POLYTECH LYON

Projet - Apprentissage Profond

Polytech Lyon
5A Informatique

Rapport de projet

Ahmad EL KAAKOUR et Matthieu RANDRIANTSOA

Table des matières

1 Perceptron **3**
- 1.1 Analyse des tenseurs . . . 4

2 Shallow network **6**
- 2.1 Implémentation . . . 7
- 2.2 Optimisation du modèle . . . 7
  - 2.2.1 Étape 1 : Préparation du jeu d'entrainement, validation et test . . . 7
  - 2.2.2 Étape 2 : Définir les hyperparamètres et plages de valeurs à explorer . . . 8
  - 2.2.3 Étape 3 : Méthode d'optimisation des hyperparamètres . . . 8
- 2.3 Analyse et comparaison des modèles . . . 9
- 2.4 Analyse des hyperparamètres . . . 10
  - 2.4.1 Taux d'apprentissage . . . 11
  - 2.4.2 Taille des mini-batchs . . . 13
  - 2.4.3 Nombre de neurones . . . 15
  - 2.4.4 Corrélation entre hyperparamètres . . . 18
- 2.5 Analyse de l'entrainement des modèles . . . 19
- 2.6 Sélection et évaluation sur le jeu de test final . . . 20

3 Deep network **22**
- 3.1 Méthodologie . . . 23
- 3.2 Implémentation . . . 23
- 3.3 Analyse des résultats des modèles . . . 24
  - 3.3.1 Variante à deux couches . . . 24
  - 3.3.2 Variante à trois couches . . . 25
- 3.4 Analyse de l'entraînement des modèles . . . 26
- 3.5 Sélection et évaluation sur le jeu de test final . . . 28

4 CNN **29**
- 4.1 Redimensionnement des images . . . 30
- 4.2 Choix de l'architecture . . . 30
- 4.3 Entraînement des modèles . . . 33
  - 4.3.1 Avec réinitialisation des poids . . . 34
  - 4.3.2 Avec réentrainement complet des poids entrainés sur ImageNet1K_V1 . . . 34
  - 4.3.3 Avec gel partiel des couches . . . 34
- 4.4 Évaluation et modèle final . . . 35

Introduction

Ce rapport présente notre projet d'apprentissage automatique, en répondant aux questions posées par le sujet. Il détaille la méthodologie employée, l'analyse des résultats obtenus, ainsi que l'organisation du travail.

Les extraits de code présents dans ce rapport illustrent la démarche suivie. L'implémentation complète est fournie dans les fichiers accompagnant ce rapport.

Les résultats obtenus le long du projet sont enregistrés au format JSON afin de faciliter leur réutilisation. Enfin, les modèles retenus sont également enregistrés et disponible avec l'archive accompagnant ce rapport.

Chapitre 1

Perceptron

1.1 Analyse des tenseurs

Nous avons à notre disposition deux implémentations d'un même perceptron. Nous allons nous intéresser au fichier `perceptron_pytorch.py`. Voici une analyse détaillée des différents tenseurs présents dans l'implémentation. Pour rappelle, par définition un tenseur représente un tableau multidimensionnel. Dans le cadre de Pytorch, la plupart des objets manipulés au sein du framework constituent des tenseurs.

1. `data_train`
   - **Dimension** : `torch.Size([63000, 784])`
   - **Description** : Contient les images du jeu d'entraînement, chaque image étant convertie en un vecteur de 784 éléments correspondant aux pixels (28x28 pixels). La dimension `(63000, 784)` indique `(nombre_d_exemples_d'entraînement, nombre_de_pixels_par_image)`.
2. `label_train`
   - **Dimension** : `torch.Size([63000, 10])`
   - **Description** : Contient les étiquettes associées aux images d'entraînement, représentées sous forme de vecteurs one-hot pour 10 classes. La dimension `(63000, 10)` correspond à `(nombre_d'exemples_d'entraînement, nombre_de_classes)`.
3. `data_test`
   - **Dimension** : `torch.Size([7000, 784])`
   - **Description** : Contient les images du jeu de test, chaque image étant aplatie en un vecteur de 784 pixels. La dimension `(7000, 784)` se traduit par `(nombre_d'exemples_de_test, nombre_de_pixels_par_image)`.
4. `label_test`
   - **Dimension** : `torch.Size([7000, 10])`
   - **Description** : Contient les étiquettes associées aux images de test, représentées sous forme de vecteurs one-hot pour 10 classes. La dimension `(7000, 10)` indique `(nombre_d'exemples_de_test, nombre_de_classes)`.
5. `w`
   - **Dimension** : `torch.Size([784, 10])`
   - **Description** : Matrice des poids du modèle reliant chaque pixel d'entrée à une classe de sortie. Sa dimension est `(nombre_de_pixels_par_image, nombre_de_classes)`, soit `(784, 10)`.
6. `b`
   - **Dimension** : `torch.Size([1, 10])`

- **Description** : Vecteur des biais du modèle, chaque biais étant associé à une classe de sortie. La dimension est `(1, nombre_de_classes)`, soit `(1, 10)`.

7. `y`
   - **Dimension** : `torch.Size([5, 10])`
   - **Description** : Résultat du calcul `x @ w + b`, représentant les sorties du modèle. La dimension est `(batch_size, nombre_de_classes)`, soit `(5, 10)`.
8. `t`
   - **Dimension** : `torch.Size([5, 10])`
   - **Description** : Contient les étiquettes correctes pour le lot en cours, exprimées sous forme de vecteurs one-hot. La dimension est `(batch_size, nombre_de_classes)`, soit `(5, 10)`.
9. `grad`
   - **Dimension** : `torch.Size([5, 10])`
   - **Description** : Gradient de la fonction de perte par rapport aux sorties du modèle `(t - y)`. La taille est `(batch_size, nombre_de_classes)`, soit `(5, 10)`.
10. `b.sum(axis=0)`
    - **Dimension** : `torch.Size([10])`
    - **Description** : Somme des gradients de biais sur le lot en cours. La somme des gradients sur la dimension 0 (le batch) donne un vecteur 1D de taille 10, correspondant à chaque classe de sortie.
11. `torch.argmax(y, 1)`
    - **Dimension** : `torch.Size([batch_size])`
    - **Description** : Retourne l'indice de la classe avec la plus haute probabilité pour chaque prédiction du lot. Le résultat est un vecteur 1D de taille `(batch_size)`, ici `(5)`.
12. `torch.argmax(t, 1)`
    - **Dimension** : `torch.Size([batch_size])`
    - **Description** : Retourne l'indice de la classe réelle (celle avec la valeur 1 dans le vecteur one-hot) pour chaque exemple du lot. Comme avec `torch.argmax(y, 1)`, cela donne un tenseur 1D de taille `(batch_size)`, soit `(5)`.
13. `grad.sum(axis=0)`
    - **Dimension** : `torch.Size([10])`
    - **Description** : Somme des gradients par rapport aux classes de sortie, agrégée sur le lot. Ce tenseur 1D contient un gradient pour chaque classe de sortie, de taille `(10)`.

