre jeu de données afin de vérifier la compatibilité de nos résultats. Voici les résultats obtenus :

**Une image contenant texte, Police, typographie, capture d’écran

Description générée automatiquement**

Nous arrivons à un taux de bonnes réponses de 0.9826. Cela montre un bon apprentissage, même si l’on a un taux de bonnes réponses initial déjà très bon (plus de 95%), le modèle a permis de généraliser à partir des données d’entrainement pour encore mieux apprendre sans surapprentissage ou autre. Nous noterons que le taux d’apprentissage aura été le paramètre le plus important et qu’il paraît ici optimal.

**Partie 3 : Deep network**

*(Voir le fichier Deep\_network.py)*

Hyperparamètres  
Avec les paramètres fixés de base :

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.00001 # taux d'apprentissage

number\_of\_hidden\_layers = 2 # nombre de couches cachées

number\_of\_neurons\_per\_hidden\_layer = [128, 128] #nombre de neurones pour chaque couche cachée

# poids initiaux

w\_min = -0.0001

w\_max = 0.0001

Et en travaillant avec deux couches cachées, nous un taux de bonnes réponses qui commence à *0.0921* et qui monte à *0.1084.* Nous allons maintenant jouer avec les hyperparamètres.

• Ici aussi nous allons commencer par changer la valeur du taux d’apprentissage. Voici les résultats que l’on obtient :

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| η | 0.00005 | 0.0001 | 0.0002 | 0.0005 | 0.001 | 0.002 | 0.005 | 0.01 | 0.02 | 0.05 |
| Taux initial de bonnes réponses | **0.1084** | **0.1084** | **0.1084** | **0.5137** | **0.8879** | **0.9361** | **0.9636** | **0.9667** | **0.9713** | **0.0979** |
| Taux final de bonnes réponses | **0.4533** | **0.8790** | **0.9376** | **0.9701** | **0.9801** | **0.9821** | **0.9827** | **0.9826** | **0.9809** | **0.0979** |

Nous savons d’expérience que le taux d’apprentissage est le paramètre crucial à prendre en compte, en effet le modèle mettra plus ou moins du temps à converger en fonction de la valeur de ce dernier. Nous remarquons ici que notre modèle se bloque à une certaine valeur (0.1084) pour des valeurs trop faibles, cette stagnation s’explique par un blocage dans un minimum local sous-optimal et ce taux d’apprentissage trop faible fait que la mise à jour des poids se fait trop lentement, causant une convergence vers un minimum local. Ici aussi, nous trouvons une valeur optimal avoisinant 0.002/0.005. Pour des valeurs trop élevées, nous retrouvons le problème d’overshooting lié à des mises à jour de poids trop élevées.

• Maintenant nous allons toucher aux poids initiaux :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0.0002 | 0.002 | 0.02 | 0.2 | 2 |
| Taux initial de bonnes réponses | **0.0921** | **0.0921** | **0.1024** | **0.4034** | **0.1763** |
| Taux final de bonnes réponses | **0.1084** | **0.1084** | **0.1084** | **0.8236** | **0.2391** |

Nous remarquons que notre modèle a du mal à apprendre des données d’entraînement pour cause d’un taux d’apprentissage trop faible. Pour la suite nous allons fixer η à 0.0001 afin d’obtenir des résultats plus vite intéressants sans que le taux d’apprentissage n’influe trop nos résultats.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0.05 | 0.1 | 0.2 |
| Taux initial de bonnes réponses | **0.7336** | **0.8409** | **0.8150** |
| Taux final de bonnes réponses | **0.9356** | **0.9409** | **0.9207** |

Ici l’on trouve les meilleurs résultats pour . Encore une fois on prend des valeurs symétriques pour favoriser la convergence et éviter la saturation. Avec ces valeurs, le modèle a permis une convergence rapide et efficace, ce qui paraît optimal.

