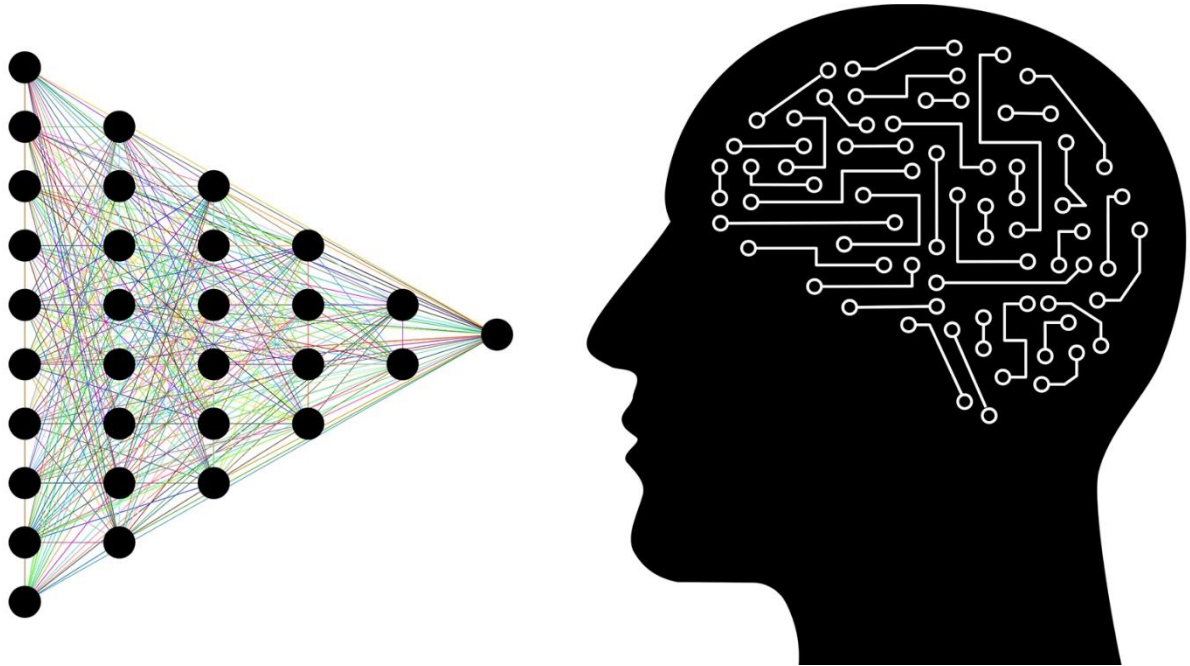


Rapport

TP Deep Learning



Partie 1 : réseaux de neurones


 **PyTorch**

Table des matières

Partie 1 : Perceptron	3
Partie 2 : Shallow Network	5
Méthode utilisée pour la recherche d'hyperparamètres :	5
Explication du code :	6
Influence de chaque hyperparamètres :	7
Influence de batch_size :	8
Influence de learning_rate :	9
Influence de hidden_size :	10
Influence de weight_init_range :	11
Influence de nb_epoch :	12
Évaluation des Combinaisons d'Hyperparamètres par Tests Imbriqués :	13
Partie 3 : Deep Network	22
Méthode utilisée pour la recherche d'hyperparamètres :	22
Explication du code :	22
Influence de chaque hyperparamètres :	23
Influence de batch_size :	24
Influence de learning_rate :	25
Influence de weight_init_range :	27
Influence du nombre de neurones dans deux couches cachées :	28
Influence du nombre de neurones dans trois couches cachées :	29
Influence du nombre de couches cachées :	31
Évaluation des Combinaisons d'Hyperparamètres par Tests Imbriqués :	32
Partie 4 : CNN	42
Explication du code	42
Influence de chaque hyperparamètres :	44
Influence de batch_size :	44
Influence de learning_rate :	45
Conclusion :	46
Annexes :	47
Organisation du répertoire de travail	47

Partie 1 : Perceptron

1. `data_train`

- **Taille** : `torch.Size([63000, 784])`
- **Description** : Contient les images d'entraînement, chaque image étant aplatie en un vecteur de 784 pixels (28x28 pixels). La taille totale est (`nombre_d_exemples_d_entrainement`, `nombre_de_pixels_par_image`), donc (63000, 784).

2. `label_train`

- **Taille** : `torch.Size([63000, 10])`
- **Description** : Contient les étiquettes associées aux images d'entraînement, représentées comme des vecteurs one-hot de 10 classes. La taille est (`nombre_d_exemples_d_entrainement`, `nombre_de_classes`), donc (63000, 10).

3. `data_test`

- **Taille** : `torch.Size([7000, 784])`
- **Description** : Contient les images de test, chaque image étant également aplatie en un vecteur de 784 pixels. La taille est (`nombre_d_exemples_de_test`, `nombre_de_pixels_par_image`), donc (7000, 784).

4. `label_test`

- **Taille** : `torch.Size([7000, 10])`
- **Description** : Contient les étiquettes associées aux images de test, représentées comme des vecteurs one-hot de 10 classes. La taille est (`nombre_d_exemples_de_test`, `nombre_de_classes`), donc (7000, 10).

5. `w`

- **Taille** : `torch.Size([784, 10])`
- **Description** : Matrice des poids du modèle. Chaque entrée représente la connexion entre un pixel d'entrée et une classe de sortie. La taille est (`nombre_de_pixels_par_image`, `nombre_de_classes`), donc (784, 10).

6. b

- **Taille** : torch.Size([1, 10])
- **Description** : Vecteur des biais du modèle, chaque biais étant associé à une classe de sortie. La taille est (1, nombre_de_classes), donc (1, 10).

7. y

- **Taille** : torch.Size([5, 10])
- **Description** : Résultat du produit matriciel entre x et w, plus b. La taille est (batch_size, nombre_de_classes). Si batch_size est 5, alors y.shape sera (5, 10) lors de l'entraînement.

8. t

- **Taille** : torch.Size([5, 10])
- **Description** : Étiquette vraie pour les exemples courants dans le lot. Représentée comme un vecteur one-hot. La taille est (batch_size, nombre_de_classes). Si batch_size est 5, alors t.shape sera (5, 10).

9. grad

- **Taille** : 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), donc (5, 10) pendant l'entraînement si batch_size est 5.

10. acc

- **Taille** : tensor([5970.])
- **Description** : Nombre total d'exemples correctement classifiés pendant l'évaluation du modèle. C'est une valeur scalaire accumulée pendant les tests.

11. nb_data_test

- **Taille** : 7000
- **Description** : Nombre total d'exemples dans l'ensemble de test.

12. acc/nb_data_test

- **Taille** : tensor([0.8533])
- **Description** : Précision du modèle sur l'ensemble de test, calculée comme la proportion d'exemples correctement classifiés par rapport au nombre total d'exemples de test. C'est un scalaire qui représente la précision du modèle.

Partie 2 : Shallow Network

Méthode utilisée pour la recherche d'hyperparamètres :

Pour identifier les meilleurs hyperparamètres pour notre modèle de perceptron multicouche, nous adoptons une approche méthodologique en deux étapes, afin d'optimiser la performance tout en limitant les coûts de calcul et en évitant les problèmes liés au surapprentissage.

Dans un premier temps, nous testons chaque hyperparamètre individuellement à l'aide de boucles simples. Nous faisons varier un seul hyperparamètre à la fois, comme la taille des mini-batches, le nombre d'époques, le taux d'apprentissage ou encore le nombre de neurones dans la couche cachée, tout en enregistrant les performances du modèle, avec une attention particulière à l'accuracy (le pourcentage de bonnes prédictions). Cette phase exploratoire nous permet de mieux comprendre l'impact de chaque hyperparamètre sur les performances du modèle, et de délimiter des plages de valeurs optimales. Par exemple, si une certaine taille de mini-batch permet d'améliorer l'accuracy de manière significative, nous la considérons comme une valeur prometteuse pour la suite. Cependant, si au cours de cette analyse nous détectons un phénomène de surapprentissage parmi les modèles ayant une bonne accuracy, nous excluons ces valeurs pour éviter de fausser les résultats dans les phases ultérieures. Les hyperparamètres non concernés par le surapprentissage sont alors conservés comme candidats potentiels.

Dans la deuxième étape, nous affinons notre recherche en explorant plus en détail les plages de valeurs identifiées lors de la première phase. Nous utilisons des boucles imbriquées pour tester diverses combinaisons d'hyperparamètres, mais cette fois, uniquement au sein des plages de valeurs retenues après l'évaluation individuelle. Cela nous permet d'explorer les interactions entre différents hyperparamètres tout en maintenant l'espace de recherche à une taille raisonnable. Cette approche graduelle est cruciale pour éviter de recourir à une stratégie brute force, qui consisterait à tester toutes les combinaisons possibles dès le début, une méthode inefficace compte tenu du grand nombre de combinaisons potentielles. Tester toutes ces configurations serait non seulement extrêmement coûteux en temps, mais pourrait également aboutir à des résultats biaisés si des modèles possèdent un accuracy trop bas, ou surapprennent les données d'entraînement.

Ainsi, en réduisant la plage des hyperparamètres à tester dès la première phase, nous minimisons la charge de calcul tout en garantissant une exploration ciblée des combinaisons les plus prometteuses.

Explication du code :

Classe PerceptronMulticouche

Dans le fichier `Shallow_network.py`, la classe `PerceptronMulticouche` implémente un réseau de neurones avec une seule couche cachée, en utilisant PyTorch. Voici un aperçu des principales composantes :

1. Constructeur `__init__`

- Initialise les couches et les paramètres du réseau, à savoir :
 - `input_size`, `hidden_size`, `output_size` : tailles des entrées, de la couche cachée et de la sortie.
 - `weight_init_range` : plage d'initialisation des poids.
 - `excel` : gestionnaire pour enregistrer les résultats.
- Détecte automatiquement si un GPU est disponible pour accélérer les calculs, en définissant `use_gpu` à `True` si cela est possible.
- Définit les couches du réseau avec `nn.Linear` et initialise les poids de manière uniforme à l'aide de `nn.init.uniform_`.

2. Méthode `forward`

- Gère la propagation avant dans le réseau :
 - Déplace les données sur le GPU si nécessaire.
 - Applique la fonction d'activation ReLU à la couche cachée.
 - Calcule la sortie finale à partir de la couche de sortie.

3. Méthode `train_and_evaluate`

- Entraîne le modèle et évalue sa performance en utilisant un jeu d'entraînement, de validation et de test, permettant ainsi de détecter le surapprentissage :
 - Utilise un optimiseur SGD et une fonction de perte `CrossEntropyLoss`.
 - Effectue l'entraînement sur plusieurs époques, tout en calculant la perte et l'accuracy pour les ensembles d'entraînement, de validation et de test.
 - Enregistre les résultats dans un fichier Excel pour une analyse ultérieure.
 - Si le modèle montre des signes de surapprentissage (par exemple, une perte de validation qui augmente alors que la perte d'entraînement diminue), cela sera noté afin d'ajuster les hyperparamètres ou la structure du modèle dans les phases ultérieures.

Cette structure permet une évaluation complète des performances du modèle tout en s'assurant qu'il ne s'adapte pas trop aux données d'entraînement, garantissant ainsi une meilleure capacité de généralisation sur des données non vues.

Influence de chaque hyperparamètres :

Comme mentionné précédemment, nous allons tester chaque hyperparamètre individuellement dans le fichier main.py. Voici les valeurs par défaut de chaque hyperparamètre :

- **Taille de lot (batch size)** : 5
- **Nombre d'époques (nb epochs)** : 10
- **Taux d'apprentissage (learning rate)** : 0.0001
- **Taille d'entrée (input size)** : 784 (28x28 pixels pour MNIST)
- **Taille de la couche cachée (hidden size)** : 128
- **Nombre de classes (output size)** : 10 (0 à 9 pour MNIST)
- **Plage d'initialisation des poids (weight init range)** : (-0.001, 0.001)

Nous avons également défini des listes de valeurs à tester pour chaque hyperparamètre :

- **Taille de lot (batch size)** : tab_batch_size = [2, 4, 8, 16, 32, 64, 128]
- **Nombre d'époques (nb epochs)** : tab_nb_epochs = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]
- **Taux d'apprentissage (learning rate)** : tab_learning_rate = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.25, 0.5, 0.6, 0.75, 1, 2]
- **Taille de la couche cachée (hidden size)** : tab_hidden_size = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
- **Plage d'initialisation des poids (weight init range)** : tab_weight_init_range = [(-0.0001, 0.0001), (-0.001, 0.001), (-0.01, 0.01), (-0.1, 0.1), (-1, 1), (-2, 2), (-5, 5), (-10, 10)]

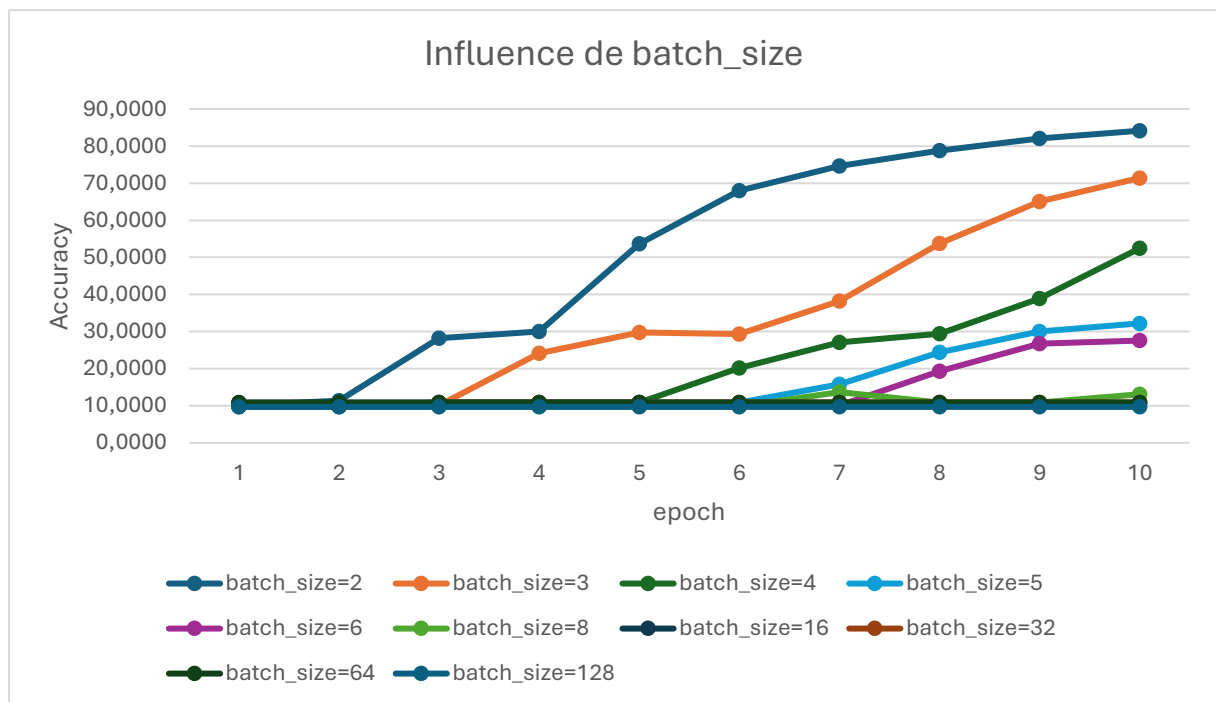
La fonction `definir_hyperparametres` est utilisée pour définir ces paramètres, en renvoyant un dictionnaire contenant les hyperparamètres nécessaires au modèle.