Chapitre 2

Shallow network

2.1 Implémentation

Voici l'implémentation de notre réseau de neurones `ShallowNetwork`. Nous avons utilisé une classe héritant de `torch.nn.Module`, nous définissons deux couches, une couche cachée et une couche de sortie. La couche d'entrée correspondant aux 28x28 pixels et la couche de sortie aux 10 classes possibles.

Listing 2.1 – Implémentation du ShallowNetwork

```
class ShallowNetwork(torch.nn.Module):
    def __init__(self, nb_of_neurons):
        super(ShallowNetwork, self).__init__()

        self.hidden_layer = torch.nn.Linear(28 * 28, nb_of_neurons
            )
        self.output_layer = torch.nn.Linear(nb_of_neurons, 10)

    def forward(self, x: torch.Tensor):
        x = self.hidden_layer(x)
        x = torch.nn.functional.relu(x)
        x = self.output_layer(x)
        return x
```

2.2 Optimisation du modèle

2.2.1 Étape 1 : Préparation du jeu d'entrainement, validation et test

Nous avons à disposition un jeu d'entrainement et un jeu de test composé respectivement de 63000 et 7000 images. Cependant, il nous manque un jeu de validation qui nous servira pour évaluer la performance des modèles avec des hyperparamètres différents et choisir le meilleur ensemble d'hyperparamètres.

Nous allons donc dédier 20% de nos données d'entrainement pour la validation. Pour cela, nous utiliserons la fonction `random_split` de `Pytorch` qui nous permet de diviser aléatoirement notre jeu de données en nouveaux sous-ensembles de données de taille spécifique.

```
train_subset, val_subset = torch.utils.data.random_split(
    training_dataset, [0.8, 0.2], generator=generator)
```

2.2.2 Étape 2 : Définir les hyperparamètres et plages de valeurs à explorer

Jusqu'à maintenant, notre modèle est défini à l'aide des hyperparamètres suivants :

- **batch_size** : Le nombre d'exemples utilisés pour chaque mise à jour des poids du modèle.
- **nb_epochs** : Le nombre de fois que l'ensemble de données complet sera parcouru pendant l'entraînement.
- η (ou *learning rate*) : Le taux d'apprentissage pour l'optimiseur, qui contrôle la taille des mises à jour des poids du modèle.
- **nombre_de_neurones** : Le nombre de neurones de la couche cachée. Ce nombre influe sur la complexité de notre réseau.

Nous voulons tester plusieurs configurations possibles afin d'évaluer le modèle le plus adapté à notre problème. Ainsi, voici les plages sélectionnées.

- **Taille des mini-batchs** : $[4, 8, 16, 32, 64, 128]$.
- **Nombre d'epochs** : $[10, 20, 50, 100]$.
- η **(Taux d'apprentissage)** : $[0.1, 0.01, 0.001, 0.0001]$.
- **Nombre de neurones** : $[10, 25, 50, 100]$.

Concernant les fonctions d'optimisation, nous partirons sur la fonction coût `Cross Entropy Loss` qui semble appropriée aux problèmes de classification multi-classes et la descente de gradient stochastique (`SGD`) pour l'optimisation des paramètres.

2.2.3 Étape 3 : Méthode d'optimisation des hyperparamètres

Les deux techniques souvent utilisées pour trouver les hyperparamètres du modèle offrant la meilleur performance sont la technique `Grid Search` et `Random Search`. Ces deux méthodes consistent à entrainer plusieurs modèles avec des hyperparamètres différents.

Recherche par grille (Grid Search) : teste de manière exhaustive toutes les combinaisons possibles d'hyperparamètres (définis sur une plage). Cette méthode est coûteuse mais efficace pour de petits espaces de recherche.

Recherche aléatoire (Random Search) : sélectionne des combinaisons aléatoires d'hyperparamètres dans l'espace défini. Elle est souvent plus efficace que la

recherche par grille, surtout quand le nombre de combinaisons est élevé.

Soucieux de la puissance de calcul de nos ordinateurs [1], notre approche consistait initialement à faire une recherche par grille hierarchique où nous évaluerons nos modèles avec des hyperparamètres définis sur une plage de données (ou plutôt un ensemble de données) progressivement restreinte. Afin de mieux cibler les hyperparamètres offrant les meilleures performances.

Finalement, nous avions eu le temps d'entraîner tous les modèles intialement prévus. En effet, afin de maximiser l'exploration des hyperparamètres, nous exécutons l'entraînement et l'évaluation de nos différents modèles sur nos deux machines respectives pour accélérer le processus de rechercher et augmenter le nombre de modèles entrainés. Nous enregistrons les résultats dans un fichier JSON à mesure que nous entraînons les modèles pour les analyser ultérieurement. Nous pourrons également par la suite les importer dans un DataFrame pour une analyse plus détaillée.

Une fois tout le processus automatisé à l'aide de fonctions dont l'implémentation est détaillée dans les fichiers fournis avec ce rapport, nous n'avions plus qu'à lancer nos programmes et patienter. À titre d'information, le temps d'exécution de l'entraînement et l'évaluation des 384 modèles est approximé à 12 heures.

2.3 Analyse et comparaison des modèles

À présent que nos modèles sont entrainés et évalués sur le jeu de validation, nous avons à notre disposition une multitude d'indicateurs permettant d'analyser la performance des modèles.

Pour chaque modèle, nous avons à disposition :

— Les hyperparamètres utilisés
— Les différentes valeurs de la perte d'entrainement (training loss).
— Les différentes valeurs de la perte de validation (validation loss).
— La justesse de notre modèle (final validation accuracy)

Le tableau suivant nous montre les hyperparamètres et les pareformances associés aux cinq modèles les plus preformants et les cinq modèles les moins perfor-

1. Machines équipées d'un processeur 13th Gen Intel(R) Core(TM) i7-1360P 2.20GHz ainsi qu'un processeur AMD Ryzen 5700U 1.80GHz

mants.

batch_size	nb_epochs	learning_rate	nb_neurons	final_validation_accuracy
8	50	0.100000	100	0.981111
16	50	0.100000	100	0.979603
8	100	0.100000	100	0.979444
16	100	0.100000	100	0.979127
16	20	0.100000	100	0.978413
...	...	...	...	...
64	10	0.000100	25	0.273333
128	20	0.000100	25	0.252698
64	20	0.000100	10	0.226746
128	50	0.000100	10	0.221349
128	10	0.000100	10	0.196349

TABLE 2.1 – Tableau récapitulatif des différents modèles et leur score final.

Nous pouvons identifier les modèles les plus performants et les comparer pour extraire des informations utiles. Nous pouvons de prime abord observer que nos modèles les plus performants atteignent un score de 98% d'exactitude. En observant, les hyperparamètres des modèles, nous pouvons faire quelques suppositions les concernant.

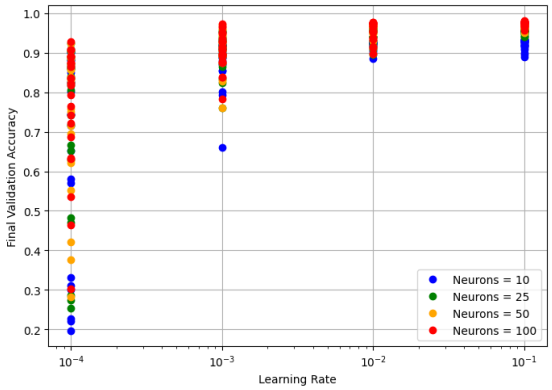
Premièrement, les modèles avec un score (exactitude) élevé sont paramétrés avec une taille de batch faible (8 et 16). Ensuite, on observe que le pas d'apprentissage adéquat est de 0.1. Enfin, plus le nombre de neurones et grand plus on obtient de bonnes performances. Par ailleurs, le nombre d'époques influence également l'apprentissage du modèle. Plus ce nombre augmente, plus notre modèle apprend et devient performant.

2.4 Analyse des hyperparamètres

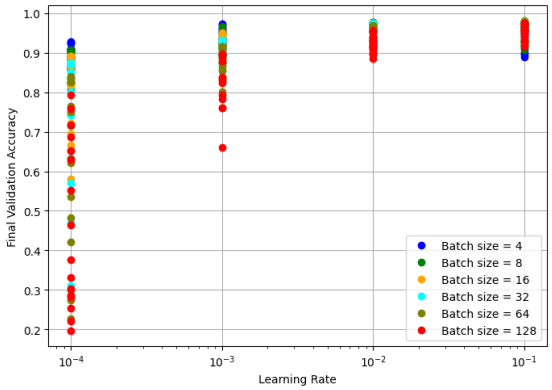
Dans cette section, nous étudierons plus en détails l'influence des hyperparamètres sur nos modèles. Pour ce faire, nous tenterons d'observer l'impact de la variation de chaque hyperparamètre à l'aide de notre table et de visualisation.

2.4.1 Taux d'apprentissage

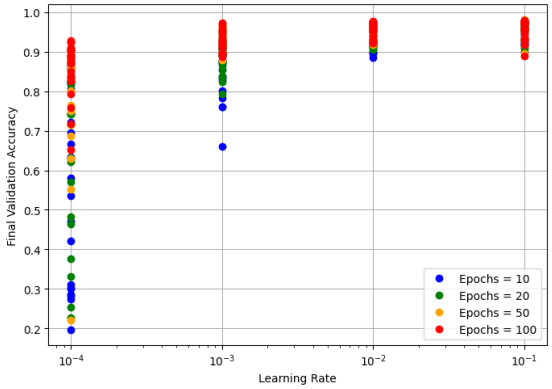
En observant de nouveau le tableau 2.1, nous remarquons que les modèles avec un taux d'apprentissage de 0.1 semblent être les plus performants, suivis de près par ceux avec un taux de 0.01. Les modèles avec un taux de 0.001 et 0.0001 sont les moins performants.



(a) Performance des modèles en fonction du learning rate et nombre de neurones



(b) Performance des modèles en fonction du learning rate et taille des lots



(c) Performance des modèles en fonction du learning rate et du nombre d'époques

FIGURE 2.1 – Comparaison des performances en fonction du learning rate et variation des hyperparamètres

Globalement, on remarque rapidement que les modèles les mieux entraînés concernent ceux avec un taux d'apprentissage fixé à 10^{-1}.

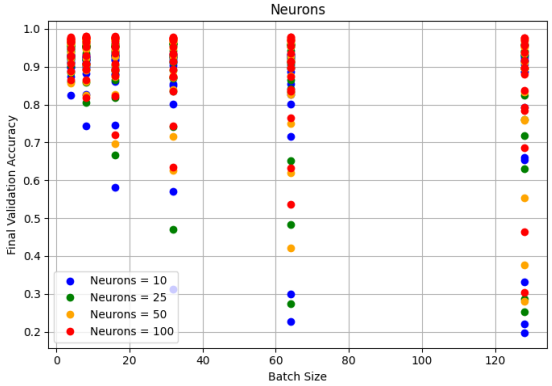
Par rapport au nombre de neurones : La précision finale de validation est fortement influencée par le taux d'apprentissage. Lorsque le taux d'apprentissage est très bas (10^{-4}), la précision varie beaucoup, mais elle est globalement plus basse pour les modèles avec un petit nombre de neurones (points bleus). Cela peut indiquer que ces modèles sous-apprennent. Lorsque le taux d'apprentissage atteint 10^{-3}, les performances s'améliorent pour les configurations neuronales. Cependant, au-delà (vers 10^{-2} et 10^{-1}), les performances tendent à se stabiliser. Donc, pour un taux d'apprentissage élevé, l'utilisation d'un nombre de neurones plus élevé (100) permet d'obtenir de meilleures performances.

Par rapport à la taille des batch : Le graphique montre également une influence de la performance par rapport au taux d'apprentissage. Les modèles avec un taux d'apprentissage de 10^{-4} semblent être les moins performants. Comme on peut le voir, les modèles avec une taille de batch de 8 et 16 semblent être les plus performants.

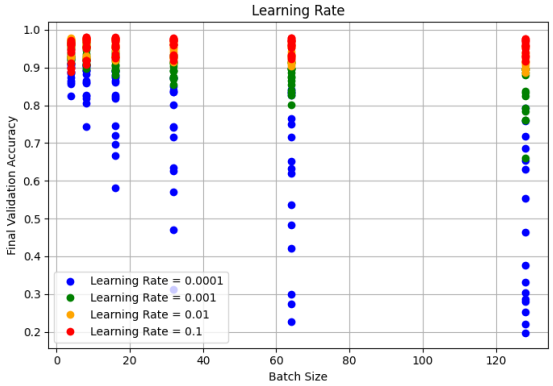
Par rapport au nombre d'epochs : Pour les taux d'apprentissage faibles (10^{-4}), la précision varie, mais semble plus basse pour les petits nombres d'époques (10). Lorsque le taux d'apprentissage augmente, la précision augmente également, mais elle semble se stabiliser pour des nombres d'époques plus élevées (50, 100). Ainsi, un nombre d'époques élevé combiné à un taux d'apprentissage élevé semble être également une bonne configuration, mais non suffisante.

2.4.2 Taille des mini-batchs

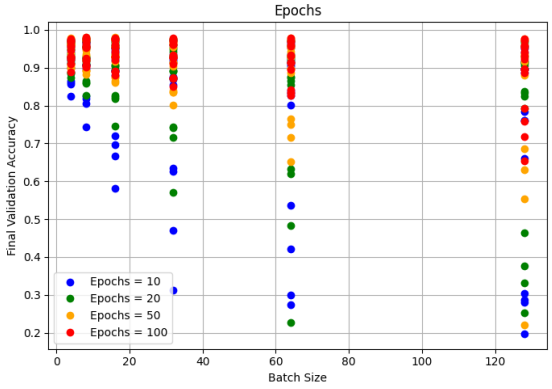
En reprenant le tableau, les modèles avec une taille de batch de 8 et 16 semblent être les plus performants, suivis par ceux avec une taille de 8. En revanche, les modèles avec une taille de 64 et 128 sont les moins performants. À partir de ces observations, nous pouvons supposer que dans notre cas, une taille de batch plus petite semble être plus efficace.



(a) Performance des modèles en fonction du la taille des lots et nombre de neurones



(b) Performance des modèles en fonction de la taille des lots et taux d'apprentissage



(c) Performance des modèles en fonction de la taille des lots et du nombre d'époques

FIGURE 2.2 – Comparaison des performances en fonction de la taille des lots et variation des hyperparamètres

Globalement, on remarque que les modèles entraînés par lots de 4 ou 8 ont généralement résultat correct ($> 80\%$). En revanche pour une taille de lot supérieure à 60, les résultats sont assez hétérogènes en fonction des autres hyperparamètres.

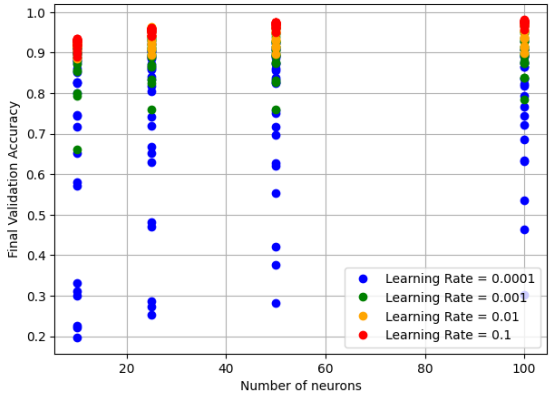
Par rapport au nombre de neurones : Les performances des modèles sont relativement bonnes pour les tailles de lots jusqu'à 32, où la majorité des points atteignent une précision proche de 1 (point rouge : nb de neurones = 100). Cependant, pour les tailles de lots plus élevées (64, 128), la précision diminue. Cela peut indiquer que les modèles avec une taille de lot plus élevée ont plus de mal à généraliser.

Par rapport au taux d'apprentissage : Pour des tailles de lots plus petites (8, 16), les taux d'apprentissage produisent de bonnes performances, surtout les taux élevés (10^{-2}, 10^{-1}). Cependant, pour des tailles de lots plus élevées (64, 128), les performances sont plus faibles.

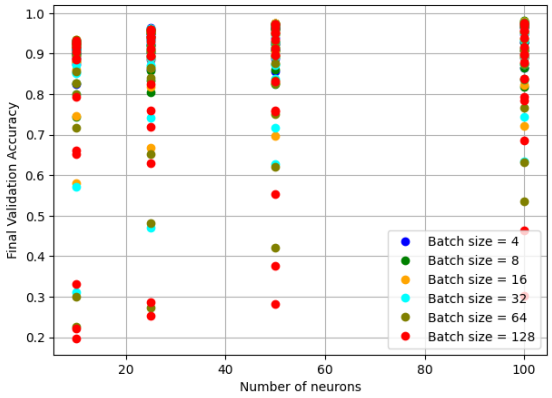
Par rapport au nombre d'epochs : Les tailles des lots entre 4 et 16 montrent des performances relativement bonnes pour les epochs plus élevées (50, 100). Cependant, avec une taille de lot plus grande, la précision devient imprévisible, ce qui pourrait indiquer une difficulté à généraliser avec un grand batch size.

2.4.3 Nombre de neurones

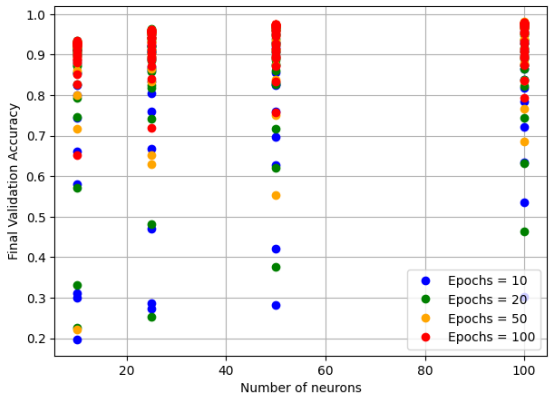
Les observations faites sur le tableau nous montrent que les modèles avec 50 et 100 neurones semblent être les plus performants. En revanche, les modèles avec 10 neurones ont tendance à être les moins performants. Nous pouvons supposer qu'augmenter le nombre de neurones peut mener à une plus grande efficacité sur notre problème. Néanmoins, plus le nombre de neurones est élevé, plus son apprentissage requiert de ressources et nous devons prendre en compte nos contraintes de calculs. Par ailleurs, un nombre de neurones trop élevé peut conduire à un sur-apprentissage.



(a) Performance des modèles en fonction du nombre de neurones et taux d'apprentissage



(b) Performance des modèles en fonction du nombre de neurones et taille des lots



(c) Performance des modèles en fonction du nombre de neurones et du nombre d'époques

FIGURE 2.3 – Comparaison des performances en fonction du nombre de neurones et variation des hyperparamètres

Par rapport au taux d'apprentissage : Un grand nombre de neurones semble être bénéfique pour les taux d'apprentissage plus élevés (10^{-2}, 10^{-1}). Cependant, les performances sont plus faibles pour les taux d'apprentissage plus faibles (10^{-4}, 10^{-3}).

Par rapport à la taille des batchs : Les performances sont relativement bonnes pour les tailles de lots plus petites (8, 16) et un nombre de neurones plus élevé. Cependant, lorsque la taille des batchs augmente, la précision diminue, surtout pour un petit nombre de neurones.

Par rapport à la taille d'epochs : Un nombre élevé d'époques (50 ou 100), couplé avec un grand nombre de neurones (100), donne de bons résultats, avec des précisions proches de 1. Cependant, un petit nombre d'époques (10) limite la performance, notamment avec un petit nombre de neurones. Les résultats montrent que plus le nombre de neurones est élevé, plus le nombre d'époques doit être important pour améliorer la performance.

Les hyperparamètres cités ci-dessus sont les plus importants à prendre en compte pour l'entraînement de notre modèle. Cependant, il est important de noter que d'autres hyperparamètres peuvent également influencer les performances du modèle. Nous avons par ailleurs tenté d'étudier l'impact du nombre d'époques sur les performances du modèle. Il en ressort que les modèles avec les meilleures performances sont ceux entraînés sur un plus grand nombre d'époques (100, voire 50). Cela peut s'expliquer par le fait que plus le modèle est entraîné, plus il a de chances de converger vers un minimum global.

2.4.4 Corrélation entre hyperparamètres

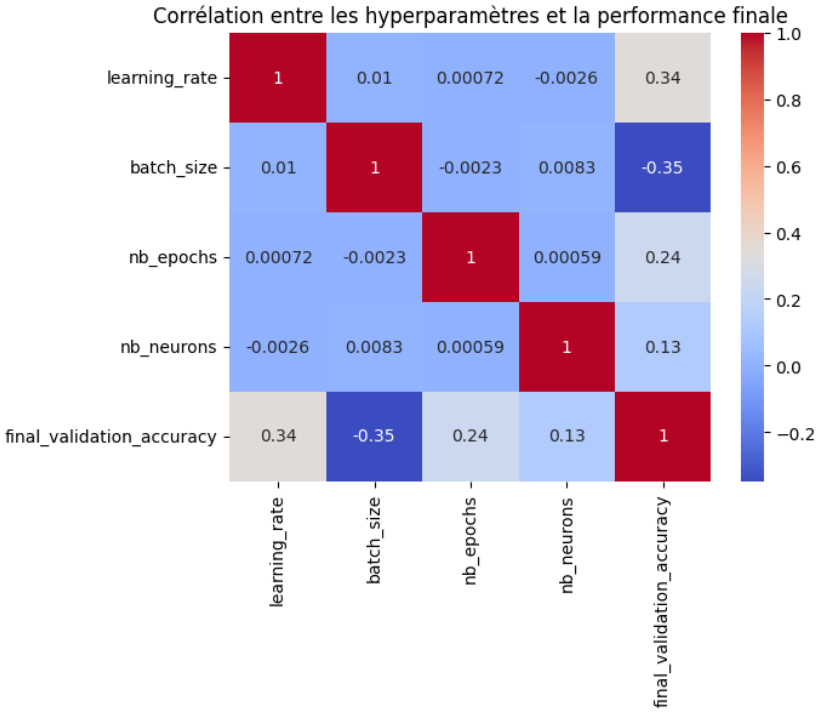


FIGURE 2.4 – Corrélation entre les hyperparamètres et la performance finale

Taux d'apprentissage (corrélation positive avec la précision de validation : 0,34) : Le taux d'apprentissage présente une corrélation positive notable avec la précision finale de validation. Cela suggère qu'à mesure que le taux d'apprentissage augmente (jusqu'à un certain point), la précision de validation tend à s'améliorer.

Taille des mini-batchs (corrélation négative avec la précision de validation : -0,35) : La taille du lot présente une corrélation modérément forte et négative avec la précision finale de validation, ce qui implique que la diminution de la taille des batchs augmente la précision de validation. Des tailles de lot plus petites pourraient donc conduire à de meilleures performances.

Nombre d'époques (corrélation positive avec la précision de validation : 0,24) : Impact positif modéré : Le nombre d'époques montre une corrélation modérément positive avec la précision de validation. Cela signifie qu'une augmentation du nombre d'époques peut entraîner une amélioration de la précision de

validation, bien que cet effet soit plus faible que pour le taux d'apprentissage. En effet, plus le nombre d'époques augmente, plus notre modèle peut apprendre sur les données et améliorer ses performances.

Nombre de neurones (corrélation positive avec la précision de validation : 0,13) : Le nombre de neurones présente une faible corrélation positive avec la précision finale de validation. L'augmentation du nombre de neurones n'a pas un impact très fort sur les performances du modèle, mais il existe tout de même une légère tendance positive.

2.5 Analyse de l'entrainement des modèles

Nous allons maintenant afficher la courbe d'apprentissage des modèles les plus performants afin d'observer leur apprentissage. Cette étape nous permettra également d'observer des éventuels surapprentissage.

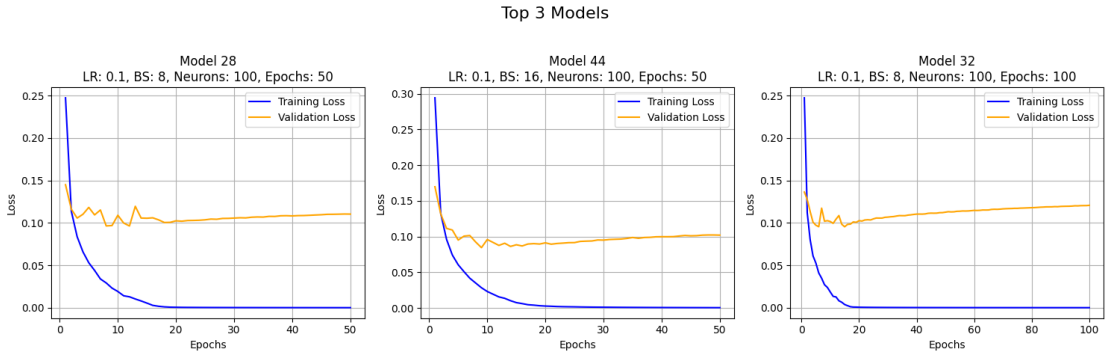


FIGURE 2.5 – Courbes de perte d'entrainement et de validation des trois meilleurs modèles

Nous observons sur les graphiques obtenus à partir des trois modèles que la courbe de perte d'entraînement décroît fortement jusque converger vers 0. Cependant, la courbe de perte de validation ne suit pas cette tendance et fini même par augmenter de nouveau. Nous sommes donc en présence d'un sur-apprentissage où nos modèles parviennent à prédire correctement nos données d'entraînement, au détriment d'une généralisation sur de nouvelles données. Une manière simple de contrer ce problème dans notre situation est d'adopter le *early stopping*, qui consiste simplement à interrompre l'entraînement du modèle lorsque la perte de validation augmente.

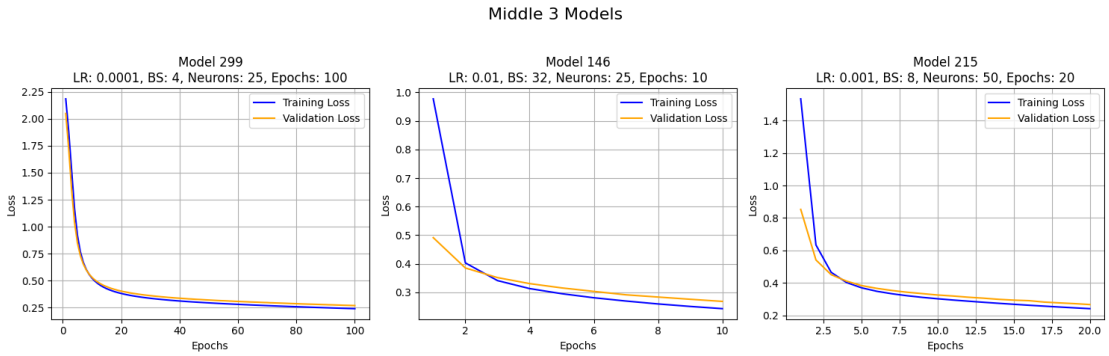


FIGURE 2.6 – Courbes de perte d'entrainement et de validation de trois modèles intérmédiaires

En prenant trois modèles figurant dans la médiane en terme de performance, nous observons que les courbes de perte d'entrainement et de validation sont similaires.

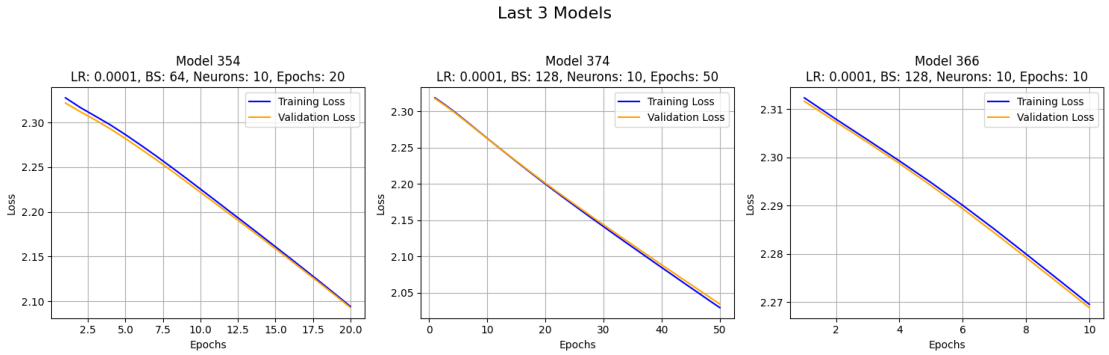


FIGURE 2.7 – Courbes de perte d'entrainement et de validation des cinq modèles les moins performants

Sur ces derniers graphiques correspondant aux trois modèles les moins performants, nous observons que les deux courbes décroient lentement et linéairement. L'apprentissage se termine avec une perte bien plus élevé que les autres modèles autant pour l'entraînement que la validation.

2.6 Sélection et évaluation sur le jeu de test final

Nous venons d'observer l'évolution de l'apprentissage de nos différents modèles. Dans notre cas, nous souhaitons le modèle qui généralise le mieux sur nos données d'entraînement.

learning_rate	batch_size	nb_epochs	nb_neurons	final_validation_accuracy
0.100000	8	50	100	0.981111

TABLE 2.2 – Tableau récapitulatif du meilleur modèle.

En appliquant ce modèle sur les données de test, nous obtenons un score final de : 98.06%

Nous utiliserons la même méthodologie pour étudier les parties suivantes.

Chapitre 3

Deep network

3.1 Méthodologie

Pour ce type de réseau de neurones, nous avons utilisé une architecture légèrement plus complexe avec davantage de couches cachées. Nous nous inspirions de l'architecture vue dans la partie précédente, ce qui signifie que chaque neurone d'une couche est connecté à tous les neurones de la couche précédente et de la couche suivante (Fully Connected Network).

Au niveau des fonctions d'activation, nous continuons d'utiliser la fonction ReLU (Rectified Linear Unit) qui permet l'ajout de non-linéarité et qui est largement employé dans les réseaux de neurones profonds.

Après avoir utilisé la même méthodologie pour la préparation des données et la définition de l'architecture, nous pouvons à présent, entamer l'entraînement de nos différents modèles. Nous utilisons le même principe de recherche par grille, à l'exception que les paramètres que nous faisons varier sont le nombre de neurones au sein de chaque couche du réseau. Par soucis d'optimisation du processus d'apprentissage, les autres hyperparamètres tels que la taille des lots, le nombre d'époques et le taux d'apprentissage sont fixés à partir des hyperparamètres identifiés comme optimaux pour notre problème lors de la partie précédente. Cela nous permet de réduire considérablement l'espace de possibilité des différents modèles que nous avons évalué à près de 12 000 modèles, ce qui est trop lourd pour la puissance de calcul à notre disposition. À noter qu'il nous sera toujours possible de peaufiner ces paramètres par la suite en adoptant une approche hiérarchique.

3.2 Implémentation

Nous avons délibérément choisi deux architectures de réseau de neurones. Une première avec deux couches cachées et une seconde avec trois couches cachées. Nous considérons qu'ajouter d'autres couches ne serait pas pertinent étant donné que nous obtenons déjà des résultats satisfaisants (>98% de justesse) avec un réseau peu profond.

Listing 3.1 – Implémentation du réseau de neurones profond (2 couches)

```
1 class DeepNeuralNetwork_2(torch.nn.Module):
2     def __init__(self, nb_neurons_1:int, nb_neurons_2:int):
3         super(DeepNeuralNetwork, self).__init__()
4 
5         self.hidden_layer_1 = torch.nn.Linear(28*28, nb_neurons_1)
6         self.hidden_layer_2 = torch.nn.Linear(nb_neurons_1,
              nb_neurons_2)
```

```
        self.output_layer = torch.nn.Linear(nb_neurons_2, 10)

    def forward(self,x:torch.Tensor):
        x = self.hidden_layer_1(x)
        x = torch.nn.functional.relu(x)
        x = self.hidden_layer_2(x)
        x = torch.nn.functional.relu(x)
        x = self.output_layer(x)
        return x
```

Listing 3.2 – Implémentation du réseau de neurones profond (3 couches)

```
class DeepNeuralNetwork(torch.nn.Module):
    def __init__(self, nb_neurons_1:int, nb_neurons_2:int,
        nb_neurons_3:int):
        super(DeepNeuralNetwork,self).__init__()

        self.hidden_layer_1 = torch.nn.Linear(28*28, nb_neurons_1)
        self.hidden_layer_2 = torch.nn.Linear(nb_neurons_1,
            nb_neurons_2)
        self.hidden_layer_3 = torch.nn.Linear(nb_neurons_2,
            nb_neurons_3)
        self.output_layer = torch.nn.Linear(nb_neurons_3, 10)

    def forward(self,x:torch.Tensor):
        x = self.hidden_layer_1(x)
        x = torch.nn.functional.relu(x)
        x = self.hidden_layer_2(x)
        x = torch.nn.functional.relu(x)
        x = self.hidden_layer_3(x)
        x = torch.nn.functional.relu(x)
        x = self.output_layer(x)
        return x
```

3.3 Analyse des résultats des modèles

3.3.1 Variante à deux couches

Une grande partie des tests effectués pour cette partie ont été réalisés sur une machine équipée d'une carte graphique dédiée [1]. En exécutant les calculs sur le GPU et en tirant profit de la parallélisation des opérations, nous avons observé un gain de vitesse significatif.

En analysant les résultats des entraînements, nous constatons qu'il est préférable d'utiliser un grand nombre de neurones dans la première couche cachée.

1. Carte graphique NVIDIA GeForce RTX 3060 Laptop.

De plus, le modèle le plus performant avec cette architecture semble obtenir de meilleurs résultats lorsque la deuxième couche cachée contient un nombre réduit de neurones.

LR	batch_size	nb_epochs	layer 1	layer 2	final_val_accuracy
0.01	32	25	2048	128	0.975159
0.01	32	25	2048	2048	0.974762
0.01	32	25	2048	1024	0.974524
0.01	32	25	1024	1024	0.974444
0.01	32	25	1024	20	0.974444

TABLE 3.1 – Résultats d'entraînement avec différents paramètres d'apprentissage pour deux couches cachées

3.3.2 Variante à trois couches

En observant les résultats des entraînements, nous remarquons que la justesse des modèles est meilleure lorsque le nombre de neurones dans la première et la dernière couche (nb_neurons_1 et nb_neurons_3) est élevé, tandis que le nombre de neurones dans la couche intermédiaire (nb_neurons_2) est plus bas. Une interprétation possible est que cela permet de capturer de nombreux détails en entrée, de les simplifier dans la couche intermédiaire, puis de les combiner efficacement avant la sortie.

LR	batch_size	nb_epochs	layer 1	layer 2	layer 3	final_val_acc
0.01	32	10	1024	10	1024	0.972143
0.01	32	10	1024	20	1024	0.971429
0.01	32	10	512	10	1024	0.971349
0.01	32	10	256	10	1024	0.970397
0.01	32	10	1024	10	512	0.970397

Pour améliorer davantage les performances des modèles, nous allons réentraîner les 5 meilleurs modèles identifiés initialement en augmentant le nombre d'époques à 50. L'augmentation des époques permettra à chaque modèle de continuer à s'améliorer sur les données d'entraînement, ce qui pourrait conduire à une meilleure convergence et donc augmenter la justesse globale. Cela permettra de voir si ces modèles peuvent atteindre un niveau de performance supérieur grâce à un entraî-

nement plus long.

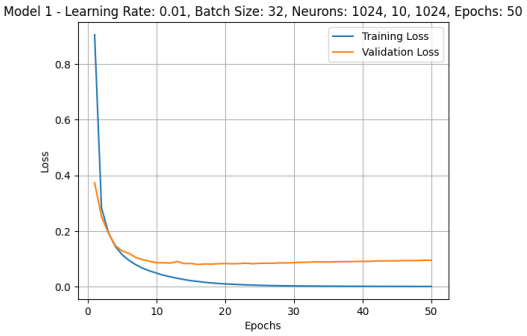
Voici le récapitualif des cinqs modèles les plus performants jusqu'à présent :

LR	batch_size	nb_epochs	layer 1	layer 2	layer 3	final_val_accuracy
0.01	32	50	1024	10	1024	0.979921
0.01	32	50	1024	20	1024	0.979841
0.01	32	50	1024	10	512	0.978810
0.01	32	50	512	10	1024	0.978333
0.01	32	50	256	10	1024	0.978333

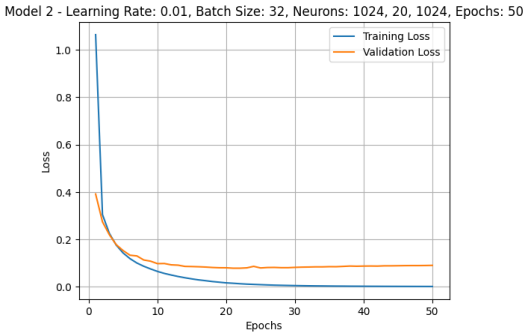
Contrairement à ce que nous attendions, nous n'observons pas d'amélioration majeure au niveau des performances.

3.4 Analyse de l'entraînement des modèles

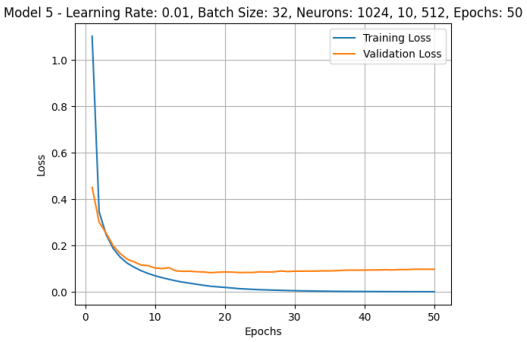
Tentons tout de même d'afficher les courbes de perte d'entraînement et de validation de nos modèles.



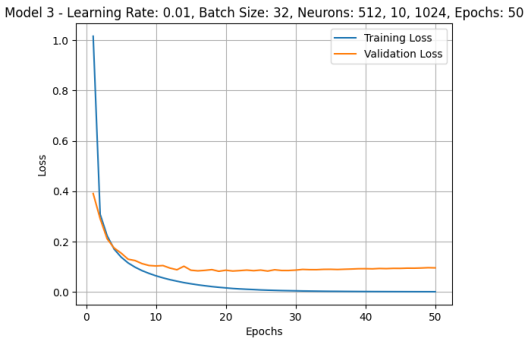
(a) Modèle 1024, 10, 1024



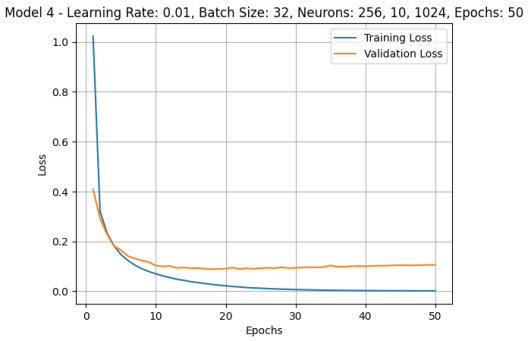
(b) Modèle 1024, 20, 1024



(c) Modèle 1024, 10, 512



(d) Modèle 512, 10, 512



(e) Modèle 256, 10, 1024

FIGURE 3.1 – Courbes de perte d'entraînement et de validation des cinq meilleurs modèles

On observe globalement sur nos courbes que la perte d'entraînement diminue jusqu'à atteindre un plateau proche de zéro, ce qui indique que le modèle apprend efficacement sur les données d'entraînement. En revanche, la perte de validation diminue également au début, mais finit par augmenter légèrement après un certain

nombre d'époques, suggérant un risque de surapprentissage. Dans ce cas particulier, le nombre d'époques a été augmenté à 50 au lieu de 10 pour améliorer la précision du modèle. Ainsi, bien que cela puisse légèrement dégrader la performance finale, il n'est pas nécessaire de réduire le nombre d'époques si le but principal est de maximiser la justesse.

3.5 Sélection et évaluation sur le jeu de test final

Après observation, voici les hyperparamètres retenus pour notre meilleur modèle.

LR	batch_size	nb_epochs	layer 1	layer 2	layer 3	final_val_accuracy
0.01	32	50	1024	10	1024	0.979921

En sélectionnant ce modèle et en lui passant notre jeu de test, nous obtenons une performance de 98.06%.

Chapitre 4

CNN

Dans cette partie, nous étudierons les réseaux de neurones convolutifs (CNN), mieux adaptés aux images. Nous avons décidé d'appliquer un fine tuning du réseau de neurones ResNet-18 appliqué à notre jeu de données MNIST.

4.1 Redimensionnement des images

Afin d'entraîner notre modèle sur nos données d'entraînement, nous devons redimensionner nos tenseurs en matrice 28x28 pour réformer l'image d'origine. Après avoir redimensionné chaque image, nous pouvons les afficher afin d'en avoir un aperçu.

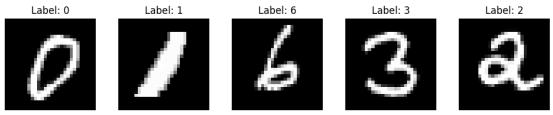


FIGURE 4.1 – Aperçu de notre jeu d'entrainement

4.2 Choix de l'architecture

À présent, il nous faut choisir un modèle pré-entraîné afin de pouvoir le *fine-tune* sur nos données. Nous nous sommes tournés vers l'architecture ResNet (Residual Networks) qui semble bien adaptée aux problèmes de classification d'image. Dans ce projet, nous utiliserons la variante ResNet18 qui correspond à la profondeur du réseau de neurones.

Une fois le modèle importé, nous avons ajusté la couche d'entrée et la couche de sortie pour correspondre à notre problème spécifique. En effet, comme les images du jeu de données MNIST sont des images en niveaux de gris de taille 28×28, tandis que le modèle ResNet-18 est conçu à l'origine pour traiter des images RGB de taille 224×224, plusieurs ajustements ont été nécessaires.

Premièrement, nous avons redimensionné les images MNIST de 28×28 à 224×224 en utilisant des techniques d'interpolation. Cela permet d'adapter la taille des images aux attentes du modèle sans compromettre les informations contenues dans les images.

Deuxièmement, comme les images MNIST sont en niveaux de gris (avec une seule couche), alors que le modèle ResNet-18 s'attend à trois canaux (RGB), nous avons modifié la première couche de convolution. Plutôt que d'accepter des images à trois canaux, la première couche de convolution a été ajustée pour accepter un

seul canal.

Enfin, comme nous le voyons dans le tableau suivant, la couche de sortie a également été modifiée pour produire 10 sorties, correspondant aux 10 classes de chiffres.

Layer (type)	**Output Shape**	**Param #**
Conv2d-1	[-1, 64, 14, 14]	3,136
BatchNorm2d-2	[-1, 64, 14, 14]	128
ReLU-3	[-1, 64, 14, 14]	0
MaxPool2d-4	[-1, 64, 7, 7]	0
Conv2d-5	[-1, 64, 7, 7]	36,864
BatchNorm2d-6	[-1, 64, 7, 7]	128
ReLU-7	[-1, 64, 7, 7]	0
Conv2d-8	[-1, 64, 7, 7]	36,864
BatchNorm2d-9	[-1, 64, 7, 7]	128
ReLU-10	[-1, 64, 7, 7]	0
BasicBlock-11	[-1, 64, 7, 7]	0
Conv2d-12	[-1, 64, 7, 7]	36,864
BatchNorm2d-13	[-1, 64, 7, 7]	128
ReLU-14	[-1, 64, 7, 7]	0
Conv2d-15	[-1, 64, 7, 7]	36,864
BatchNorm2d-16	[-1, 64, 7, 7]	128
...	...	...
Conv2d-51	[-1, 512, 1, 1]	1,179,648
BatchNorm2d-52	[-1, 512, 1, 1]	1,024
ReLU-53	[-1, 512, 1, 1]	0
Conv2d-54	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-55	[-1, 512, 1, 1]	1,024
Conv2d-56	[-1, 512, 1, 1]	131,072
BatchNorm2d-57	[-1, 512, 1, 1]	1,024
ReLU-58	[-1, 512, 1, 1]	0
BasicBlock-59	[-1, 512, 1, 1]	0
Conv2d-60	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-61	[-1, 512, 1, 1]	1,024
ReLU-62	[-1, 512, 1, 1]	0
Conv2d-63	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-64	[-1, 512, 1, 1]	1,024
ReLU-65	[-1, 512, 1, 1]	0
BasicBlock-66	[-1, 512, 1, 1]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Linear-68	[-1, 10]	5,130
Total params :		11,175,370
Trainable params :		11,175,370

TABLE 4.1 – Architecture ResNet-18

Le tableau montre les différentes couches composant le modèle : des couches de convolution, de pooling, ainsi que les fonctions d'activation (ReLU) et d'autres types, comme la normalisation par lot (Batch Normalization). Chaque couche a des dimensions de sortie spécifiques, ainsi qu'un certain nombre de paramètres.

À présent que notre modèle est défini, nous pouvons entamer l'entraînement.

4.3 Entraînement des modèles

Nous avons exploré trois approches pour l'apprentissage de notre architecture de réseau de neurones convolutif :

- **Architecture ResNet-18 avec réinitialisation complète des poids** : Tous les poids du modèle sont réinitialisés et entraînés à partir de zéro sur notre jeu de données.
- **Architecture ResNet-18 en utilisant des poids préentraînés sur `IMAGENET1K_V1` avec réentraînement complet** : Les poids du modèle préentraîné sur le jeu de données `ImageNet1K_V1` sont utilisés, mais tous les paramètres sont réentraînés sur notre jeu de données.
- **Architecture ResNet-18 en utilisant des poids préentraînés avec gel partiel des couches** : Les poids des couches sont initialisés à partir du modèle préentraîné sur `ImageNet1K_V1`, mais seules les deux dernières couches sont ajustées. Toutes les autres couches sont figées pour adapter le modèle à nos données d'entraînement. Cette approche utilise le transfert d'apprentissage pour tirer parti des connaissances acquises lors de la tâche de reconnaissance d'images sur le jeu de données `ImageNet1K_V1`.

Dans cette partie, nous avons décidé d'expérimenter la fonction d'optimisation `Adam` et d'utiliser `Cross Entropy Loss` comme fonction coût.

Avant d'entraîner notre modèle sur nos données d'entrainement, nous pouvons noter que chacune des approches nous fournit un score inférieur à 10% sur notre jeu de validation.

Pour chaque approche, nous entrainerons les modèles sur plusieurs époques et observerons laquelle nous permet d'obtenir le meilleur résultat.

4.3.1 Avec réinitialisation des poids

En réinitialisant les poids et entraînant le modèle sur 10 époques, nous obtenons le modèle suivant :

nb_epochs	LR	loss_func	optimizer	final_validation_accuracy
20	0.01	CrossEntropyLoss	Adam	0.9909

TABLE 4.2 – Récapitulatif - ResNet-18 (Approche 1)

Avec seulement 10 époques, nous obtenons un résultat très satisfaisant.

4.3.2 Avec réentrainement complet des poids entrainés sur ImageNet1K_V1

En utilisant les poids du modèle complet et en entrainant de nouveau les poids nous obtenons un score élevé, bien que légèrement inférieur au modèle précédent comme le montre le tableau suivant :

nb_epochs	LR	loss_func	optimizer	final_validation_accuracy
10	0.01	CrossEntropyLoss	Adam	0.9874

TABLE 4.3 – Récapitulatif - ResNet-18 (Approche 2)

4.3.3 Avec gel partiel des couches

Dans cette approche, seules les deux dernières couches du modèle sont réentraînées, tandis que les autres couches conservent les poids préentraînés sur `ImageNet1K_V1`. Cette méthode de transfert d'apprentissage permet de tirer parti des représentations déjà apprises pour la tâche de reconnaissance d'images, tout en adaptant à notre problème. Pour ce faire nous pouvons désactiver l'apprentissage des paramètres des couches cibles.

Listing 4.1 – Implémentation du gel de couche

```
for name, param in resnet18.named\_parameters():
  if not (name.startswith('layer4') or name.startswith('fc')):
        param.requires\_grad = False
```

Après avoir réentraîné uniquement les deux dernières couches sur 10 époques, nous obtenons les résultats suivants :

nb_epochs	LR	loss_func	optimizer	final_validation_accuracy
10	0.01	CrossEntropyLoss	Adam	0.9157

TABLE 4.4 – Récapitulatif - ResNet-18 (Approche 3)

Les résultats démontrent que cette approche est contre-productif car nous observons une perte de performance comparé aux deux autres approhes.

4.4 Évaluation et modèle final

En sélectionnant le modèle avec la meilleure évaluation sur notre jeu de validation, nous obtenons sur notre jeu de test un score de 99.24%.

Conclusion

Dans ce projet, nous avons testé différentes architectures de réseaux de neurones sur le jeu de données MNIST. Cela nous a permis d'observer l'impact des hyperparamètres et d'analyser les phénomènes de sous-apprentissage et de sur-apprentissage.

Bien que nous pouvions obtenir des résultats convenables avec une architecture simple, parmi les modèles évalués, les réseaux de neurones convolutifs (CNN) se sont avérés les plus performants. Ils ont montré une meilleure capacité à extraire les caractéristiques des images, ce qui a conduit à de meilleurs résultats en comparaison avec les architectures plus simples. Ajuster les hyperparamètres, comme le taux d'apprentissage et la taille des mini-batchs, s'est également révélé essentiel pour optimiser les performances.