• Nous allons maintenant étudier l’influence du nombre de neurones par couche cachée.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nb neurones | 1 | 10 | 50 | 200 |
| Taux initial de bonnes réponses | **0.1113** | **0.1084** | **0.1084** | **0.1084** |
| Taux final de bonnes réponses | **0.2263** | **0.1969** | **0.1084** | **0.1084** |

Pour rappel, la couche cachée est une couche entre la couche d’entrée et de sortie qui joue le rôle de transformateur par apprentissage. Plus il y a de neurones dans la couche cachée, plus le modèle peut apprendre de représentations complexes et abstraites des données. Ainsi une couche cachée avec un petit nombre de neurones peut avoir du mal à capturer des motifs complexes dans les données. Ces résultats montrent la complexité du modèle, en choisissant peu de neurones on limite la complexité de notre modèle.

Nous allons donc réessayer nos tests en prenant des meilleures valeurs de poids initiaux (à savoir ± 0.1) et η = 0.0001 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nb neurones | 8 | 64 | 256 | 512 |
| Taux initial de bonnes réponses | **0.2240** | **0.7720** | **0.8803** | **0.9047** |
| Taux final de bonnes réponses | **0.6921** | **0.9267** | **0.9501** | **0.9591** |

Avec ces paramètres, nous avons ici encore des meilleurs résultats avec un nombre plus conséquent, mais cela nécessite plus de temps pour l’entraînement car plus de paramètres à ajuster. Pour un modèle plus complexe, il sera alors préférable de choisir un nombre bien moins important, c’est pourquoi nous choisirons 256 neurones ici aussi. Nos résultats indiquent qu’un modèle avec plus de neurones est capable de capturer des motifs plus complexes dans les données et d'atteindre une meilleure performance.

• Nombre de couches

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.0001 # taux d'apprentissage

number\_of\_hidden\_layers = 2 # nombre de couches cachées

# poids initiaux

w\_min = -0.05

w\_max = 0.05

Et 128 neurones par couche (On change évidement le code de façon à pouvoir intégrer plus de deux couches cachées).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nb couches cachées | 2 | 3 | 4 | 5 |
| Taux initial de bonnes réponses | **0.7594** | **0.4467** | **0.1084** | **0.1084** |
| Taux final de bonnes réponses | **0.9367** | **0.9166** | **0.7160** | **0.1084** |

On remarque plusieurs points importants face à ces résultats. Premièrement, l’ajout de couche cachée augmente la complexité du modèle, ce qui signifie une plus grande sensibilité aux problèmes de surapprentissage donc moins de performances. Ce problème de surapprentissage se traduit par le fait que le modèle semble avoir du mal à généraliser à partir des données d’entraînement des nouvelles données. Nous remarquons aussi que le taux initial de bonnes réponses change drastiquement selon le nombre de couches cachées, le modèle avec deux couches cachées donne le meilleur résultat initial et il sera donc plus simple d’apprendre pour ce modèle. Cependant, le modèle avec 3 couches cachées à un taux initial beaucoup plus élevé mais le taux final est quasiment le même que pour le premier modèle, ce qui montre une meilleure performance même si la performance finale est un peu moins intéressante. On peut donc imaginer qu’il est le modèle qui « apprend » le plus vite et, il faut prendre en compte l’influence des autres hyperparamètres (si l’on prend les valeurs optimales trouvées avant pour les autres paramètres, alors il sera tout aussi intéressant de travailler avec plus de couches cachées). En résumé, le nombre de couches cachées le plus « sûr » est de 2 mais on ne peut pas conclure quant à un choix de nombre de couches cachées optimales car ce choix dépendra des autres paramètres choisis globalement.

• La taille des mini batchs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Taille mini batchs | 2 | 5 | 15 | 50 | 200 |
| Taux initial de bonnes réponses | **0.7263** | **0.7404** | **0.7370** | **0.7423** | **0.7394** |
| Taux final de bonnes réponses | **0.9330** | **0.9329** | **0.9343** | **0.9336** | **0.9334** |

La taille des mini batchs correspond au nombre de données lues à chaque fois, c’est-à dire au nombre d’exemples de données d’entraînement utilisés lors d’une itération. On peut alors imaginer que plus cette taille sera élevée, plus l’apprentissage sera rapide. Ici on remarque que ce paramètre n’influe que très peu sur nos résultat (voir de façon négligeable).