Dans le fichier main1.py, la fonction `evaluer_hyperparametre` évalue l'impact de chaque hyperparamètre sur les performances du modèle. À chaque itération, les résultats sont sauvegardés dans un fichier Excel. Cela permet de conserver les résultats même si le programme plante, évitant ainsi de devoir tout recommencer à partir de zéro.

En total, cette procédure a duré **1h47m56s**.

Influence de batch_size :

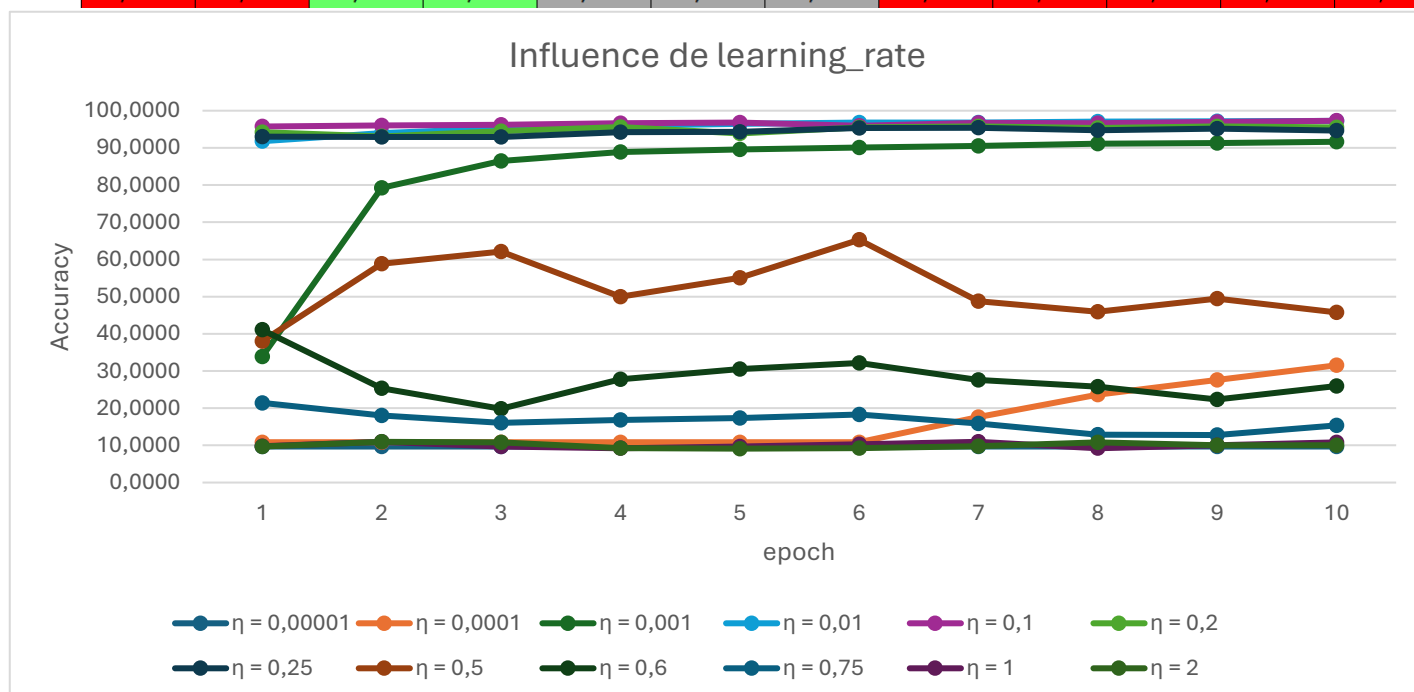
epoque	batch_size=2	batch_size=3	batch_size=4	batch_size=5	batch_size=6	batch_size=8	batch_size=16	batch_size=32	batch_size=64	batch_size=128
1	10,0429	10,0857	10,0857	9,6857	10,0429	9,7857	10,8429	9,7857	10,8429	9,6857
2	11,3571	10,0857	10,0857	9,6857	10,0429	9,7857	10,8429	9,7857	10,8429	9,6857
3	28,1571	10,0857	10,8429	10,8429	10,0429	9,7857	10,8429	10,8429	10,8429	9,6857
4	29,9714	24,1857	10,8429	10,8429	10,0429	9,7857	10,8429	10,8429	10,8429	9,6857
5	53,6714	29,7857	10,8429	10,8429	10,0429	9,7857	10,8429	10,8429	10,8429	9,6857
6	68,0000	29,3143	20,2000	10,8429	10,0429	9,7857	10,8429	10,8429	10,8429	9,6857
7	74,6714	38,2286	27,0429	15,7571	10,0429	13,6714	10,8429	10,8429	10,8429	9,6857
8	78,8143	53,7143	29,4429	24,3714	19,3286	10,8429	10,8429	10,8429	10,8429	9,6857
9	82,1000	65,0714	38,9429	30,0286	26,7714	10,8571	10,8429	10,8429	10,8429	9,6857
10	84,1429	71,4000	52,4429	32,2000	27,5571	13,1143	10,8429	10,8429	10,8429	9,6857



En examinant les résultats, les petites tailles de *batch_size* comme 2 et 3 montrent de meilleures performances, atteignant respectivement **84,14 %** et **71,40 %** de précision à la **10e époque**. Les tailles intermédiaires, comme 4 et 5, offrent des résultats corrects, mais commencent à montrer des signes de stabilisation, avec **52,44 %** pour *batch_size* = 4 et **32,20 %** pour *batch_size* = 5 à la même époque. Cependant, à partir de *batch_size* = 5, on observe une dégradation progressive des performances avec des tailles plus grandes, indiquant que le modèle a plus de mal à généraliser. Par exemple, pour *batch_size* = 6, la précision chute à **27,56 %**, et avec des tailles plus grandes (comme *batch_size* = 8 ou plus), les résultats stagnent autour de **10 %**. Cela montre que le modèle devient inefficace à mesure que le *batch_size* augmente. Pour nos tests finaux, nous nous concentrerons sur des *batch_size* allant de 2 à 5, où les performances sont les meilleures.

Influence de learning_rate :

numéro epo	$\eta = 0,00001$	$\eta = 0,0001$	$\eta = 0,001$	$\eta = 0,01$	$\eta = 0,1$	$\eta = 0,2$	$\eta = 0,25$	$\eta = 0,5$	$\eta = 0,6$	$\eta = 0,75$	$\eta = 1$	$\eta = 2$
1	9,6857	10,8429	33,8714	91,8000	95,7429	94,2000	93,0429	38,0143	41,0714	21,4286	9,7429	9,6857
2	9,6857	10,8429	79,2286	93,9286	96,0571	93,0714	92,9429	58,8571	25,3143	18,0286	10,9429	10,9429
3	9,6857	10,8429	86,4857	95,2571	96,2286	94,4857	92,9286	62,1286	19,8429	16,0857	9,6857	10,8429
4	9,6857	10,8429	88,8857	95,9286	96,6429	95,5714	94,1857	49,9714	27,8000	16,8000	9,2143	9,2143
5	9,6857	10,8429	89,5714	96,4429	96,7857	93,9143	94,2714	55,0286	30,4857	17,3571	9,7000	9,1143
6	9,6857	10,8429	90,1143	96,7857	95,9143	95,5286	95,3286	65,2714	32,1714	18,2571	10,2429	9,2143
7	9,6857	17,6143	90,5429	96,7714	96,6857	95,7714	95,4000	48,7571	27,6286	15,9143	10,9429	9,7857
8	9,6857	23,6571	91,1286	97,0286	96,6143	95,3143	94,7571	45,9143	25,8143	12,8714	9,2143	10,8429
9	9,6857	27,6143	91,3286	97,1143	96,9714	95,7571	95,2000	49,4286	22,3571	12,7714	10,0429	10,0429
10	9,6857	31,5286	91,6429	97,2571	97,2857	95,5429	94,6857	45,7857	25,9286	15,3571	10,8429	10,0429



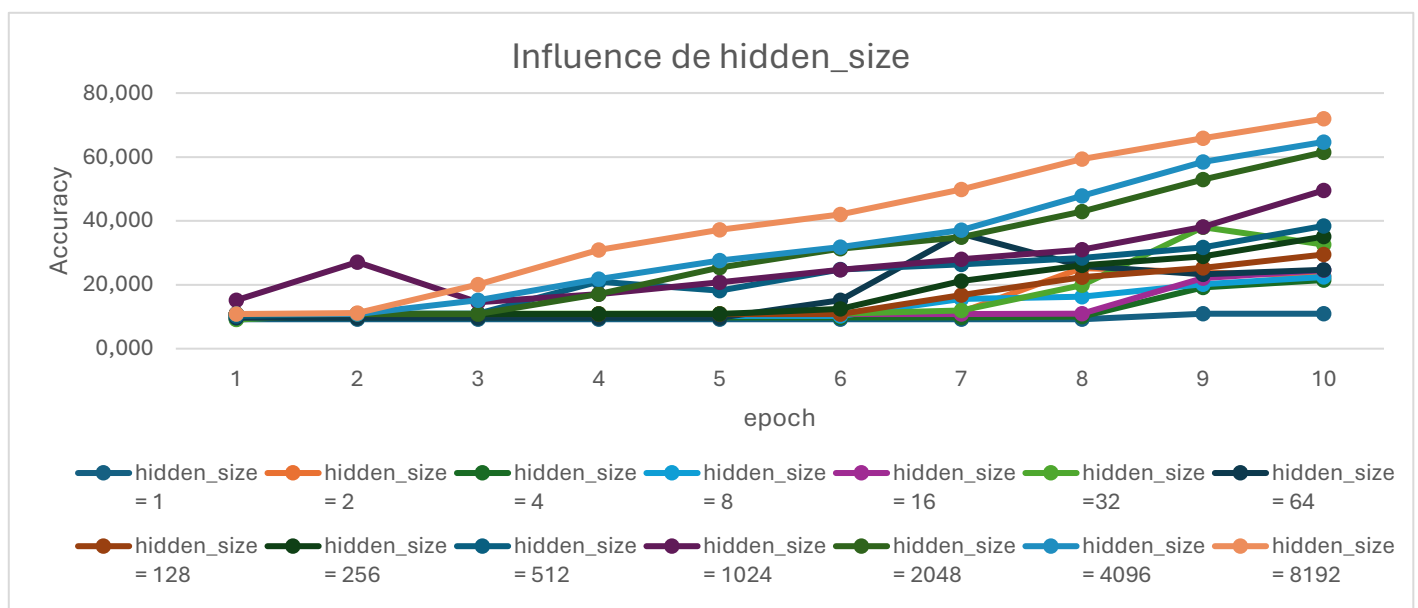
En examinant les résultats de la dixième époque, nous constatons que les performances du modèle varient considérablement en fonction du taux d'apprentissage (η). Pour un η de 0,00001, le pourcentage de bonnes réponses est de 9,69 %, tandis qu'à 0,5, il atteint 45,79 %. Les valeurs optimales semblent se situer autour de 0,01 à 0,1, où nous observons des pourcentages de bonnes réponses dépassant 97 %.

Bien que les taux d'apprentissage de 0,1, 0,2 et 0,25 affichent de bons résultats en termes d'accuracy, nous ne les prendrons pas en compte pour les futurs tests. En effet, l'analyse des pertes dans le fichier Excel qui stocke les résultats du modèle montre qu'en parcourant les époques, ces valeurs entraînent une perte de validation qui augmente alors que la perte d'entraînement diminue. Cette situation indique un risque de surapprentissage et nécessite une attention particulière pour garantir la robustesse du modèle.

Pour nos futurs tests, nous allons nous concentrer sur des valeurs de taux d'apprentissage comprises entre $\eta = 0,001$ et $\eta = 0,01$.

Influence de hidden_size :

numéro epoch	hidden_size = 1	hidden_size = 2	hidden_size = 4	hidden_size = 8	hidden_size = 16	hidden_size = 32	hidden_size = 64	hidden_size = 128	hidden_size = 256	hidden_size = 512	hidden_size = 1024	hidden_size = 2048	hidden_size = 4096	hidden_size = 8192
1	9,214	10,843	9,786	10,243	10,843	9,114	9,686	10,843	10,843	10,243	15,200	10,843	10,843	10,843
2	9,214	10,843	9,786	10,243	10,843	10,943	9,686	10,843	10,843	10,943	27,057	10,843	10,843	11,171
3	9,214	10,843	9,786	10,243	10,843	10,943	9,686	10,843	10,843	10,943	14,529	10,843	15,200	20,100
4	9,214	10,843	9,786	10,243	10,843	10,943	9,686	10,843	10,843	20,971	17,071	17,043	21,800	30,914
5	9,214	10,843	9,786	10,243	10,843	10,943	9,686	10,843	10,843	18,200	20,786	25,371	27,543	37,200
6	9,214	10,843	9,786	10,243	10,843	10,943	15,157	10,843	12,457	24,800	24,686	31,300	31,771	42,057
7	9,214	11,443	9,786	15,557	10,843	11,957	35,986	16,800	21,186	26,400	27,986	34,929	37,114	49,857
8	9,214	25,229	10,029	16,286	10,914	19,871	25,914	22,400	26,129	28,400	31,014	42,929	47,871	59,400
9	10,943	23,929	19,186	20,186	22,086	38,029	23,271	25,329	28,929	31,714	38,086	52,943	58,400	65,843
10	10,943	22,671	21,486	22,429	24,571	32,557	24,657	29,471	35,071	38,414	49,529	61,486	64,700	71,971

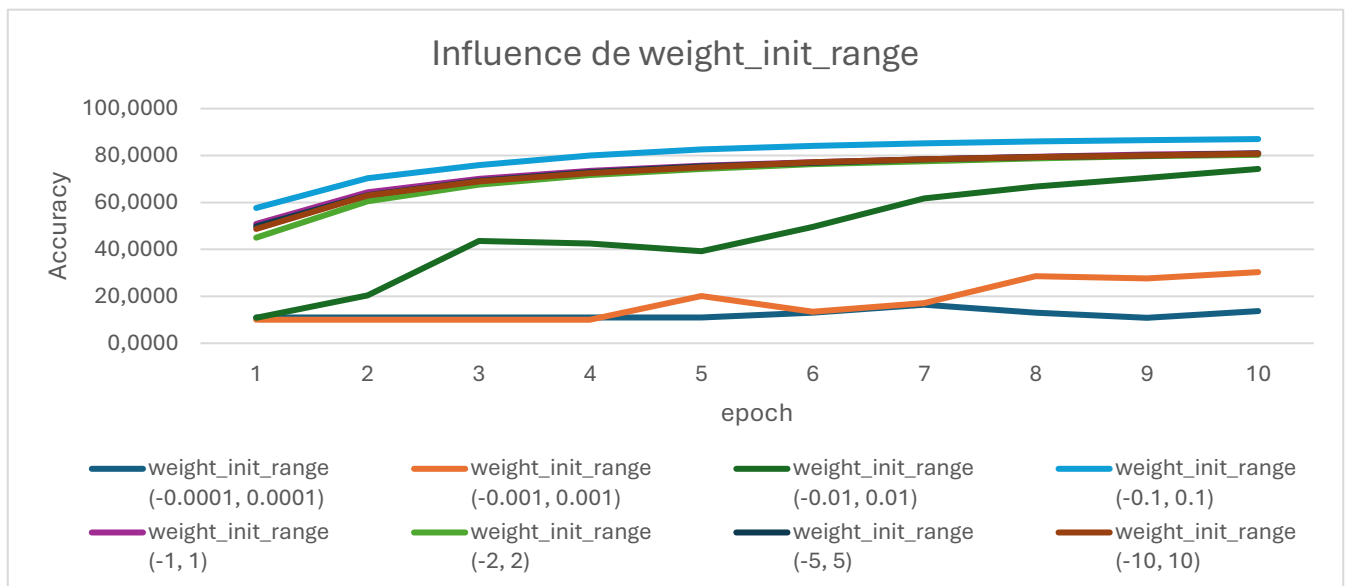


En analysant les résultats de la **dixième époque**, nous constatons que l'augmentation du nombre de neurones dans la couche cachée améliore progressivement les performances du modèle. Par exemple, avec un **hidden_size de 1**, le pourcentage de bonnes réponses est de **10,94 %**, tandis qu'avec un **hidden_size de 8192**, ce pourcentage atteint **71,97 %**. Cette tendance montre que plus le nombre de neurones dans la couche cachée est élevé, plus le modèle parvient à mieux apprendre et à généraliser. Pour des **hidden_size faibles** (comme **1, 2, 4**), la précision reste relativement basse (inférieure à **25 %**), ce qui suggère que le modèle manque de capacité pour capturer les caractéristiques complexes des données. Cependant, à partir de **hidden_size = 64**, on observe une amélioration notable, avec une précision qui passe à **24,66 %**, et continue d'augmenter de façon significative pour les tailles de couches cachées plus importantes. Au-delà de **hidden_size = 512**, l'augmentation des neurones montre des gains considérables, avec des performances atteignant **49,53 %** pour **1024**, et culminant à **71,97 %** pour **8192**. Cela indique que, bien que l'augmentation du nombre de neurones améliore les résultats, il existe un point où les gains deviennent plus progressifs.

Ainsi, pour nos tests finaux, nous allons nous concentrer sur des **hidden_size** élevés, notamment entre **128** et **4096** (8192 exclus car 4096 devrait être une valeur assez grande selon nous), afin de maximiser la capacité d'apprentissage du modèle et d'explorer l'impact de cette augmentation sur la performance globale.

Influence de weight_init_range :

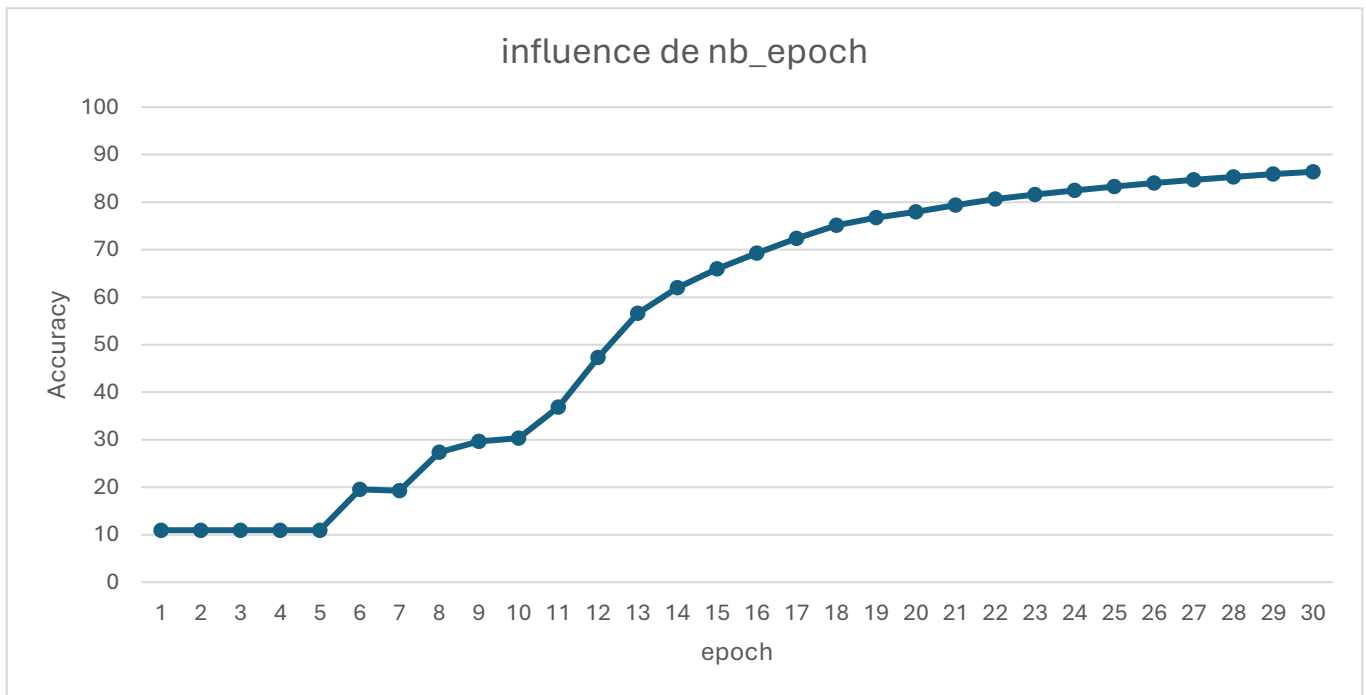
numéro epoch	weight_init_range (-0.0001, 0.0001)	weight_init_range (-0.001, 0.001)	weight_init_range (-0.01, 0.01)	weight_init_range (-0.1, 0.1)	weight_init_range (-1, 1)	weight_init_range (-2, 2)	weight_init_range (-5, 5)	weight_init_range (-10, 10)
1	10,9429	10,0429	10,8571	57,6571	50,8429	45,0000	49,8143	48,7143
2	10,9429	10,0429	20,4286	70,3143	64,3429	60,4429	63,1286	63,0143
3	10,9429	10,0429	43,5714	75,9714	70,0857	67,6286	69,2857	69,0286
4	10,9429	10,0429	42,5000	79,9571	73,4857	71,7143	72,9429	72,4714
5	10,9429	20,1857	39,2857	82,6000	75,7000	74,3143	75,3571	74,9571
6	13,0571	13,4571	49,5286	84,0571	77,1286	76,2714	76,9286	77,1143
7	16,4143	17,1857	61,6714	85,2000	78,3143	77,6143	78,4429	78,4000
8	13,0571	28,5429	66,8286	86,0714	79,5000	78,8143	79,3857	79,2714
9	10,8429	27,6857	70,4429	86,6286	80,3714	79,7000	80,0429	79,9714
10	13,7571	30,3143	74,2857	87,0143	80,9429	80,3000	80,9000	80,7714



En analysant les résultats de la dixième époque, nous observons que les performances du modèle varient selon la plage d'initialisation des poids. Par exemple, avec une plage de $(-0.0001, 0.0001)$, le pourcentage de bonnes réponses est faible, autour de **10,94 %**, tandis qu'une plage plus large comme $(-10, 10)$ permet d'atteindre **80,77 %**. Une plage intermédiaire comme $(-0.1, 0.1)$ produit l'une des meilleures performances avec **87,01 %** de bonnes réponses. Cela montre qu'à partir de $\text{weight_init_range} = (-1, 1)$, la valeur de accuracy se stabilise autour de 80%, ce qui améliore significativement la capacité du modèle à apprendre.

Donc pour nos future tests nous allons prendre des valeurs de weight_init_range allant de $(-0.001, 0.001)$ car c'est une valeur par défaut proposé au début du TP, jusqu'à $(-1, 1)$.

Influence de nb_epoch :



Bien que l'impact du nombre d'époques n'ait pas été explicitement demandé, nous avons souhaité explorer son influence.

Observations :

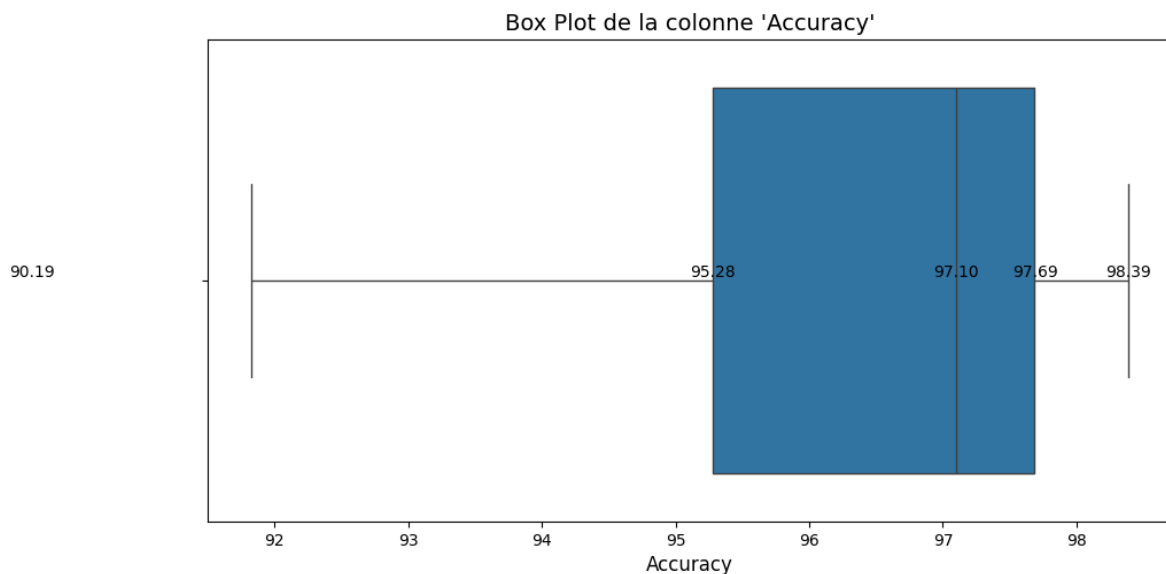
- L'accuracy commence à **10,94 %** et augmente progressivement pour atteindre **86,39 %** après 30 époques.
- Les gains d'accuracy deviennent significatifs à partir de la dixième époque, où l'on dépasse les **50 %**, avec une progression constante.
- Bien que l'accuracy continue d'augmenter, il est essentiel de surveiller les risques de surapprentissage si cette tendance persiste au-delà.

Conclusion : Nous avons décidé de fixer nb_epochs à **10** pour nos tests avec les boucles imbriquées. Ce seuil permet d'observer une nette amélioration sans entrer dans une phase de surapprentissage, optimisant ainsi l'évaluation des autres hyperparamètres.

Évaluation des Combinaisons d'Hyperparamètres par Tests Imbriqués :

Après l'évaluation individuelle des hyperparamètres, nous entrons dans une phase d'exploration plus ciblée. Dans cette deuxième partie, nous utilisons des boucles imbriquées pour tester des combinaisons de plages d'hyperparamètres identifiées comme prometteuses. Cette approche permet d'optimiser les performances du modèle sans surcharger nos ressources de calcul. En se concentrant sur des valeurs efficaces, nous maximisons l'accuracy tout en minimisant le risque de surapprentissage.

Nous avons exploré les hyperparamètres de notre modèle de perceptron multicouche en testant un total de 480 itérations, résultant de la combinaison des valeurs suivantes : quatre tailles de mini-batches (`tab_batch_size` = [2, 3, 4, 5]), cinq taux d'apprentissage (`tab_learning_rate` = [0.001, 0.0025, 0.005, 0.0075, 0.01]), six tailles de couches cachées (`tab_hidden_size` = [128, 256, 512, 1024, 2048, 4096]), et quatre plages d'initialisation des poids (`tab_weight_init_range` = [(-0.001, 0.001), (-0.01, 0.01), (-0.1, 0.1), (-1, 1)]). L'exécution de toutes ces combinaisons a duré un total de 18 heures, 52 minutes et 10 secondes.



Les résultats obtenus lors de l'optimisation des hyperparamètres de notre modèle de perceptron multicouche s'avèrent plutôt prometteurs. En effet, avec une moyenne de 96,33 % de bonnes réponses, nous observons des performances solides. L'écart type de 1,73 indique une variabilité plutôt basse dans les résultats, mais cela reste acceptable dans le contexte de l'apprentissage automatique, où des fluctuations peuvent se produire en fonction des hyperparamètres choisis.

De plus, les mesures de position, telles que le premier quartile (Q1) à 95,28%, la médiane à 97,10 % et le troisième quartile (Q3) à 97,69 %, renforcent cette impression positive. Ces valeurs suggèrent que la majorité des itérations ont produit des performances élevées, avec une concentration notable de résultats au-dessus de 93 %.

Pour faciliter l'analyse de ces données et leur interprétation, nous avons également importé le fichier Excel contenant les résultats dans une base de données PHPMyAdmin. Cela nous permet d'explorer plus facilement les performances et de réaliser des analyses plus approfondies sur les résultats obtenus.

Voici la requête que nous avons utilisée pour identifier les modèles de perceptron multicouche qui ne présentent pas de surapprentissage et qui maximisent la précision (accuracy). Cette requête SQL, exécutée dans notre base de données, nous permet de filtrer et d'analyser les combinaisons d'hyperparamètres prometteuses après 10 époques d'entraînement, tout en évitant le surapprentissage :

```
INSERT INTO shallow_network_no_of (batch_size, learning_rate, hidden_size,
weight_init_range, training_loss, validation_loss, test_loss, accuracy)
WITH EpochComparison AS (
    SELECT
        batch_size, learning_rate, hidden_size, weight_init_range,
        numero_epoch, training_loss, validation_loss, test_loss, accuracy,
        LAG(training_loss) OVER (PARTITION BY batch_size, learning_rate,
hidden_size, weight_init_range ORDER BY numero_epoch) AS prev_training_loss,
        LAG(validation_loss) OVER (PARTITION BY batch_size, learning_rate,
hidden_size, weight_init_range ORDER BY numero_epoch) AS prev_validation_loss,
        CASE
            WHEN LAG(training_loss) OVER (PARTITION BY batch_size,
learning_rate, hidden_size, weight_init_range ORDER BY numero_epoch) IS NOT
NULL
            AND LAG(validation_loss) OVER (PARTITION BY batch_size,
learning_rate, hidden_size, weight_init_range ORDER BY numero_epoch) IS NOT
NULL
            AND training_loss < LAG(training_loss) OVER (PARTITION BY
batch_size, learning_rate, hidden_size, weight_init_range ORDER BY
numero_epoch)
            AND validation_loss > LAG(validation_loss) OVER (PARTITION BY
batch_size, learning_rate, hidden_size, weight_init_range ORDER BY
numero_epoch)
            THEN 1
            ELSE 0
        END AS overfitting_flag
    FROM
        shallow_network_combinaison
),
OverfittingDetection AS (
    SELECT
        batch_size, learning_rate, hidden_size, weight_init_range,
        numero_epoch, training_loss, validation_loss, test_loss, accuracy,
        SUM(overfitting_flag) OVER (PARTITION BY batch_size, learning_rate,
hidden_size, weight_init_range ORDER BY numero_epoch ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS cumulative_overfitting
    FROM
```

```
EpochComparison
)
SELECT
    batch_size,
    learning_rate,
    hidden_size,
    weight_init_range,
    training_loss,
    validation_loss,
    test_loss,
    accuracy
FROM (
    SELECT
        batch_size,
        learning_rate,
        hidden_size,
        weight_init_range,
        training_loss,
        validation_loss,
        test_loss,
        accuracy,
        CASE
            WHEN cumulative_overfitting > 0 THEN TRUE
            ELSE FALSE
        END AS overfitting
    FROM
        OverfittingDetection
    WHERE
        numero_epoch = 10
) AS final_result
WHERE
    overfitting = FALSE
ORDER BY
    accuracy DESC;
```

Cette requête utilise des fonctions de fenêtre pour détecter le surapprentissage en comparant la perte d'entraînement et la perte de validation entre les époques successives. Les modèles qui n'ont montré aucun signe de surapprentissage après 10 époques sont ensuite triés par leur précision afin de mettre en avant les plus performants. Grâce à cette requête, nous avons pu isoler des combinaisons d'hyperparamètres qui maximisent les performances du modèle tout en minimisant le risque de surapprentissage.

Après avoir utilisé la requête précédente pour identifier les modèles qui ne présentent pas de surapprentissage, nous avons modifié la requête pour obtenir une vue d'ensemble du nombre

de modèles qui sont en surapprentissage (via `SELECT COUNT(*)`). En comptant les modèles, nous avons constaté que sur les 480 modèles testés, 331 ne présentent pas de surapprentissage, tandis que 149 montrent des signes de surapprentissage.

Pour stocker les résultats des modèles qui ne sont **pas** en surapprentissage, nous avons utilisé une requête SQL pour insérer ces résultats dans une nouvelle table appelée *shallow_network_no_of*.

Dans cette nouvelle table qui contient uniquement les modèles qui n'ont pas subi de surapprentissage. Les statistiques suivantes ont été obtenues pour ces modèles :

- **Valeur minimum de précision (accuracy) : 90.1857**
- **Moyenne de précision : 96.04**
- **Valeur maximum de précision : 98.38**

Ces résultats indiquent que les modèles sans surapprentissage présentent des performances robustes, avec une précision moyenne élevée et un écart notable entre les valeurs minimales et maximales.

Pour mieux comprendre l'impact des hyperparamètres sur les performances de nos modèles, nous avons exécuté plusieurs requêtes SQL afin d'évaluer le pourcentage de modèles dépassant la moyenne de précision en fonction de différents hyperparamètres. Voici les résultats obtenus :

La requête SQL suivante a été utilisée pour évaluer le pourcentage de modèles avec différents `batch_size` dépassant la moyenne de précision :

```
SELECT
    batch_size,
    COUNT(*) * 100.0 / (
        SELECT COUNT(*)
        FROM shallow_network_no_of
        WHERE
            accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
    ) AS percentage_above_avg
FROM
    shallow_network_no_of
WHERE
    accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
GROUP BY
    batch_size
ORDER BY percentage_above_avg DESC;
```


<u>batch_size</u>	<u>percentage above avg 1</u>
4	27.22772
5	26.23762
3	25.24752
2	21.28713

L'analyse des résultats révèle que le batch_size de 4 présente le pourcentage le plus élevé (27.23 %) d'itérations ayant dépassé la moyenne de précision, suivi par 5, 3 et 2.

Pour évaluer l'impact du learning_rate, la requête suivante a été exécutée :

```
SELECT
  learning_rate,
  COUNT(*) * 100.0 / (
    SELECT COUNT(*)
    FROM shallow_network_no_of
    WHERE
      accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
  ) AS percentage_above_avg
FROM
  shallow_network_no_of
WHERE
  accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
GROUP BY
  learning_rate
ORDER BY percentage_above_avg DESC;
```

<u>learning_rate</u>	<u>percentage above avg 1</u>
0.005	30.69307
0.0025	23.26733
0.0075	22.27723
0.01	19.30693
0.001	4.45545

Les résultats montrent qu'un learning_rate de 0.005 donne le meilleur pourcentage (30.69 %) d'itérations dépassant la moyenne de précision.

La requête SQL utilisée pour évaluer l'impact de la taille de la couche cachée est la suivante :

```
SELECT
  hidden_size,
  COUNT(*) * 100.0 / (
    SELECT COUNT(*)
    FROM shallow_network_no_of
    WHERE
      accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
  ) AS percentage_above_avg
FROM
  shallow_network_no_of
WHERE
  accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
GROUP BY
  hidden_size
ORDER BY percentage_above_avg DESC;
```

<u>hidden size</u>	<u>percentage above avg 1</u>
1024	18.81188
512	18.31683
2048	16.33663
4096	16.33663
256	15.34653
128	14.85149

Les résultats indiquent que la taille de la couche cachée de 1024 a le meilleur pourcentage d'itérations dépassant la moyenne de précision.

Enfin, la requête pour évaluer l'impact de la plage d'initialisation des poids a été la suivante :

```
SELECT
  weight_init_range,
  COUNT(*) * 100.0 / (
    SELECT COUNT(*)
    FROM shallow_network_no_of
    WHERE
      accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
  ) AS percentage_above_avg
FROM
  shallow_network_no_of
WHERE
  accuracy > (SELECT AVG(accuracy) FROM shallow_network_no_of)
GROUP BY
  weight_init_range
ORDER BY percentage_above_avg DESC;
```

<u>weight init range</u>	<u>percentage above avg 1</u>
(-0,1, 0,1)	35.14851
(-0,01, 0,01)	33.16832
(-0,001, 0,001)	29.70297
(-1, 1)	1.98020

L'analyse montre que la plage d'initialisation des poids de (-0.1, 0.1) a le pourcentage le plus élevé (35.15 %) d'itérations dépassant la moyenne de précision.

En exécutant la requête SQL suivante avec les meilleurs hyperparamètres théoriques

```
SELECT * FROM shallow_network_no_of
WHERE batch_size = 4
AND learning_rate = 0.005
AND hidden_size = 1024
AND weight_init_range = "(-0,1,0,1);
```

nous constatons qu'elle ne renvoie aucun résultat. Cela indique qu'avec cette combinaison précise d'hyperparamètres, il y a sur-apprentissage, empêchant le modèle d'obtenir de bonnes performances. Cette combinaison, bien qu'ayant potentiellement des paramètres prometteurs, ne fonctionne pas bien dans ce contexte.

Pour trouver la meilleure combinaison d'hyperparamètres sans sur-apprentissage, nous avons exécuté la requête suivante pour classer les modèles en fonction de leur précision :

```
SELECT * FROM shallow_network_no_of
ORDER BY accuracy DESC
LIMIT 10;
```

<u>batch_size</u>	<u>learning_rate</u>	<u>hidden_size</u>	<u>weight_init_range</u>	<u>training_loss</u>	<u>validation_loss</u>	<u>test_loss</u>	<u>accuracy_1</u>
2	0.01	2048	(-0,01, 0,01)	0.0135914	0.0702664	0.0561175	98.3857
2	0.01	1024	(-0,001, 0,001)	0.0171214	0.0715717	0.0613217	98.2571
3	0.01	4096	(-0,01, 0,01)	0.0241938	0.0626876	0.0584222	98.1857
2	0.005	1024	(-0,1, 0,1)	0.0233742	0.0662467	0.0650532	98.1857
4	0.0075	2048	(-0,1, 0,1)	0.0240373	0.0719485	0.0666183	98.1429
2	0.0075	4096	(-0,01, 0,01)	0.0207878	0.0639063	0.0585818	98.1286
2	0.0075	128	(-0,01, 0,01)	0.0345852	0.0781257	0.0695063	98.1
4	0.0075	4096	(-0,1, 0,1)	0.016278	0.0694052	0.0641613	98.1
4	0.01	1024	(-0,1, 0,1)	0.0219657	0.0768322	0.0662268	98.1
4	0.01	4096	(-0,1, 0,1)	0.0107123	0.0663018	0.0640308	98.0857

Ces résultats montrent que le modèle avec un batch_size de 2, un learning_rate de 0.01, une taille de couche cachée de 2048, et une plage d'initialisation des poids de (-0,01, 0,01) a obtenu la meilleure précision, atteignant **98.38 %**. Les autres combinaisons proches, comme batch_size de 2 avec learning_rate de 0.01 et hidden_size de 1024, ont également bien performé.

Partie 3 : Deep Network

Méthode utilisée pour la recherche d'hyperparamètres :

Pour l'optimisation des hyperparamètres dans les réseaux de neurones profonds, nous suivons la même approche méthodologique que celle déjà décrite pour les *shallow networks*. Cette méthode se déroule en deux étapes : d'abord, nous testons chaque hyperparamètre individuellement, puis nous explorons les combinaisons entre ces hyperparamètres.

L'objectif reste d'identifier les plages de valeurs optimales pour chaque paramètre (taille des mini-batches, nombre d'époques, taux d'apprentissage, caractéristiques des couches-cachées ...) et de minimiser les risques de surapprentissage. Une fois les plages prometteuses définies, nous affinons la recherche en combinant ces hyperparamètres afin d'obtenir les meilleures performances possibles, tout en limitant les coûts de calcul et en préservant la capacité de généralisation du modèle

Explication du code :

Classe DeepNetwork

Dans le fichier `Deep_Network.py`, la classe `DeepNetwork` implémente un réseau de neurones avec plusieurs couches cachées en utilisant PyTorch. Nous avons conservé la même structure de code que pour le *shallow network*, en apportant quelques modifications nécessaires pour gérer les réseaux profonds.

- **Nombre de Couches Cachées :**

Le *shallow network* contient une seule couche cachée, tandis que le *deep network* est conçu avec plusieurs couches cachées. Cela permet au *deep network* de capturer des caractéristiques plus complexes des données d'entrée, augmentant ainsi sa capacité d'apprentissage.

- **Structure des Couches :**

Dans le *deep network*, la définition des couches utilise `nn.ModuleList`, permettant d'ajouter dynamiquement plusieurs couches cachées, ce qui n'était pas nécessaire dans le *shallow network* où une seule couche suffisait.

- **Initialisation des Poids :**

Bien que les deux réseaux utilisent une initialisation uniforme des poids, le *deep network* inclut des couches supplémentaires dont les poids doivent être initialisés individuellement, contrairement au *shallow network*.

- **Vérification des Couches Cachées :**

Le *deep network* inclut une validation pour s'assurer qu'il y a au moins deux couches cachées valides, renforçant la structure du réseau et évitant les configurations non valides.

- **Propagation Avant :**

La méthode forward du *deep network* applique la fonction d'activation ReLU à chaque couche cachée, permettant ainsi de mieux gérer la non-linéarité à travers plusieurs couches, par rapport à une seule application dans le *shallow network*.

- **Entraînement et Évaluation :**

Bien que la méthode `train_and_evaluate` soit similaire, le *deep network* doit gérer plusieurs couches, ce qui peut influencer le calcul de la perte et de l'accuracy, et offrir une complexité accrue dans l'entraînement.

Influence de chaque hyperparamètres :

Comme mentionné précédemment, nous allons tester chaque hyperparamètre individuellement dans le fichier `main3.py`, de la même manière que dans `main1.py`. Voici les valeurs par défaut de chaque hyperparamètre pour le *deep network* :

- **Taille de lot (batch size) :** 5
- **Taux d'apprentissage (learning rate) :** 0.001
- **Taille des couches cachées (hidden size) :** [16, 16] (deux couches cachées par défaut)
- **Plage d'initialisation des poids (weight init range) :** (-0.01, 0.01)

Nous avons également défini des listes de valeurs à tester pour chaque hyperparamètre :

- **Taille de lot (batch size) :** `tab_batch_size` = [2, 4, 8, 16, 32, 64, 128]
- **Taux d'apprentissage (learning rate) :** `tab_learning_rate` = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.25, 0.5, 0.6, 0.75, 1, 2]
- **Taille des couches cachées (hidden size) :**
 - *Pour rappel, nous allons tester avec deux couches ayant des valeurs initiales de (16, 16).*
 - *Nous ferons ensuite varier la profondeur de chaque couche en utilisant des puissances de 2 allant de 8 à 256, ce qui nous donne 36 possibilités.*
 - *Enfin, nous ferons la même chose avec trois couches cachées, puis quatre, et enfin, nous examinerons l'influence exercée par le nombre de couche caché.*

- **Plage d'initialisation des poids (weight init range) :** `tab_weight_init_range = [(-0.0001, 0.0001), (-0.001, 0.001), (-0.01, 0.01), (-0.1, 0.1), (-1, 1), (-2, 2), (-5, 5), (-10, 10)]`

Comme nous pouvons le voir, nous testons hyperparamètres similaires lors de l'exécution du fichier `main1.py`. La différence notable se situe au niveau des tailles des couches cachées. Pour le deep network, nous testons différentes configurations de couches cachées (deux couches par défaut, mais avec des variantes allant jusqu'à plusieurs couches). Pour les tests, nous ajustons non seulement la taille des couches, mais aussi leur nombre, permettant d'explorer l'impact de réseaux plus profonds sur les performances du modèle.

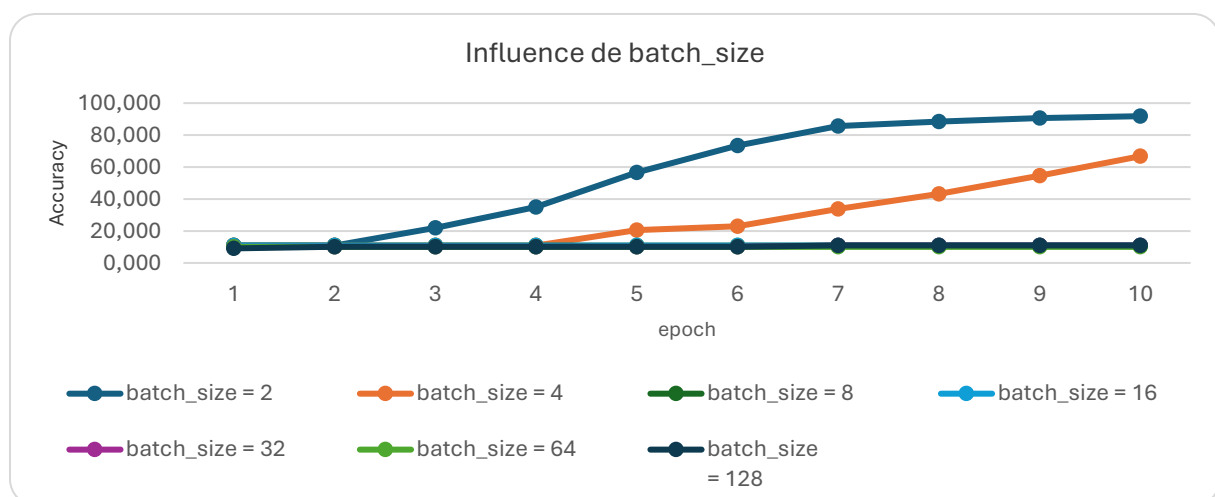
Tout comme dans `main1.py`, les résultats sont sauvegardés dans un fichier Excel à chaque itération pour éviter de perdre les données si le programme plante.

Durée des tests d'influence des hyperparamètres : 15 heures, 40 minutes et 10 secondes.

Une grande partie de cette durée est dû au fait que les tests sur les différentes couches cachées nous ont pris beaucoup de temps.

Influence de `batch_size` :

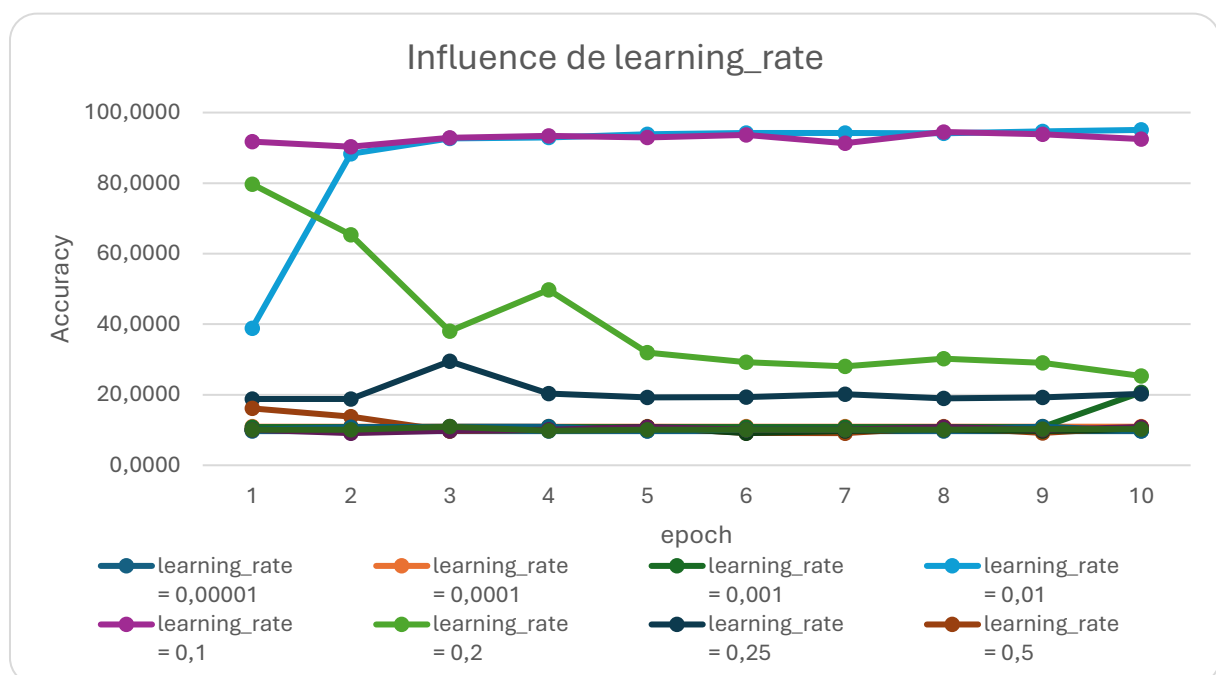
numéro epoch	<code>batch_size</code> = 2	<code>batch_size</code> = 4	<code>batch_size</code> = 8	<code>batch_size</code> = 16	<code>batch_size</code> = 32	<code>batch_size</code> = 64	<code>batch_size</code> = 128
1	10,843	10,843	10,843	10,943	10,086	10,043	9,114
2	10,843	10,843	10,843	10,843	10,086	10,043	10,043
3	22,043	10,843	10,843	10,843	10,086	10,043	10,043
4	35	10,843	10,843	10,843	10,086	10,043	10,043
5	56,529	20,486	10,843	10,843	10,086	10,043	10,043
6	73,329	22,914	10,843	10,843	10,086	10,043	10,043
7	85,714	33,829	10,843	10,843	10,843	10,043	10,943
8	88,529	43,186	10,843	10,843	10,843	10,043	10,943
9	90,629	54,657	10,843	10,843	10,843	10,043	10,943
10	91,814	66,771	10,843	10,843	10,843	10,043	10,943



En examinant les résultats obtenus pour le réseau profond, les petites tailles de batch, comme `batch_size = 2`, montrent des performances nettement supérieures, atteignant 91,81 % de précision à la 10e époque, tandis que les tailles intermédiaires, comme `batch_size = 4`, offrent une progression régulière, culminant à 66,77 %. Cependant, à partir de `batch_size = 8`, les performances stagnent rapidement autour de 10,84 %, et cela se maintient pour des tailles plus grandes, comme `batch_size = 16, 32, 64, et 128`, avec des résultats avoisinant les 10 % tout au long de l'entraînement. Ces résultats sont très similaires à ceux observés dans le réseau peu profond, où des petites tailles de batch permettaient également une meilleure capacité d'apprentissage, tandis que les batchs plus grands compromettaient la généralisation du modèle et entraînaient une stagnation des performances. Pour les tests finaux, nous allons donc nous en tenir à des valeurs prometteuses comme `batch_size = [2, 3, 4, 5]`, comme pour le réseau peu profond, où ces tailles ont montré les meilleures performances.

Influence de learning_rate :

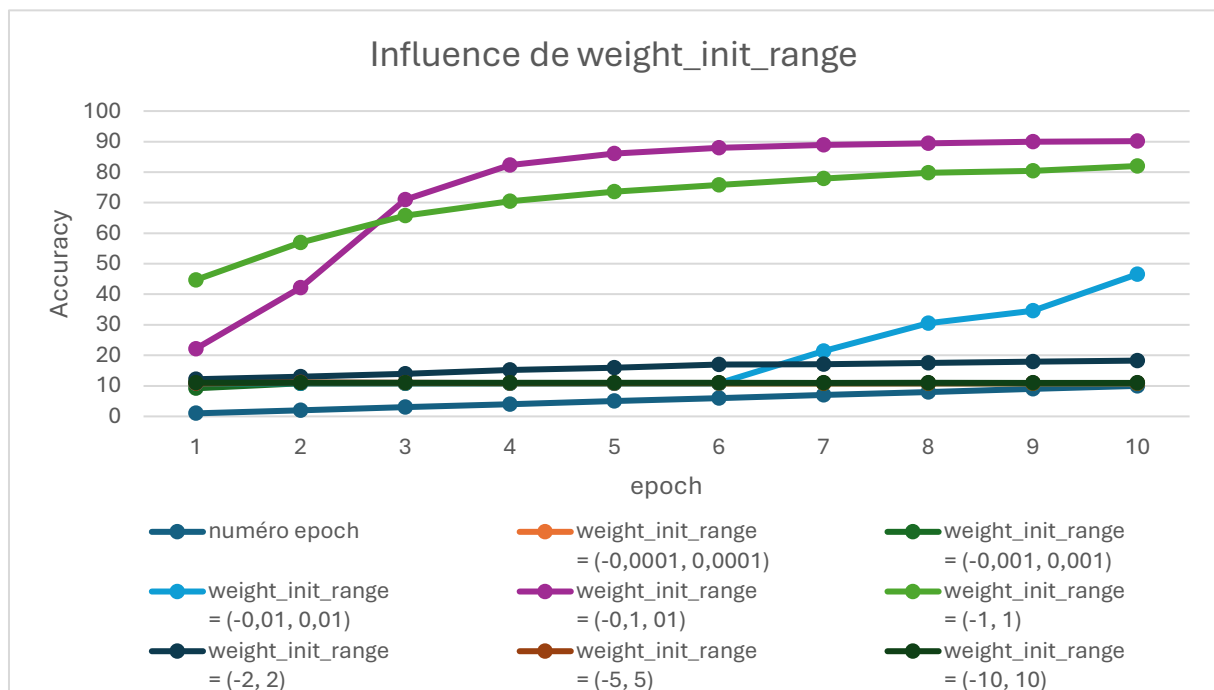
numéro epoch	learning_rate = 0,00001	learning_rate = 0,0001	learning_rate = 0,001	learning_rate = 0,01	learning_rate = 0,1	learning_rate = 0,2	learning_rate = 0,25	learning_rate = 0,5	learning_rate = 0,6	learning_rate = 0,75	learning_rate = 1	learning_rate = 2
1	9,6857	10,9429	10,8429	38,8571	91,7714	79,7143	18,8286	16,1286	10,0429	10,0429	10,0429	10,0429
2	9,6857	10,9429	10,8429	88,3	90,3286	65,3714	18,8	13,8429	10,2429	10,8429	9,1143	10,0429
3	9,6857	10,9429	10,8429	92,7	92,8143	38,0714	29,4714	9,6857	10,0429	10,9429	9,7857	10,9429
4	9,6857	10,9429	10,8429	92,9429	93,4143	49,7286	20,3571	10,0857	10,0429	10,9429	10,0429	9,7857
5	9,6857	10,9429	10,8429	93,8143	92,9429	31,9857	19,2857	10,8429	10,8429	9,7857	10,8429	10,0429
6	9,6857	10,9429	10,8429	94,2143	93,6857	29,2143	19,3	9,2143	9,1143	10,0429	10,0429	10,0857
7	9,6857	10,9429	10,8429	94,2571	91,2714	28,0429	20,1143	9,1143	9,6857	10,0429	10,0857	10,0429
8	9,6857	10,9429	10,8429	94,1571	94,5	30,2429	18,9714	10,8429	10,0857	10,0429	10,8429	10,0857
9	9,6857	10,9429	10,8429	94,6571	93,8571	29,0857	19,2857	9,2143	9,6857	10,9429	10,0429	10,2429
10	9,6857	10,9429	20,6714	95,0857	92,5	25,3286	20,2	10,8429	10,0429	9,7857	10,8429	10,2429



Lors de l'examen des résultats de la dixième époque, nous constatons que les performances du réseau profond varient considérablement selon le taux d'apprentissage (η). Entre $\eta = 0,00001$ et $\eta = 0,001$, la précision finale de la 10e époque varie de 9,68 % à 20,67 %. Un grand écart se produit à $\eta = 0,2$, où la précision atteint 25,33 %. Les valeurs optimales se situent autour de 0,01, où nous observons une précision de 95,08 %. Bien que des taux d'apprentissage comme 0,1 affichent également de bonnes performances avec 92,5 % de bonnes réponses, l'analyse des pertes dans le fichier Excel "deep_network.xlsx", feuille "learning_rate", montre que la perte d'entraînement diminue constamment tandis que la perte de validation fluctue. Cela indique un risque de surapprentissage et suggère une mauvaise généralisation. Pour nos tests finaux, nous allons donc nous concentrer sur un taux d'apprentissage de $\eta = 0,01$, car il équilibre bien les performances et la robustesse du modèle. Il est à noter que les résultats en termes de précision sont similaires à ceux observés avec le réseau peu profond (shallow network), qui atteint également de bons niveaux de précision.

Influence de weight_init_range :

numéro epoch	weight_init_range = (-0,0001, 0,0001)	weight_init_range = (-0,001, 0,001)	weight_init_range = (-0,01, 0,01)	weight_init_range = (-0,1, 0,1)	weight_init_range = (-1, 1)	weight_init_range = (-2, 2)	weight_init_range = (-5, 5)	weight_init_range = (-10, 10)
1	10,843	9,214	10,843	22,171	44,714	12,171	10,786	11,014
2	10,843	10,843	10,843	42,143	56,886	13,043	11,143	11
3	10,843	10,843	10,843	70,986	65,686	13,929	11	11,014
4	10,843	10,843	10,843	82,343	70,5	15,157	10,886	10,986
5	10,843	10,843	10,843	86,029	73,557	15,986	10,886	10,986
6	10,843	10,843	10,843	87,971	75,757	17,029	10,886	10,986
7	10,843	10,843	21,357	88,871	77,886	17,1	10,929	10,971
8	10,843	10,843	30,5	89,471	79,771	17,471	10,943	10,986
9	10,843	10,843	34,557	89,957	80,371	17,886	10,943	10,971
10	10,843	10,843	46,557	90,143	82	18,271	10,943	10,971



En analysant les résultats de la dixième époque pour le réseau profond, nous constatons que les performances du modèle varient selon la plage d'initialisation des poids. Par exemple, avec une plage de $(-0,0001, 0,0001)$, le pourcentage de bonnes réponses est très faible, autour de 10,84 %. Cependant, en élargissant la plage à $(-0,1, 0,1)$ et $(-1, 1)$, nous observons une amélioration significative, atteignant respectivement 90,14 % et 82 %.

Contrairement au réseau peu profond, qui a montré de bonnes performances avec plusieurs plages d'initialisation, le réseau profond présente des résultats optimaux uniquement avec les plages de $(-0,1, 0,1)$ et $(-1, 1)$. Cela indique que le modèle profond est plus restrictif en termes de plages d'initialisation des poids pour atteindre des résultats satisfaisants. Pour nos tests finaux, nous allons donc nous concentrer sur les plages d'initialisation des poids suivantes : $[(-0,001, 0,001), (-0,01, 0,01), (-0,1, 0,1), (-1, 1)]$.

Influence du nombre de neurones dans deux couches cachées :

C1/C2	8	16	32	64	128	256
8	<u>10,843</u>	21,771	25,271	29,214	38,8	79,343
16	20,929	<u>47,514</u>	25,129	43,8	68,1	73,886
32	25,157	49,786	<u>64,1</u>	48,47143	74,6	85,043
64	23,957	50,571	58,529	<u>62,8</u>	84,457	85,729
128	20,729	34,814	83,4	84,243	<u>84,9</u>	87,629
256	65,7	84,457	86,943	87,743	88,4	<u>89,129</u>

En examinant les résultats obtenus pour le modèle de réseau de neurones avec deux couches cachées, nous pouvons tirer plusieurs conclusions en fonction du nombre de neurones dans chaque couche :

1. **Nombre de neurones trop bas dans les deux couches** : Lorsque le nombre de neurones est limité (par exemple, 8 neurones dans chaque couche), la précision du modèle reste relativement faible, avec des valeurs atteignant seulement 10,84 % à 21,77 %. Cela indique que le modèle n'a pas suffisamment de capacité pour apprendre les représentations complexes des données.
2. **Augmentation du nombre de neurones dans la première couche** : En augmentant le nombre de neurones dans la première couche (par exemple, en passant de 8 à 256 neurones), nous observons une amélioration significative des performances. La précision atteint 79,34 % avec 256 neurones dans la première couche. Cela suggère que la première couche joue un rôle crucial dans l'extraction des caractéristiques des données.
3. **Augmentation du nombre de neurones dans la deuxième couche** : Lorsque nous augmentons le nombre de neurones dans la deuxième couche tout en maintenant le nombre de neurones dans la première couche, la précision varie également. Par exemple, avec 32 neurones dans la première couche et 256 neurones dans la deuxième, la précision peut atteindre jusqu'à 85,04 %. Cela indique que la deuxième couche contribue également à la complexité du modèle, lui permettant d'apprendre des représentations plus riches.
4. **Augmentation du nombre de neurones dans chaque couche** : Lorsque le nombre de neurones est augmenté dans chaque couche simultanément, les performances s'améliorent de manière significative. Par exemple, avec 256 neurones dans les deux couches, la précision atteint 89,13 %. Cela souligne l'importance d'avoir une

architecture suffisamment complexe pour capturer les variations des données d'entrée.

En conclusion, ces résultats montrent que l'augmentation du nombre de neurones dans l'une des couches, ou mieux encore, dans les deux, permet d'améliorer les performances du modèle. Cependant, il est essentiel de ne pas avoir un écart trop important entre le nombre de neurones dans les deux couches.

Influence du nombre de neurones dans trois couches cachées :

Pour les tests avec trois couches cachées, nous avons effectué des combinaisons de nombre de neurones avec 16, 32, 64, 128, 256 et 512 dans chaque couche. Après avoir réalisé ces tests, nous nous sommes rendu compte que tous nos résultats avaient une valeur d'accuracy de 10,84 %. Ces tests se sont hélas avérés inutiles, entraînant des heures gaspillées inutilement.

C'est alors que, dans la partie de l'étude du Shallow network, nous nous sommes souvenus que nous avions également étudié l'influence du nombre d'époques par rapport à l'accuracy. Bien qu'il ne nous ait pas été demandé d'étudier cet hyperparamètre, nous voulions quand même connaître son effet. Nous avons remarqué que plus le nombre d'époques était grand, plus cela permettait d'avoir une accuracy élevée à l'époque finale.

Nous avons donc recommencé nos tests pour trois couches en triplant le nombre d'époques (nous en avons pris 33 => 3 de plus par précaution). Il s'est avéré que, pour une certaine majorité, nous avons obtenu des résultats supérieurs à 10,84 %, dépassant même quelque fois 80%, pour l'accuracy, à une époque comprise entre 22 et 30. Ainsi, notre étude et notre analyse concernant l'influence du nombre d'époques s'est révélée exacte et nous a bien été utile.

couches cachées ▼	epoch croissance ▼	valeur epoch ▼	valeur final ▼
(16, 16, 16)	INFINI	10,84	10,84
(16, 16, 32)	INFINI	10,84	10,84
(16, 16, 64)	INFINI	10,84	10,84
(16, 16, 128)	INFINI	10,84	10,84
(32, 32, 32)	INFINI	10,84	10,84
(32, 32, 64)	INFINI	10,84	10,84
(32, 32, 128)	INFINI	10,84	10,84
(32, 32, 256)	INFINI	10,84	10,84
(32, 64, 32)	INFINI	10,84	10,84
(32, 64, 64)	INFINI	10,84	10,84
(32, 64, 128)	INFINI	10,84	10,84
(128, 128, 128)	INFINI	10,84	10,84
(128, 128, 256)	30	20,31	44,47
(128, 128, 512)	31	20,53	35,97
(128, 256, 128)	INFINI	10,84	10,84
(128, 256, 256)	33	18,96	18,96
(128, 256, 512)	25	18,33	79,36
(128, 512, 128)	28	16,87	51,36
(128, 512, 256)	23	20,47	83,13
(128, 512, 512)	19	10,96	92,86
(256, 128, 128)	INFINI	10,84	10,84
(256, 128, 256)	30	20,46	44,34
(256, 128, 512)	27	20,53	78,6
(256, 256, 128)	33	23,71	23,71
(256, 256, 256)	27	18,46	73,57
(256, 256, 512)	20	10,87	92,71
(256, 512, 128)	26	20,47	84,47
(256, 512, 256)	22	22,4	90,31
(256, 512, 512)	16	19,53	93,51
(512, 128, 128)	32	18,51	22,99
(512, 128, 256)	26	20,6	85,73
(512, 128, 512)	21	20,5	92,49
(512, 256, 128)	27	20,59	85,16
(512, 256, 256)	21	20,51	92,29
(512, 256, 512)	16	20,61	94,67
(512, 512, 128)	23	20,44	90,93
(512, 512, 256)	17	20,54	94,19
(512, 512, 512)	14	20,54	95,57

Nous observons ici que lorsque le nombre de neurones dans les couches est très bas, les résultats obtenus sont très faibles, autour de 10,84 %. Cela s'applique pour des nombres de neurones compris entre 16 et 128 inclus. À partir de 256 neurones dans une des couches, nous commençons à observer une légère augmentation de l'accuracy, mais uniquement à la toute fin des époques, avec une valeur relativement faible. Par exemple, pour une configuration de

couches (128, 128, 256), à partir de l'époque 30, l'accuracy passe de 10,84 % à 20,31 %, et à l'époque finale (33), nous atteignons 44,47 %.

Ensuite, lorsque nous utilisons un nombre de neurones plus élevé, comme 512 dans une des couches, et que le nombre minimum dans une autre couche est de 128, non seulement l'augmentation de l'accuracy se produit beaucoup plus tôt, mais la valeur finale de l'accuracy est également plus élevée. Par exemple, pour la configuration (128, 256, 512), à partir de l'époque 25, l'accuracy atteint déjà 18,33 %, et l'accuracy finale est de 79,36 %.

Enfin, il apparaît dans les dernières lignes du tableau que l'augmentation du nombre de neurones joue un rôle crucial, mais qu'il est également nécessaire de maintenir un certain équilibre entre le nombre de neurones dans chaque couche. Plus cet équilibre est respecté, plus l'augmentation de l'accuracy intervient tôt dans les époques, et plus la valeur finale de l'accuracy est élevée."

Cela met en lumière l'importance non seulement d'augmenter le nombre de neurones, mais également de veiller à leur répartition équilibrée dans les couches pour maximiser les performances du modèle.

Influence du nombre de couches cachées :

Encore une fois, nous avons effectué des tests en vain. En effet, même après avoir quintuplé le nombre d'époques à 50, nous obtenions des valeurs d'accuracy toujours égales à 10,84 %. Ces résultats étaient frustrants et inefficaces.

C'est pour cette raison que nous avons décidé de changer les valeurs par défaut de nos hyperparamètres. Nous avons ajusté le taux d'apprentissage (learning rate) à 0,001 et modifié l'initialisation des poids à (-0,1 ; 0,1). Jusqu'ici, ces valeurs s'étaient montrées efficaces pour nos tests à la fois avec des Shallow networks et des Deep networks, renvoyant constamment de bonnes valeurs d'accuracy.

Résultats après ajustement :

Non seulement cette modification a fonctionné, mais nous avons également constaté que 10 époques étaient déjà largement suffisantes pour obtenir des performances satisfaisantes. Dès la première époque, nous avons enregistré des résultats dépassant 90 % d'accuracy. Grâce à cette modification, nos résultats étaient même trop élevés, car à partir de certaines époques, entre la 6^e et la 8^e par exemple, les modèles continuaient d'apprendre trop longtemps, ce qui entraînait une dégradation de la performance en validation, donc un surapprentissage.

Ainsi, cette modification des hyperparamètres nous a permis non seulement d'améliorer considérablement les résultats pour obtenir une bonne accuracy, mais aussi de réduire le nombre d'époques.

Pour étudier l'influence du nombre de couches, nous avons ajouté 256 neurones à chaque nouvelle couche, en faisant varier le nombre de couches cachées de 3 à 20.

epoque	nb_couche =3	nb_couche =4	nb_couche =5	nb_couche =6	nb_couche =7	nb_couche =8	nb_couche =9	nb_couche =10	nb_couche =11	nb_couche =12	nb_couche =13	nb_couche =14	nb_couche =15	nb_couche =16	nb_couche =17	nb_couche =18	nb_couche =19	nb_couche =20
1	94,93	95,61	95,54	95,57	95,27	95,11	94,46	93,11	93,46	91,87	93,7	83,97	45,26	20,93	48,79	10,84	10,84	10,84
2	96,9	96,33	96,36	96,94	95,01	96,79	96,09	96,56	96,14	96	95,33	95,24	93,6	93,21	93,43	64,6	50,17	10,84
3	97,46	97,46	97,37	96,99	95,99	97,11	96,84	95,26	95,97	95,99	95,53	96,5	96,8	93,69	96,01	92,57	89,16	10,84
4	97,5	97,53	97,19	96,99	97,11	97,6	97	96,99	97,27	97,37	96,71	96,16	97,27	95,81	96,8	94,99	90,73	46,86
5	97,29	96,79	97,7	97,64	97,51	97,5	97,23	97,13	97,49	97,34	97,43	97,23	97,41	96,33	95,4	95,67	92,4	62,61
6	97,59	97,6	96,69	97,7	97,53	97,13	97,6	97,24	97,43	97,64	97,33	97,47	96,71	95,7	97,44	97,24	93,9	85,87
7	97,57	97,69	96,77	97,49	97,63	97,47	97,7	96,89	97,33	97,1	97,6	97,87	97,51	96,96	97,4	96,71	96,03	90,67
8	97,57	97,73	97,63	97,7	97,24	97,63	98,06	97,66	97,74	97,66	97,37	97,63	97,96	97,06	97,06	97,54	95,7	94,13
9	97,83	97,53	97,8	97,1	97,84	97,24	97,8	97,51	97,76	97,73	97,6	97,41	97,77	96,81	97,67	97,01	96,76	95,79
10	98,04	97,83	97,6	97,76	97,79	97,91	97,71	97,83	97,7	97,57	97,43	97,67	97,46	97,24	97,69	97,07	97,29	95,94

Ici, on constate que, quel que soit le nombre de couches, les résultats finaux sont excellents, variant de 95 % à 98 %. Cependant, le moment où le surapprentissage commence (indiqué en gris) est hétérogène et ne semble pas suivre une logique claire en fonction de l'augmentation du nombre de couches. En effet, on peut observer que le surapprentissage débute dès l'époque numéro 3 pour un nombre de couches cachées compris entre 10 et 12. De plus, il semblerait qu'à partir de 15 couches cachées, les résultats de la première époque commencent à se détériorer, même si, au fil des époques, le modèle apprend rapidement jusqu'à atteindre un surapprentissage. Les seuls modèles ne présentant pas de surapprentissage sont ceux à 3 couches cachées et à 20 couches cachées. Pour nos tests finaux, nous allons faire varier le nombre de couches cachées de 2 à 5, car nous souhaitons limiter l'utilisation des ressources. Rappelons que même avec un nombre réduit de couches cachées, nous obtenons d'excellents résultats.

Évaluation des Combinaisons d'Hyperparamètres par Tests

Imbriqués :

Comme pour le réseau peu profond, nous entrons ici dans une phase d'exploration ciblée des hyperparamètres. Dans cette deuxième partie, nous utilisons des boucles imbriquées pour tester des combinaisons de plages d'hyperparamètres identifiées comme prometteuses. Cette approche permet d'optimiser les performances du modèle sans surcharger nos ressources de calcul. En se concentrant sur des valeurs efficaces, nous maximisons l'accuracy tout en minimisant le risque de surapprentissage.

Nous avons exploré les hyperparamètres de notre modèle de perceptron multicouche profond en testant un total de 140 itérations (modèles différents), résultant de la combinaison des valeurs suivantes : quatre tailles de mini-batchs (`tab_batch_size` = [2, 3, 4, 5]), deux taux d'apprentissage (`tab_learning_rate` = [0.001, 0.01]), six tailles de couches cachées (`tab_hidden_size` = [(128, 256), (256, 128), (256, 256), (256, 256, 256), (256, 256, 256, 256), (256, 256, 256, 256, 256)]), et trois plages d'initialisation des poids (`tab_weight_init_range` =

$[(-0,001, 0,001), (-0,01, 0,01), (-0,1, 0,1)]$). L'exécution de toutes ces combinaisons a duré un total de 12 heures, 19 minutes et 30 secondes.

Après avoir exécuté ces tests, nous avons adopté une approche proactive pour surveiller et gérer le surapprentissage au sein de nos différents modèles de perceptron multicouche profond. Pour cela, nous avons construit un code permettant de filtrer les résultats des différentes combinaisons d'hyperparamètres en se basant sur les valeurs de perte de validation et d'entraînement.

Le code, illustré ci-dessous, lit les résultats à partir d'un fichier Excel contenant toutes les itérations effectuées pour chaque modèle, et applique les critères suivants : si la perte de validation actuelle est supérieure à la perte de validation de l'itération précédente et que la perte d'entraînement actuelle est inférieure à celle de l'itération précédente, cela signifie que nous commençons à faire face à un surapprentissage. Dans ce cas, nous enregistrons les résultats de l'itération précédente. Si aucun surapprentissage n'est observé, nous conservons les résultats de la dernière époque d'apprentissage.

```
from Excel import ExcelManager
from time import time, sleep

def main():
    input_file = 'excel/deep_network_combinaison.xlsx'
    output_file = 'excel/deep_network_combinaison_filtered.xlsx'
    sheet_name = 'EVERYTHING'
    column_titles = ["numero epoch", "batch_size", "nb_epochs",
"learning_rate", "input_size", "hidden_size", "output_size",
"weight_init_range", "Training Loss", "Validation Loss", "Test Loss",
"Accuracy"]

    manager = ExcelManager(input_file, column_titles)
    rows = manager.read_rows(sheet_name)

    # Ignorer la première ligne qui contient les titres des colonnes
    rows = rows[1:]

    filtered_manager = ExcelManager(output_file, column_titles)

    i_val_loss = column_titles.index("Validation Loss")
    i_train_loss = column_titles.index("Training Loss")
    i_num_epoch = column_titles.index("numero epoch")
    i_nb_epochs = column_titles.index("nb_epochs")

    i = 1
    while i < len(rows):
```

```

    current_row = rows[i]
    previous_row = rows[i - 1]
    epoque = 0
    print("ligne actuel : ", i + 1)
    if current_row[i_val_loss] > previous_row[i_val_loss] and
current_row[i_train_loss] < previous_row[i_train_loss]:
        print(f"\t{current_row[i_num_epoch]} : {current_row[i_val_loss]} >
{previous_row[i_val_loss]} et {current_row[i_train_loss]} <
{previous_row[i_train_loss]}")
        filtered_manager.add_row(sheet_name, previous_row)
        epoque = previous_row[i_num_epoch]
        i += previous_row[i_nb_epochs] - (epoque)
    elif current_row[i_num_epoch] == current_row[i_nb_epochs]:
        print(f"\t{current_row[i_num_epoch]} : {current_row[i_num_epoch]}
== {current_row[i_nb_epochs]}")
        filtered_manager.add_row(sheet_name, current_row)
        epoque = current_row[i_num_epoch]

    if epoque != 0:
        print(f"\tÉpoque de sauvegarde: {epoque}, pour le modele i={i}")

    i += 1

if __name__ == "__main__":
    main()

```

En appliquant cette approche systématique à chaque combinaison d'hyperparamètres, nous avons optimisé la sélection des modèles, maximisant ainsi l'accuracy tout en minimisant le risque de surapprentissage.

Ensuite, nous avons importé le fichier Excel filtré dans phpMyAdmin afin d'étudier les différentes données des modèles. Cette étape nous a permis d'analyser les performances de chaque modèle en fonction des hyperparamètres utilisés et d'explorer les relations entre les différentes métriques telles que la perte d'entraînement, la perte de validation et l'accuracy.

Après avoir importé le fichier Excel filtré dans phpMyAdmin pour étudier les différentes données des modèles, nous avons observé que chaque modèle s'est entraîné sur un nombre d'époques différent. En effet, certains modèles nécessitent moins de temps (époques) pour générer de bons résultats. En ne conservant que l'époque juste avant le surapprentissage, cette base de données devient pratique pour régénérer un modèle avec ses hyperparamètres, surtout dans des contextes où les contraintes de temps sont un facteur clé.

Grâce à la base de données que nous avons construite, nous avons pu analyser les performances des différents modèles. Les résultats montrent que l'accuracy minimale enregistrée est de 10.84 %, la moyenne des accuracies est de 63.97 %, et l'accuracy maximale atteint 98.01 %. Ces statistiques mettent en évidence la diversité des performances selon les configurations d'hyperparamètres utilisées.

De plus, en exécutant la requête SQL suivante :

```
SELECT COUNT(*) FROM `deep_network_combinaison_filtered`  
WHERE Accuracy = (SELECT MIN(Accuracy) FROM  
deep_network_combinaison_filtered);
```

Nous avons constaté que 51 modèles sur les 140 testés avaient l'accuracy minimale de 10.84 %. Cela reflète le fait que ces modèles n'ont pas réussi à apprendre efficacement à partir des données, probablement en raison de mauvaises combinaisons d'hyperparamètres ou d'un surapprentissage précoce.

Par ailleurs, la requête suivante :

```
SELECT COUNT(*) FROM `deep_network_combinaison_filtered` WHERE Accuracy > 90;
```

a révélé que 83 modèles ont obtenu une accuracy supérieure à 90 %, ce qui montre qu'une grande partie des modèles testés ont été capables de générer des résultats très performants. Ces modèles, ayant atteint une accuracy élevée avant le surapprentissage, pourraient être réentraînés avec les mêmes hyperparamètres, en particulier lorsque les contraintes de temps sont importantes.

En analysant plus en détail les époques nécessaires pour atteindre une accuracy supérieure à 90 %, nous avons obtenu les résultats suivants :

- Le nombre minimal d'époques pour atteindre une accuracy supérieure à 90 % est de 2:

```
SELECT MIN(numero_epoch) FROM deep_network_combinaison_filtered  
WHERE Accuracy > 90;
```

- Le nombre moyen d'époques pour atteindre cette accuracy est de 7.53 :

```
SELECT AVG(numero_epoch) FROM deep_network_combinaison_filtered  
WHERE Accuracy > 90;
```

- Enfin, le nombre maximal d'époques pour les modèles ayant une accuracy supérieure à 90 % est de 10 :

```
SELECT MAX(numero_epoch) FROM deep_network_combinaison_filtered  
WHERE Accuracy > 90;
```

Ces résultats montrent qu'il est possible d'obtenir une accuracy élevée après un nombre d'époques relativement faible, ce qui est un avantage significatif dans des situations où les contraintes de temps sont importantes.

En étudiant les modèles ayant une accuracy supérieure à 90 %, nous avons extrait plusieurs statistiques intéressantes concernant la répartition des hyperparamètres clés tels que la taille des mini-batches, le taux d'apprentissage, la plage d'initialisation des poids, la taille des couches cachées, et le nombre d'époques.

Taille des Mini-Batches

La répartition des tailles de mini-batches pour les modèles ayant obtenu une accuracy supérieure à 90 % montre une légère préférence pour les tailles de mini-batch plus petites :

```
SELECT batch_size, COUNT(*) * 100.0 / (SELECT COUNT(*) FROM
deep_network_combinaison_filtered WHERE accuracy > 90) AS percentage_above_avg
FROM deep_network_combinaison_filtered
WHERE accuracy > 90
GROUP BY batch_size
ORDER BY percentage_above_avg DESC;
```

<u>batch_size</u>	<u>percentage above avg 1</u>
2	27.71084
3	26.50602
4	24.09639
5	21.68675

Nous constatons que la taille de batch de 2 est légèrement plus efficace que les autres tailles, représentant 27.71 % des modèles avec une accuracy supérieure à 90 %.

Taux d'Apprentissage

Concernant le taux d'apprentissage, les modèles avec un taux de 0.01 semblent obtenir de meilleurs résultats que ceux utilisant 0.001 :

```
SELECT learning_rate, COUNT(*) * 100.0 / (SELECT COUNT(*) FROM
deep_network_combinaison_filtered WHERE accuracy > 90) AS percentage_above_avg
FROM deep_network_combinaison_filtered
WHERE accuracy > 90
GROUP BY learning_rate
ORDER BY percentage_above_avg DESC;
```

<u>learning_rate</u>	<u>percentage above avg 1</u>
0.01	60.24096
0.001	39.75904

Ainsi, le taux d'apprentissage de 0.01 est plus performant dans la majorité des cas pour obtenir des accuracies élevées.

Plage d'Initialisation des Poids

La plage d'initialisation des poids joue également un rôle crucial dans les performances des modèles :

```
SELECT weight_init_range, COUNT(*) * 100.0 / (SELECT COUNT(*) FROM
deep_network_combinaison_filtered WHERE accuracy > 90) AS percentage_above_avg
FROM deep_network_combinaison_filtered
WHERE accuracy > 90
GROUP BY weight_init_range
ORDER BY percentage_above_avg DESC;
```

<u>weight init range</u>	<u>percentage above avg 1</u>
(-0,1, 0,1)	53.01205
(-0,01, 0,01)	28.91566
(-0,001, 0,001)	18.07229

Une plage d'initialisation plus large, comme (-0.1, 0.1), semble favoriser de meilleurs résultats, représentant plus de 50 % des modèles performants.

Taille des Couches Cachées

Les modèles avec des couches cachées de taille plus modeste, comme (256, 256), sont les plus représentés parmi ceux avec une accuracy supérieure à 90 % :

```
SELECT hidden_size, COUNT(*) * 100.0 / (SELECT COUNT(*) FROM
deep_network_combinaison_filtered WHERE accuracy > 90) AS percentage_above_avg
FROM deep_network_combinaison_filtered
WHERE accuracy > 90
GROUP BY hidden_size
ORDER BY percentage_above_avg DESC;
```

<u>hidden_size</u>	<u>percentage above avg 1</u>
(256, 256)	24.09639
(128, 256)	22.89157
(256, 128)	21.68675
(256, 256, 256)	16.86747
(256, 256, 256, 256)	9.63855
(256, 256, 256, 256, 256)	4.81928

Les réseaux avec deux couches de taille 256 semblent être plus efficaces que des réseaux plus profonds, ce qui suggère qu'un modèle relativement peu profond peut souvent suffire à obtenir une accuracy élevée.

Nombre d'Époques

Enfin, l'analyse du nombre d'époques nécessaire pour obtenir une accuracy supérieure à 90 % montre que la majorité des modèles atteignent ces résultats en moins de 10 époques :

```
SELECT numero_epoch, COUNT(*) * 100.0 / (SELECT COUNT(*) FROM
deep_network_combinaison_filtered WHERE accuracy > 90) AS percentage_above_avg
FROM deep_network_combinaison_filtered
WHERE accuracy > 90
GROUP BY numero_epoch
ORDER BY numero_epoch DESC;
```

<u>numero epoch</u>	<u>1</u>	<u>percentage above avg</u>
10		44.57831
9		1.20482
8		3.61446
7		8.43373
6		22.89157
5		6.02410
4		4.81928
3		4.81928
2		3.61446

44.58 % des modèles ont atteint une accuracy supérieure à 90 % après 10 époques, mais des résultats satisfaisants peuvent être atteints en aussi peu que 2 époques, comme le montre la proportion de 3.61 % des modèles.

Grâce à notre étude approfondie des hyperparamètres, nous avons pu identifier les meilleures combinaisons qui produisent des modèles avec les performances les plus élevées. En analysant les pourcentages d'apparition de ces combinaisons, nous avons sélectionné les hyperparamètres qui se sont montrés les plus efficaces.

Ainsi, nous avons gardé la configuration suivante pour optimiser les performances :

- **Batch size** : 2
- **Learning rate** : 0.01
- **Hidden size** : (256, 256)
- **Weight init range** : (-0.1, 0.1)

En utilisant cette combinaison de paramètres, nous avons obtenu une accuracy maximale de **97.70%**, comme le montre la requête suivante :

```
SELECT * FROM deep_network_combinaison_filtered
WHERE batch_size = 2 AND learning_rate = 0.01 AND hidden_size = "(256, 256)"
AND weight_init_range = "(-0.1, 0.1)";
```

Cette combinaison se classe également en **13ème position** dans le classement global des modèles en fonction de leur accuracy, comme révélé par la requête suivante :

```
WITH RankedResults AS (
    SELECT RANK() OVER (ORDER BY Accuracy DESC) AS rank_position, batch_size,
    learning_rate, hidden_size, weight_init_range
    FROM deep_network_combinaison_filtered
)
SELECT rank_position
FROM RankedResults
WHERE batch_size = 2 AND learning_rate = 0.01 AND hidden_size = '(256, 256)'
AND weight_init_range = '(-0,1, 0,1)';
```

Ces résultats confirment que cette combinaison d'hyperparamètres est particulièrement efficace pour obtenir des modèles avec des performances optimales, tout en minimisant le nombre d'époques nécessaire.

```
SELECT numero_epoch, batch_size, learning_rate, hidden_size, Accuracy
FROM deep_network_combinaison_filtered
ORDER BY Accuracy DESC
LIMIT 10;
```

<u>numero_epoch</u>	<u>batch_size</u>	<u>learning_rate</u>	<u>hidden_size</u>	<u>Accuracy 1</u>
7	3	0.010000	(256, 256)	98.014286
6	3	0.010000	(256, 128)	97.871429
6	5	0.010000	(256, 256, 256)	97.842857
6	2	0.010000	(256, 128)	97.842857
6	4	0.010000	(256, 256, 256, 256)	97.800000
7	3	0.010000	(256, 256)	97.800000
6	5	0.010000	(128, 256)	97.785714
8	5	0.010000	(256, 256)	97.757143
3	3	0.010000	(256, 256, 256, 256)	97.728571
10	3	0.010000	(256, 128)	97.728571

Le meilleur modèle que nous avons obtenu est le suivant :

- **Numero d'époches final avant surapprentissage : 7**
- **Batch size : 3**
- **Learning rate : 0.01**
- **Hidden size : (256, 256)**
- **Accuracy : 98.01%**

Ce qui est particulièrement intéressant, c'est que ce modèle utilise un nombre relativement faible de neurones et de couches, tout en atteignant une époque finale avant le surapprentissage de seulement 7. Cela signifie que, dans un contexte où des contraintes de temps sont présentes et où nous n'avons pas de limites en termes de ressources (neurones), nous pourrions envisager une autre configuration, qui serait légèrement moins précise mais toujours performante :

- **Numero d'époches final avant surapprentissage : 3**
- **Batch size : 3**
- **Learning rate : 0.01**
- **Hidden size : (256, 256, 256, 256)**
- **Accuracy : 97.73%**

Ce modèle, classé 9ème, pourrait être une alternative viable si nous sommes prêts à sacrifier quelques points d'accuracy pour un entraînement plus rapide.

Partie 4 : CNN

Explication du code

Pour implémenter un réseau convolutif, nous nous sommes basés sur le modèle LeNet-5, présenté dans le sujet. Ce dernier consiste en une alternance de couches de convolution et de couches de pooling, suivies d'un aplatissement pour appliquer deux couches linéaires, chacune étant suivie d'une fonction d'activation ReLU. Enfin, une couche linéaire de sortie est ajoutée. Les paramétrages des différentes couches ont été réalisés selon l'architecture LeNet-5, mais adaptés pour une taille d'image d'entrée de 28x28 pixels (contrairement au 32x32 du modèle original).

Il est important de noter que l'initialisation des poids a été retirée pour cette partie, car nous avons choisi de nous concentrer sur les performances de l'architecture sans cette complexité. De plus, les dimensions des tenseurs de données ont été modifiées pour s'adapter à l'architecture convolutive (utilisant la couche `torch.nn.Conv2d`). Les dimensions des couches cachées ne sont pas spécifiées, car elles sont gérées automatiquement par le modèle, ce qui simplifie la configuration.

Le code correspondant se trouve dans le fichier `CNN.py`.

Classe `MNISTModel`

- **init :**
 - **Paramètres :**
 - `num_classes` : Nombre de classes de sortie (10 pour MNIST).
 - `batch_size` : Taille des lots pour l'entraînement.
 - `learning_rate` : Taux d'apprentissage pour l'optimiseur.
 - `nb_epochs` : Nombre d'époques pour l'entraînement.
 - `excel` : Objet pour la gestion des résultats dans un fichier Excel.
 - `gpu_automatic` : Utilisation automatique du GPU si disponible.
 - **Attributs :**
 - `self.device` : Définit l'appareil (CPU ou GPU) pour l'entraînement.
 - `self.model` : Construit le modèle en appelant `self.build_model()`.
 - `self.loss_func` : Fonction de perte (`CrossEntropyLoss`).
 - `self.optim` : Optimiseur (SGD).

- **build_model :**
 - **Architecture :**
 - Le modèle se compose de deux couches de convolution, suivies de normalisation par lots (BatchNorm2d) et de couches de pooling (MaxPool2d).
 - Après ces couches, une couche d'aplatissement (Flatten) transforme les données en un vecteur pour les couches linéaires.
 - Trois couches linéaires sont ensuite ajoutées avec des fonctions d'activation ReLU intercalées.
 - **Retour :** Le modèle est transféré sur l'appareil défini (self.device).

```

class MNISTModel:
    def __init__(self, num_classes=10, batch_size=5,
learning_rate=0.001, nb_epochs=10, excel=None, gpu_automatic=True):
        self.num_classes = num_classes
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.nb_epochs = nb_epochs
        self.excel = excel
        self.device = torch.device('cuda' if gpu_automatic and
torch.cuda.is_available() else 'cpu')
        self.model = self.build_model()
        self.loss_func = torch.nn.CrossEntropyLoss()
        self.optim = torch.optim.SGD(self.model.parameters(),
lr=self.learning_rate)

    def build_model(self):
        kernel_size = 5
        pooling_params = (2, 2)
        model = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=6,
kernel_size=kernel_size),
            torch.nn.BatchNorm2d(6),
            torch.nn.MaxPool2d(kernel_size=pooling_params[0],
stride=pooling_params[1]),
            torch.nn.Conv2d(in_channels=6, out_channels=16,
kernel_size=kernel_size),
            torch.nn.BatchNorm2d(16),
            torch.nn.MaxPool2d(kernel_size=pooling_params[0],
stride=pooling_params[1]),
            torch.nn.Flatten(),
            torch.nn.Linear(16 * 4 * 4, 120),
            torch.nn.ReLU(),
            torch.nn.Linear(120, 84),
            torch.nn.ReLU(),

```

```
torch.nn.Linear(84, self.num_classes)
)
return model.to(self.device)
```

Ce modèle suit l'architecture de LeNet-5, adaptée pour des images de 28x28 pixels, et intègre des techniques modernes telles que la normalisation par lots, ce qui améliore la stabilité et la vitesse de convergence lors de l'entraînement. Les dimensions des couches cachées sont gérées automatiquement par le modèle, simplifiant ainsi la configuration et permettant une flexibilité lors de l'entraînement.

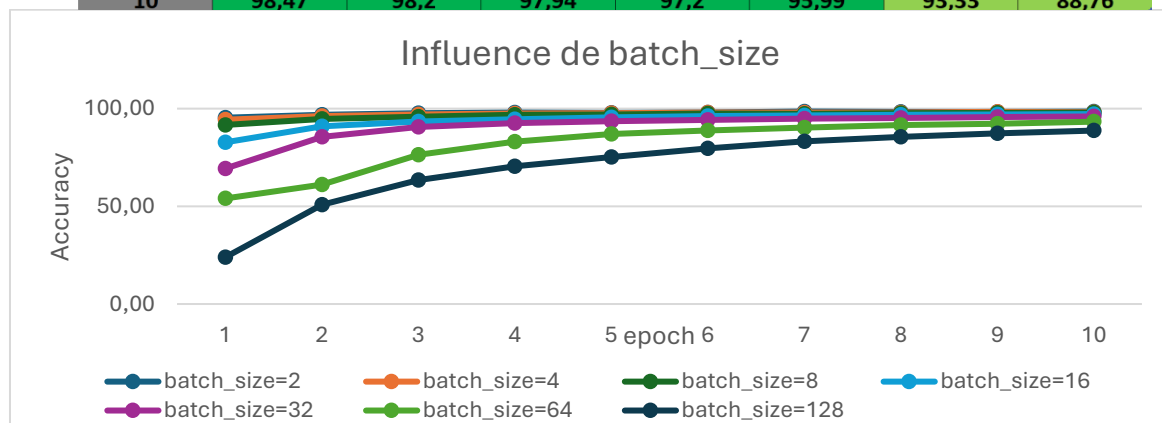
Dans cette configuration, les seuls hyperparamètres que l'on ajuste sont la taille du lot (batch_size) et le taux d'apprentissage (learning_rate), ce qui réduit la complexité lors de l'expérimentation tout en permettant de tester diverses configurations pour optimiser les performances du modèle.

Influence de chaque hyperparamètres :

Pour nos tests, nous avons utilisé le fichier main6.py, qui a été spécifiquement conçu pour évaluer les performances de notre modèle, à savoir le CNN que nous avons créé avec notre classe. Comme expliqué précédemment, notre étude se concentre uniquement sur l'influence de deux hyperparamètres clés : la taille du lot (batch_size) et le taux d'apprentissage (learning_rate). Les valeurs testées pour la taille du lot sont les suivantes : tab_batch_size = [2, 4, 8, 16, 32, 64, 128]. Quant au taux d'apprentissage, nous avons exploré les valeurs : tab_learning_rate = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.25, 0.5, 0.6, 0.75, 2]. Les résultats de ces tests sont sauvegardés dans le fichier main5.xlsx. L'exécution de toutes ces combinaisons a duré un total de 29 minutes et 56 secondes.

Influence de batch_size :

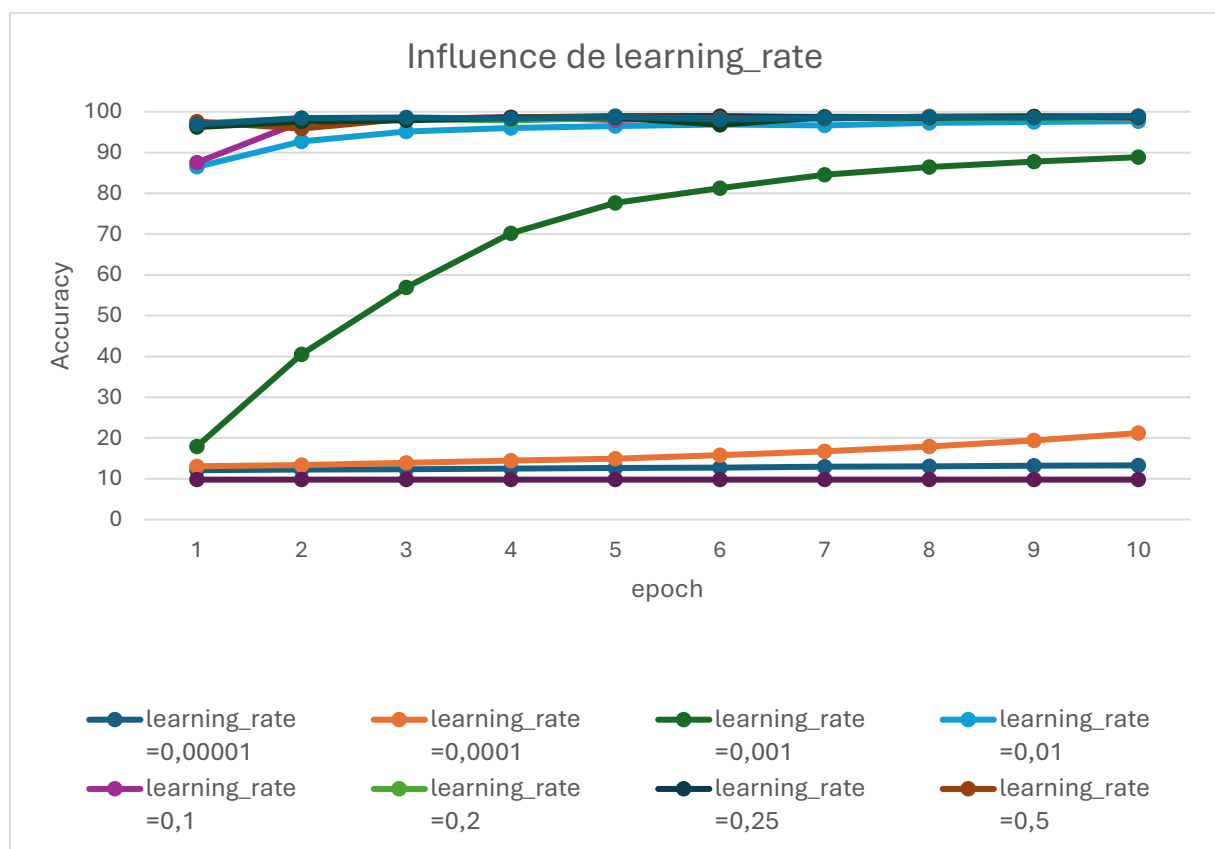
numéro epoqe	batch_size=2	batch_size=4	batch_size=8	batch_size=16	batch_size=32	batch_size=64	batch_size=128
1	95,37	94,33	91,66	82,76	69,34	54,03	23,93
2	96,91	96,26	94,73	91	85,5	61,07	50,74
3	97,6	97,04	95,91	93,41	90,61	76,43	63,47
4	97,96	97,56	96,89	94,56	92,63	83,04	70,46
5	97,89	97,66	97,09	95,54	93,51	87,06	75,3
6	98,01	97,97	97,47	96,16	94,24	88,87	79,59
7	98,53	97,96	97,36	96,5	94,83	90,37	83,24
8	98,24	97,94	97,64	96,79	95,21	91,6	85,54
9	98,37	98,36	97,9	96,96	95,64	92,33	87,41
10	98,47	98,2	97,94	97,2	95,99	93,33	88,76



On voit que des valeurs de batch_size plus petites, telles que 2 ou 4, permettent au CNN d'atteindre rapidement des précisions élevées, comme en témoigne une performance de 95,37 % lors de la première époque avec un batch_size de 2. Cette rapidité d'apprentissage peut être attribuée à la capacité du CNN à ajuster ses poids de manière plus fréquente avec des lots plus petits, ce qui favorise une meilleure convergence initiale. Cependant, avec des batch_size plus importants, comme 64 ou 128, bien que les performances initiales soient plus faibles, on observe que le modèle parvient à stabiliser et à maintenir des précisions élevées, dépassant 88%. Cela suggère que le CNN, avec son architecture adaptée et sa capacité à généraliser, est capable de compenser les effets d'une taille de lot plus élevée au fil du temps.

Influence de learning_rate :

numéro époque	learning_rate =0,00001	learning_rate =0,0001	learning_rate =0,001	learning_rate =0,01	learning_rate =0,1	learning_rate =0,2	learning_rate =0,25	learning_rate =0,5	learning_rate =0,6	learning_rate =0,75	learning_rate =2
1	12,1	13,07	17,91	86,47	87,54	97,13	96,73	97,54	96,23	96,84	9,79
2	12,24	13,39	40,54	92,71	97,47	98,33	98,26	95,84	97,6	98,44	9,79
3	12,36	13,91	56,9	95,14	98,31	98,17	98,2	98,29	97,91	98,59	9,79
4	12,5	14,41	70,19	95,97	98,61	97,89	98,61	98,63	98,56	98,26	9,79
5	12,67	14,94	77,64	96,44	98	98,47	98,89	98,79	98,59	98,76	9,79
6	12,76	15,8	81,21	96,86	98,61	98,56	98,94	98,4	96,77	98,09	9,79
7	12,93	16,73	84,53	96,61	98,73	98,53	98,64	98,26	98,64	98,51	9,79
8	13,04	17,91	86,41	97,27	98,64	98,51	98,69	98,8	98,5	98,66	9,79
9	13,17	19,43	87,76	97,51	98,8	98,46	98,79	98,87	98,76	98,49	9,79
10	13,29	21,19	88,84	97,69	98,86	98,67	98,83	98,56	98,79	98,89	9,79



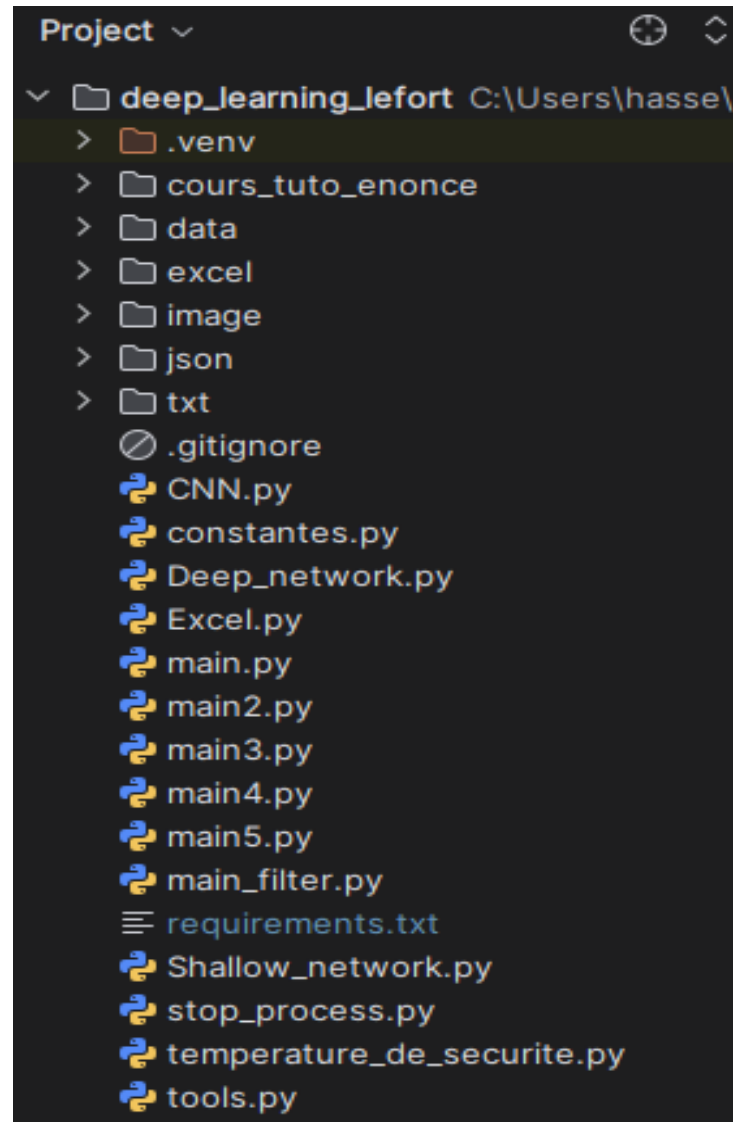
Après avoir analysé l'influence du learning rate, nous constatons que, comme d'habitude, les meilleurs résultats se situent entre 0.001 et 0.01. En dessous de 0.001, les résultats sont trop faibles, tandis qu'au-dessus de 0.01, le modèle commence à montrer des signes de surapprentissage malgré des résultats initiaux prometteurs. En effet, avec des learning rates très élevés, les performances se dégradent rapidement.

Conclusion :

Après avoir examiné l'influence des hyperparamètres batch_size et learning_rate, nous avons constaté que des valeurs de batch_size égales à 2 et de learning_rate à 0,001 fournissent déjà d'excellents résultats avec notre modèle CNN, atteignant plus de 95 % de précision dès les premières époques d'entraînement. Il est intéressant de noter que, dans la configuration initiale suivant le modèle LeNet-5, nos résultats sont supérieurs à ceux obtenus après de nombreux tests et ajustements des hyperparamètres dans les parties précédentes. Cela démontre clairement l'efficacité des réseaux de neurones convolutifs, comme LeNet-5, dans le traitement des images.

Annexes :

Organisation du répertoire de travail



- **cours_tuto_enonce** : Contient l'énoncé et les codes sources présents sur Moodle.
- **data** : Contient le fichier mnist.pkl.gz et la base de données avec les tables utilisées pour l'interprétation des données pour les réseaux shallow et deep.
- **excel** : Contient les fichiers Excel utilisés pour stocker les résultats. Les fichiers dont le nom se termine par "combinaison" stockent les résultats des tests imbriqués. Les fichiers CSV sont des fichiers XLSX convertis pour importation dans la base de données.
- **img** : Stocke des images comme les boxplots.

- **txt** : Contient des fichiers texte où nous avons sauvegardé la durée d'exécution de nos différents tests.
- **json** : Stocke des données sous forme JSON, notamment les PID des programmes effectuant nos tests. Cela est pratique pour arrêter les processus si nécessaire et enregistrer la durée d'exécution dans le dossier txt.
- **main?.py** : Les fichiers se terminant par un nombre impair représentent l'exécution de tests avec des boucles for séquentielles pour étudier l'influence de chaque hyperparamètre. Les fichiers pairs utilisent des boucles for imbriquées pour tester différentes combinaisons d'hyperparamètres.
- **shallow_network.py** : Contient la classe du réseau shallow.
- **Deep_network.py** et **CNN.py** : Contiennent les classes pour les réseaux deep et CNN respectivement.
- **stop_process.py** : Définit une durée maximale pour l'exécution des tests. Si cette durée est dépassée, le programme arrête les tests, sauvegarde la durée, effectue une sauvegarde sur le dépôt git et éteint le PC.
- **temperature_de_securite.py** : Assure que la température de la carte graphique ne dépasse pas 65°C pendant les tests. Si cette limite est dépassée, le programme arrête les tests en cours, enregistre les résultats dans le dépôt git et éteint le PC.
- **tools.py** : Contient des méthodes communes à tous les fichiers, comme l'enregistrement dans le dépôt git, la sauvegarde de la durée des tests, la vérification de la température du GPU, et la fourniture des `train_loader`, `val_loader`, `test_loader` à partir des `train_dataset` et `test_dataset`.