Ce qui signifie que nous pouvons choisir une grande taille des mini batchs sans compromettre la performance du modèle, cela jouera un rôle plus important sur des jeux de données plus grands encore.

Encore une fois nous allons essayer de reprendre des paramètres plus optimaux afin de voir si la taille des mini batchs aura plus d’influence, nous prendrons et η=0.0001. Voici ce que l’on obtient :

|  |  |  |
| --- | --- | --- |
| Taille mini batchs | 5 | 500 |
| Taux initial de bonnes réponses | **0.8131** | **0.7971** |
| Taux final de bonnes réponses | **0.9397** | **0.9381** |

Ces résultats nous montrent que l’on ne peut pas vraiment conclure quant à l’influence de la taille des mini batchs. Nous remarquerons tout de même qu’une grande valeur permet de faire l’entraînement plus rapidement.

Comme pour la partie précédente, nous allons faire un dernier test avec les valeurs optimales :

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.002 # taux d'apprentissage

number\_of\_hidden\_layers = 3 # nombre de couches cachées

number\_of\_neurons\_per\_hidden\_layer = [128, 128, 128] # nombre de neurones pour chaque couche cachée

# poids initiaux

w\_min = -0.1

w\_max = 0.1

Et nous obtenons :

Une image contenant texte, Police, typographie, capture d’écran

Description générée automatiquement

Encore une fois nous arrivons à un taux de bonnes réponses excellent (0.9811). Donc encore meilleur que le précédent, cela montre qu’une meilleure combinaison comptant encore plus de paramètres obtiennent que l’on trouve des résultats toujours plus excellents. Nous noterons que le taux d’apprentissage aura été le paramètre le plus important encore une fois !

**Partie 4 : CNN**

Un réseau de neurones convolutif simple est un type de réseau de neurones artificiels conçu pour traiter efficacement des données en grille, telles que des images. Elle repose sur plusieurs aspects :  
• La Convolution : on applique un filtre à une région de l’image d’entrée, puis l’on déplace le filtre sur l’ensemble des images pour extraire des caractéristiques spécifiques (bordures, formes…)

• La couche de pooling : on réduit la résolution spatiale de l’image en préservant les caractéristiques essentielles relevées.

• La couche connectée : certaines couches fusionnent alors les caractéristiques extraites par les couches précédentes pour effectuer une classification.

Pour implémenter notre réseau CNN, nous allons nous baser sur LeNet5 (réseau convolutif utilisé pour la reconnaissance de caractères manuscrits). Selon le document LeCun5 (développeur de LeNet5), ce réseau est composé de 6 couches : deux de convolutions, deux de pooling puis deux couches connectées.

Une image contenant texte, ligne, diagramme, capture d’écran

Description générée automatiquement

En suivant ce schéma, nous pouvons déduire que :  
• La première couche est une couche de Convolution prenant en entrée une image 32x32 (il faudra alors redimensionner notre image).

• La deuxième couche est une couche de pooling qui réduira la taille à 14x14

• La troisième couche est une couche de convolution

• La quatrième couche est une ouche de pooling qui réduira la taille à 5x5

• La cinquième couche est une couche connectée d’entrée 120

• La dernière couche est une couche connectée d’entrée 84

A l’aide d’informations trouvées dans le document et la documentation des fonctions de Pytorch trouvées sur internet, j’ai pu proposer le code correspondant au réseau de neurones convolutif simple.

*(Voir le fichier CNN.py)*

Et ainsi, avec les données suivantes :

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.001 # taux d'apprentissage

Nous obtenons :

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

Ce qui montre la grande efficacité du modèle CNN.

**Partie 5 : Pour aller plus loin**

*(Voir le fichier perceptron pytorch extend.py)*

Nous avions de base comme résultat (en gardant le même modèle que perceptron avec Pytorch) et les paramètres suivants :

batch\_size = 5 # nombre de données lues à chaque fois

nb\_epochs = 10 # nombre de fois que la base de données sera lue

eta = 0.00001 # taux d'apprentissage

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

En utilisant la fonction de coût de cross entropie nous obtenons :

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

Donc la fonction de coût de cross entropie ne semble pas plus efficace. On garde cette fonction et on ajoute la fonction de gradient Adam :

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement