Arthur Desbiaux, p2006393
Valentin Cuzin-Rambaud, p2003442 oct. 2024

Introduction à l'apprentissage Profond

Rapport de projet

1. Introduction

L'objectif de ce projet est d'implémenter les modèles du MLP et du CNN vus en cours, de comprendre leur fonctionnement et de prendre en main le framework PyTorch. Le jeu de données MNIST contient des données de taille 28*28 de chiffres manuscrits, les labels sont encodés en one-hot par exemple :

Pour l'image  son label sera [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

Pour entraîner nos modèles, nous devons séparer notre jeu de données en trois parties : le Train sera utilisé pour entraîner notre modèle et mettre à jour nos poids. La validation sera une phase d'évaluation intermédiaire permettant de mesurer la perte et l'accuracy de la classification. Enfin le test avec lequel l'on calcule l'accuracy du meilleur modèle. Nous avons découpé notre data avec 0.72 % train, 0.18 % validation et enfin 0.1 % test. Voici comment on charge notre donnée :

```
def load_split_data(
        self, path: str = "mnist.pkl.gz", percentage: list() = [0.8, 0.2]
    ):
        ((data_train, label_train), (data_test, label_test)) = torch.load(
            gzip.open(path), weights_only=True
        )
        train_dataset = torch.utils.data.TensorDataset(data_train,
label_train)
        train_dataset, val_dataset = random_split(train_dataset, percentage)
        test_dataset = torch.utils.data.TensorDataset(data_test, label_test)
        return train_dataset, val_dataset, test_dataset
```

En utilisant des dataloaders, on sélectionne la donnée par mini-lots de façon aléatoire afin qu'elle soit idd (Independent and identically distributed) :

```
torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
```

2. Perceptron

Voici les détails de chaque tensors dans le fichier perceptron_pytorch.py :

- poids(w) : [784, 10] => 784 est le nombre de pixels de l'image que l'on possède (nombre d'entrées du perceptron) et 10 est le nombre de sorties
- biais(b) : [1, 10] => 1 chiffre qui sera le biais ajouté lors du calcul de la sortie du neurone et 10 est le nombre de sorties
- data_test : [7000, 784] => 7000 le nombre d'images total de test et 784 le nombre de pixels de chaque image.

- data_train : [63000, 784] => 63000 le nombre d'images total d'entraînement et 784 le nombre de pixels de chaque image.
- label_test : [7000, 10] => 7000 le nombre d'images total de test et 10 est le nombre de sortie.
- label_train : [63000, 10] => 63000 le nombre d'images total de test et 10 est le nombre de sortie.
- x : [5, 784] => 5 car c'est la taille du batch size et 784 est le nombre de pixels de l'image.
- y : [5, 10] => on fait le produit scalaire de deux matrices (x et w) on se retrouve donc avec 1 sorties par neurone. Nous avons 10 neurones et nous prenons 5 images donc 5x10.
- t : [5, 10] => Pour les 5 images on regarde le label en 10 sorties.
- grad : [5, 10] => On soustrait deux matrices de taille 5x10, on se retrouve donc avec une matrice de taille 5x10.

partie test :
- x : [1, 784] => 1 car dans la partie test on veut voir une image et 784 est le nombre de pixels de l'image.
- y : [1, 10] => on fait le produit scalaire de deux matrices (x et w) on se retrouve donc avec 1 sorties par neurone. Nous avons 10 neurones et nous prenons 1 image donc 1x10.
- t : [1, 10] => Pour l'image on regarde le label en 10 sorties.
- acc : [1] => On compte le nombre de bonnes réponses.

3. Shallow network

3.1. Définition du modèle

Le Shallow network est un perceptron multicouches avec une seule couche cachée et une sortie linéaire. Notre classe est étendue de nn.Module, lors de l'initialisation de notre classe, nous définissons le nombre de données en entrée, en sortie, mais également le nombre de neurones de la couche cachée. On peut donc initialiser notre classe avec $\mathbb{N}$ neurones, notre entrée est de 784 et une sortie de 10.
Ensuite dans notre méthode forward, nous passons par notre entrée, puis on appelle une fonction d'activation (Relu) et finalement, on passe par la sortie.
Le neurone fait le produit scalaire entre ses entrées et les poids.

```
class ShallowNet(nn.Module):

    def __init__(self, input_num, hidden_num, output_num):
        super(ShallowNet, self).__init__()
        self.hidelayer1 = nn.Linear(input_num, hidden_num)
        self.output = nn.Linear(hidden_num, output_num)

    def forward(self, x):
        x = self.hidelayer1(x)
        x = F.relu(x)
        x = self.output(x)
        return x
```

3.2. Hyper paramétrage naïf

Dans le but d'optimiser notre accuracy lors de nos tests, nous devons hyper paramétrer notre modèle. Pour cela, nous pouvons modifier :

- la taille du lot
- le nombre de neurones dans la couche cachée
- le taux d'apprentissage
- le nombre d'époques

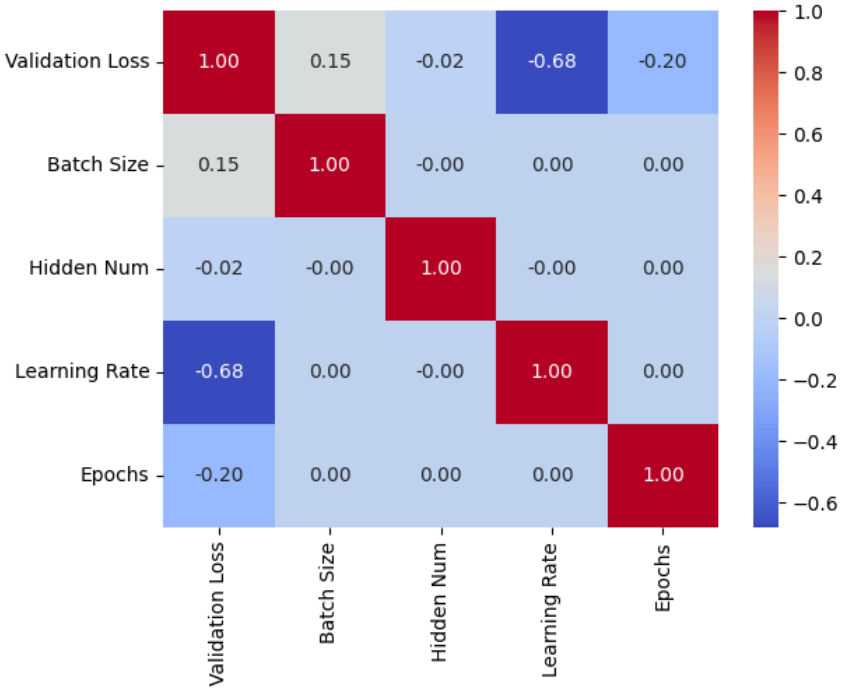
La méthode que nous avons implémentée est un grid search, c'est-à-dire une recherche de la plus faible perte pour toute combinaison possibles des valeurs des hyper-paramètres.

À la fin d'une époque, on va garder le dernier modèle avec son erreur calculée sur le jeu de validation. Lors de notre avancée, nous avons pu à plusieurs reprises reprendre notre façon de tester nos hyper paramètres. Pour garder le meilleur modèle (et pouvoir le tester), nous stockons tous les tests effectués sur le jeu de validation. Pour pouvoir observer l'influence des paramètres, nous avons notamment décidé dans un premier temps d'écrire nos informations d'hyper paramètres associées à une perte dans un fichier csv. Toutes nos traces sont disponibles dans le dossier csv. Un premier test a été effectué pour voir globalement l'impact des paramètres sur la perte. Ce premier test a été effectué avec les valeurs suivantes :

- taille du lot 3, 5 ou 10
- nombre de neurones dans la couche cachée 150, 200, 250 ou 300
- taux d'apprentissage 0.00001, 0.0001, 0.001 ou 0.01
- nombre d'époques 5, 10 ou 20

Notre meilleur score avec ces données est le suivant : 0.9807 soit 98,07% d'accuracy avec les paramètres taille du lot 3, nombre de neurones dans la couche cachée 300, taux d'apprentissage 0.01 et 20 époques.

Pour se rendre compte de l'influence de chaque paramètre, nous avons calculé la corrélation entre chaque paramètre.



Graphe de corrélation

Comme on peut le voir, le paramètre le plus important pour ces tests est le taux d'apprentissage suivi par le nombre d'époques.

Regardons les données plus précisément, commençons par l'impact seul du taux d'apprentissage:

Validation Loss	Batch Size	Hidden num	Learning rate	Epochs
0.091793440282 34482	3	150	1e-05	5
0.053398732095 9568	3	150	0.0001	5
0.025701027363 538742	3	150	0.001	5
0.012341232970 356941	3	150	0.01	5

Sur ces 3 exemples on peut voir la validation loss descendre énormément à chaque augmentation du taux d'apprentissage. On voit même une diminution de la perte de 0.08 .

Regardons maintenant l’impact du nombre de neurones dans la couche cachée :

Validation Loss	Batch Size	Hidden num	Learning rate	Epochs
0.012341232970356941	3	150	0.001	5
0.012302754446864128	3	200	0.001	5
0.011891470290720463	3	250	0.001	5
0.011730164289474487	3	300	0.001	5

On voit ici une légère augmentation entre 200 et 250 neurones. Alors qu'entre 150 et 200 il y avait une augmentation, mais plus légère.

Pour le nombre d’époque :

Validation Loss	Batch Size	Hidden num	Learning rate	Epochs
0.011730164289474487	3	300	0.001	5
0.009531934745609760976	3	300	0.001	10
0.008148823864758015	3	300	0.001	20

On voit une bonne augmentation tant qu'on augmente le nombre d'époques.
et finalement voyons l’impact de la taille du lot:

Validation Loss	Batch Size	Hidden num	Learning rate	Epochs
0.008148823864758015	3	300	0.001	5
0.009015226736664772	5	300	0.001	10
0.01074074488	10	300	0.001	20

Comme on le peut le remarquer trop l'augmenter nous fait augmenter la perte.

3.3. Hyper paramétrage naïf + Early-stopping

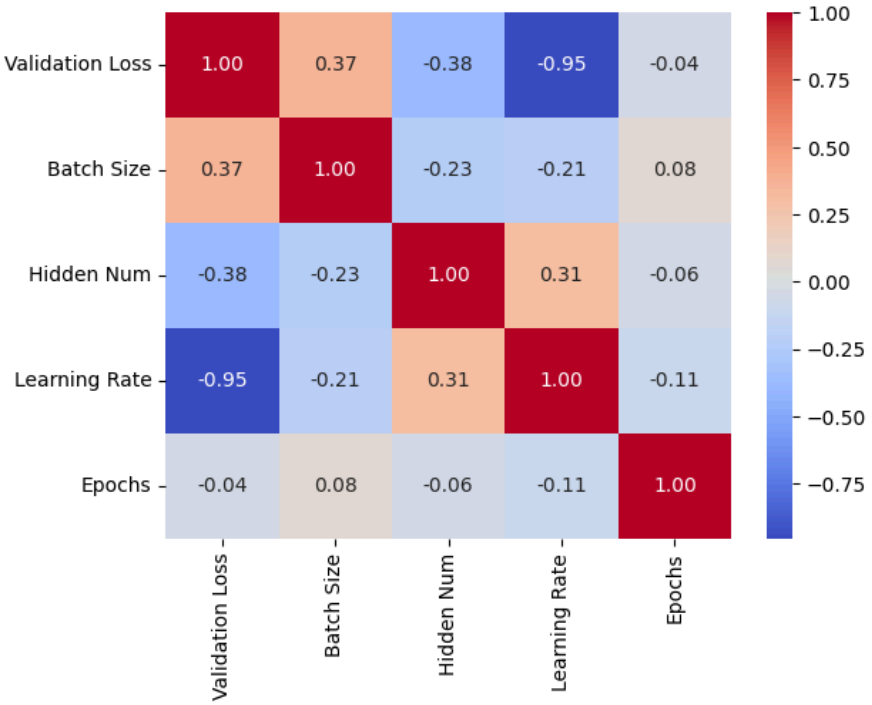
Pour éviter de tester le nombre d'époques à la volée dans nos hyper paramètres (complexité en temps plus faible car espace des solutions plus petit), nous avons implémenté un early-stopping, c'est une méthode qui détermine automatiquement le bon nombre d'époques nécessaire pour la convergence. Le principe est simple, si l'amélioration entre 2 époques n'est pas suffisamment grande pour continuer (inférieur à min_delta) alors on perd de la patience, jusqu'à ne plus en avoir et arrêter l'entraînement de ce modèle. En fixant le min_delta à 0.001 et la patience à 2, le nombre d'époques était souvent aux alentours de 6. En plus de l'implémentation de l'early-stopping, nous avons réalisé de nouveaux tests en prenant en compte les résultats des précédents. Ainsi, nous avons testé avec :

- taille du lot 1, 3 ou 5
- nombre de neurones dans la couche cachée 250, 350, 500 ou 600
- taux d'apprentissage 0.00001, 0.0001, 0.001 ou 0.01

Ainsi voici notre 2ème meilleur modèle :

- taille du lot 1
- nombre de neurones couche cachée 600
- taux d'apprentissage 0.01
- early stop arrêté à 6 époques

Pour un score de 98,50% d'accuracy au test.



Sur les nouvelles données les corrélations ont bien changé, on peut voir que la taille du lot et le nombre de neurones de la couche cachée sont beaucoup plus corrélés à la perte.
Un paramètre qui pourrait potentiellement être intéressant de continuer à monter serait le nombre de neurones dans la couche cachée et voir l'impact.

Validation Loss	Batch Size	Hidden num	Learning rate	Epochs
0.007044011261314154	1	250	0.01	6
0.006845048628747463	1	300	0.01	6
0.006447544787079096	1	500	0.01	6
0.006414782721549272 5	1	600	0.01	6

Comme on peut le voir ici, on voit une différence entre 350 et 500, mais cette différence est moins importante entre 500 et 600. Il serait intéressant de comparer cela à des valeurs allant bien au-delà, par exemple [600, 1200, 1500]. Nous faisons donc de nouveau test avec les paramètres suivants :

- taille du lot 1 ou 3
- nombre de neurones dans la couche cachée 600, 1200 ou 1500
- taux d'apprentissage 0.05, 0.01 ou 0.1

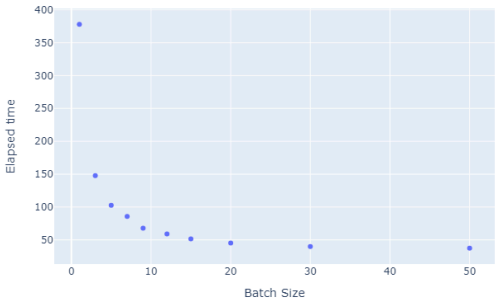
Les meilleurs paramètre que nous avons eu sont :

- taille du lot 3
- nombre de neurones couche cachée 1500
- taux d'apprentissage 0.1
- early stop arrêté à 6 époque

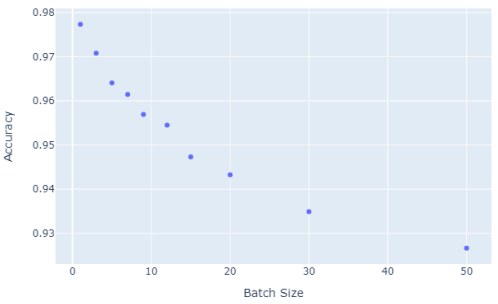
Le tout pour un score de 0.9870 soit 98,70%.

Une question s'est posée lors des différentes exécutions, quel est l'impact des variables sur le temps d'exécution, mais également à quel point cela impacte l'accuracy. Pour répondre à ces questions, nous avons modifié notre code pour calculer le temps d'exécution de notre training, mais aussi le taux de l'accuracy sur le jeu de validation. Nous avons donc également créé un nouveau csv qui contient toutes ces informations.

Afin de mieux comprendre l'impact, nous avons réalisé différents graphes :



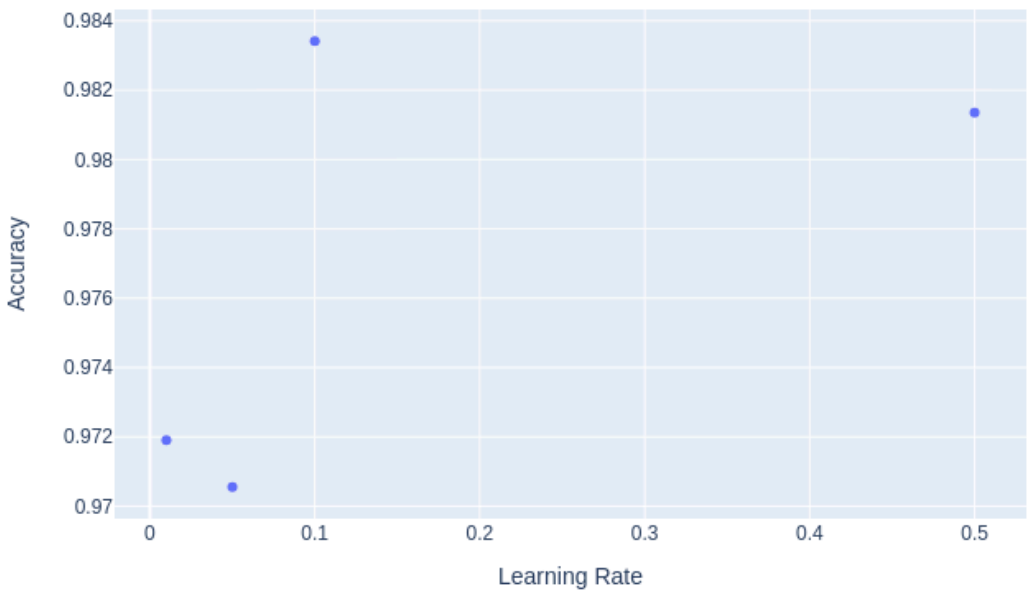
Temps en fonction de la taille du lot



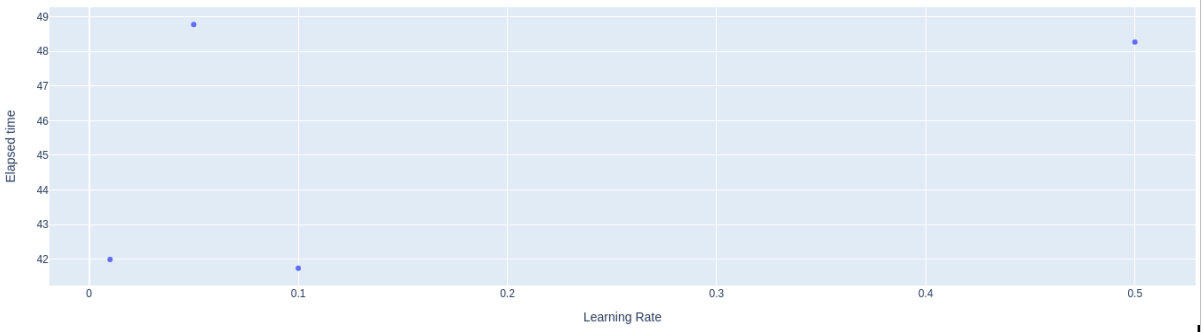
Accuracy en fonction de la taille du lot

Comme on peut le voir, une taille du lot a 1 nous permet d'être le plus précis, mais cela implique surtout un temps de calcul énorme. En passant d'une taille du lot de 1 à 3 on remarque une perte très légère (baisse de 0.007), alors qu'on voit une grosse différence de temps de calcul (377s pour taille du lot 1 contre 147s pour taille du lot 3). Aux alentours de la taille du lot 10, le temps de calcul diminue de moins en moins alors que l'accuracy continue a bien diminuer.

Regardons maintenant l'impact du taux d'apprentissage:

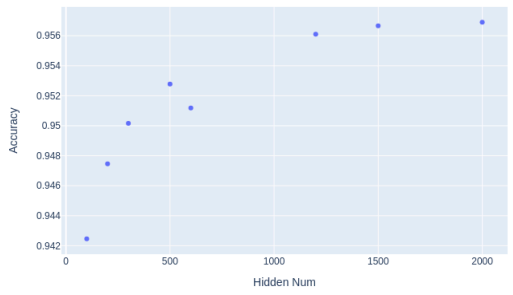


Accuracy en fonction du taux d'apprentissage

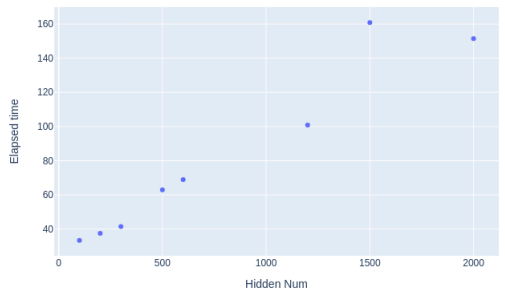


Temps en fonction du taux d'apprentissage

À l'aide de ces deux graphes, on remarque que le meilleur taux d'apprentissage en termes d'accuracy est 0.1 et qu'il est également (avec 0.01) l'un des seuls à avoir un taux d'apprentissage si minime.
Regardons maintenant l'impact du nombre de neurones dans la couche cachée:



Accuracy en fonction du nombre de neurone



Temps en fonction du nombre de neurone

Cette fois-ci, on voit une très légère augmentation de l'accuracy (à partir de 600 neurones, petite évolution). Mais cela impacte beaucoup le temps de calcul (Presque le double du temps de calcul entre 600 et 1200). Il serait donc plus intéressant de rester à un nombre de neurones < 600.

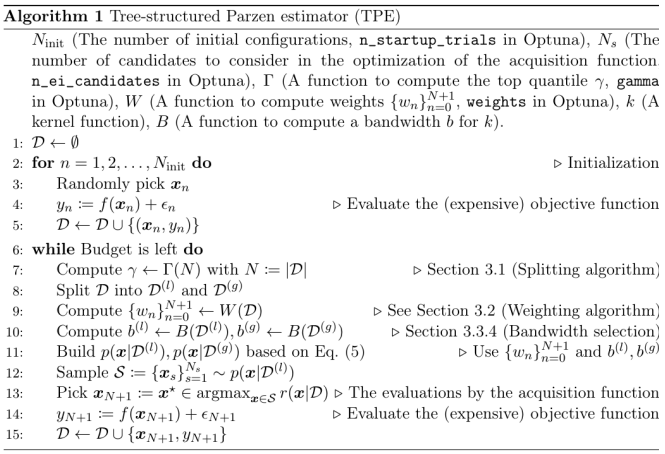
3.4. Automatique Hyper paramétrage

La recherche d'hyper-paramètres est un problème complexe, et nous avons remarqué plusieurs défauts à l'approche grid-search. Le premier problème est le temps de calcul trop long sur nos machines (nous n'avons pas CUDA), le deuxième problème, c'est que l'espace des solutions est construit par nous, et que nous ne sommes pas fiables pour supposer quels paramètres peuvent fonctionner. On a alors pensé à regarder si des solutions de recherche automatique d'hyper-paramétrage existaient.

3.4.1. Optuna

Optuna est une bibliothèque pour l'hyper-paramétrage de modèle, incluant une multitude d'algorithmes. Optuna va chercher à maximiser une fonction *objective* qui retourne

chez nous l'accuracy. Elle gère automatiquement l'espace de recherche des solutions via l'objet Trial qui permet de définir les bornes de recherche pour chaque hyper-paramètres. L'algorithme par défaut de sampling est le Tree-structured Parzen Estimator Understanding (TPE) qui est une variante de Bayesian Optimisation. Pour faire simple, il est composé de 5 parties :



Algorithm 1 Tree-structured Parzen estimator (TPE)

N_{init} (The number of initial configurations, `n_startup_trials` in Optuna), N_s (The number of candidates to consider in the optimization of the acquisition function `n_ei_candidates` in Optuna), Γ (A function to compute the top quantile γ, `gamma` in Optuna), W (A function to compute weights $\{w_n\}_{n=0}^{N+1}$, `weights` in Optuna), k (A kernel function), B (A function to compute a bandwidth b for k).

$\mathcal{D} \leftarrow \emptyset$
for $n = 1, 2, \ldots, N_{\text{init}}$ **do** ▷ Initialization
  Randomly pick $\boldsymbol{x}_n$
  $y_n \coloneqq f(\boldsymbol{x}_n) + \epsilon_n$ ▷ Evaluate the (expensive) objective function
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_n, y_n)\}$
while Budget is left **do**
  Compute $\gamma \leftarrow \Gamma(N)$ with $N \coloneqq |\mathcal{D}|$ ▷ Section 3.1 (Splitting algorithm)
  Split $\mathcal{D}$ into $\mathcal{D}^{(l)}$ and $\mathcal{D}^{(g)}$
  Compute $\{w_n\}_{n=0}^{N+1} \leftarrow W(\mathcal{D})$ ▷ See Section 3.2 (Weighting algorithm)
  Compute $b^{(l)} \leftarrow B(\mathcal{D}^{(l)}), b^{(g)} \leftarrow B(\mathcal{D}^{(g)})$ ▷ Section 3.3.4 (Bandwidth selection)
  Build $p(\boldsymbol{x}|\mathcal{D}^{(l)}), p(\boldsymbol{x}|\mathcal{D}^{(g)})$ based on Eq. (5) ▷ Use $\{w_n\}_{n=0}^{N+1}$ and $b^{(l)}, b^{(g)}$
  Sample $\mathcal{S} \coloneqq \{\boldsymbol{x}_s\}_{s=1}^{N_s} \sim p(\boldsymbol{x}|\mathcal{D}^{(l)})$
  Pick $\boldsymbol{x}_{N+1} \coloneqq \boldsymbol{x}^* \in \operatorname{argmax}_{\boldsymbol{x} \in \mathcal{S}} r(\boldsymbol{x}|\mathcal{D})$ ▷ The evaluations by the acquisition function
  $y_{N+1} \coloneqq f(\boldsymbol{x}_{N+1}) + \epsilon_{N+1}$ ▷ Evaluate the (expensive) objective function
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\boldsymbol{x}_{N+1}, y_{N+1}\}$

1. Premiers essais aléatoires N fois et évaluations.
2. Algo de splitting, qui divise l'espace de recherche.
3. Algo de weighting, qui pondère les KDE (Estimateur de la densitée par noyau voir : wiki)
4. Sélection de la Bandwidth, une large bande veut dire plus d'exploration et vice versa.
5. Évaluation de la fonction objective avec les hyper-paramètres choisis, puis boucle sur 2.

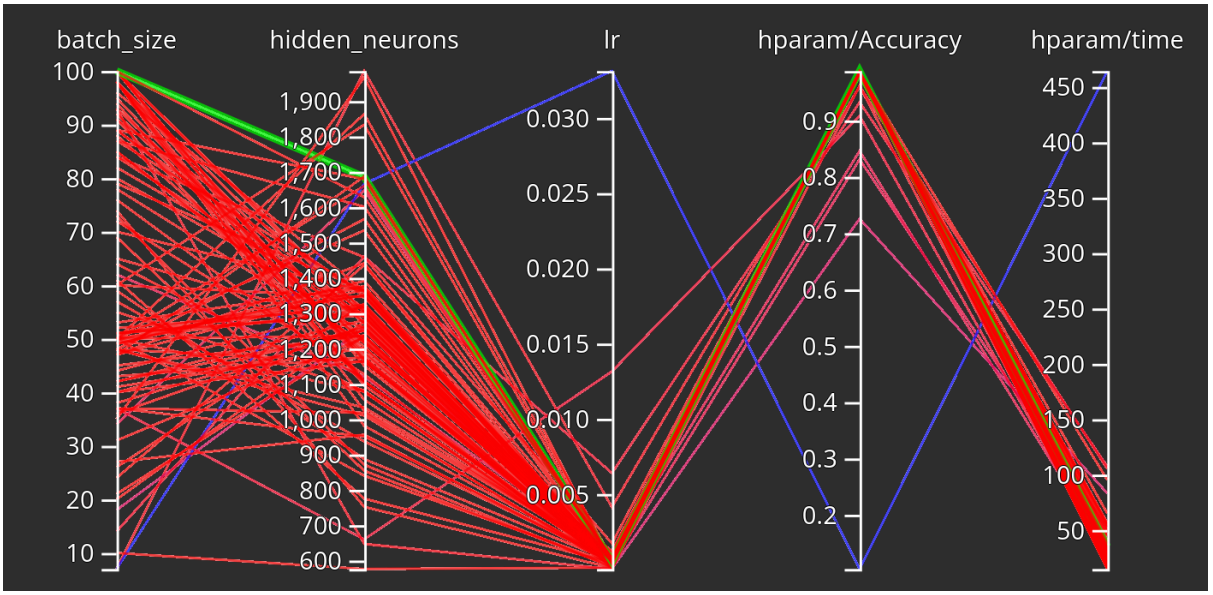
De plus, Optuna implémente le pruner, un algorithme permettant de stopper un essai d'hyper-paramétrage en cours s'il ne semble pas être efficace. Il fonctionne en deux phases. (1) Regarde périodiquement la valeur intermédiaire à optimiser de l'essai, puis (2) termine l'essai si sa comparaison avec les autres valeurs intermédiaires (des autres essais), ne démontre pas qu'il est concurrent.

Nous faisons donc de nouveau test avec les paramètres suivants :

- taille du lot entre 3 et 100
- nombre de neurones dans la couche cachée entre 500 et 2000
- taux d'apprentissage entre 1e-5 et 1e-1
- optimiser ADAM ou SGD
- nombre de trials 1000

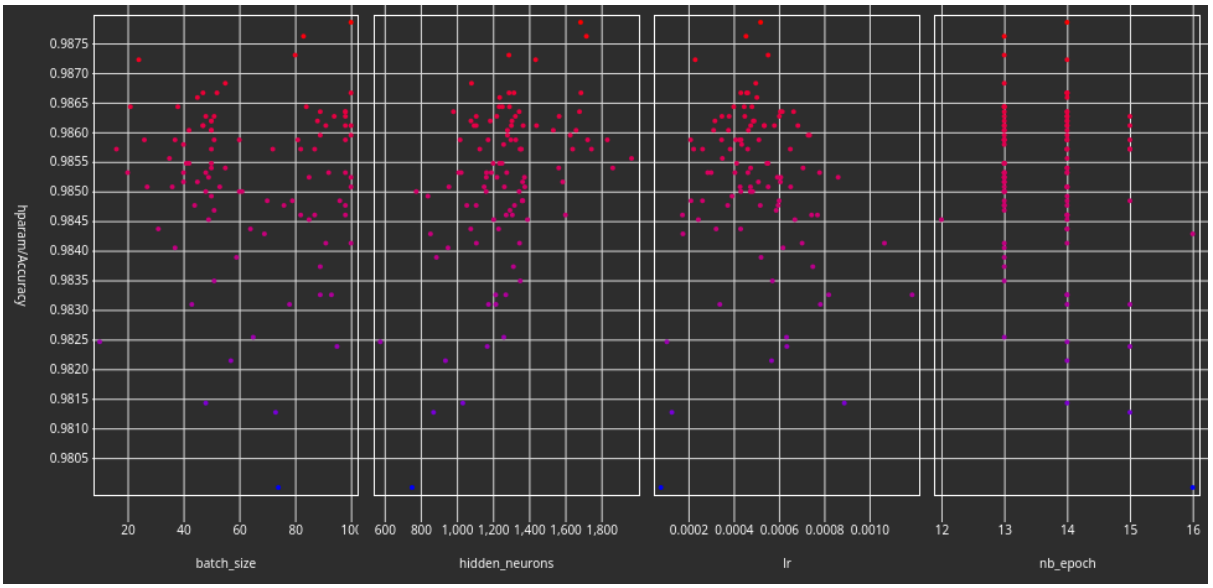
3.4.2. TensorBoard

L'utilisation de TensorBoard pour la visualisation d'hyperparamètres permet de tirer beaucoup de connaissance, en comparaison à l'analyse de fichier CSV. Parmi les 1000 trials seuls 110 ont été épargnés par le pruner, ce qui nous fait gagner un temps considérable.



En vert, le meilleur hyper-paramétrage.

L'algorithme semble déterminer qu'un taux d'apprentissage compris entre 0.005 et 0.0001 maximise l'accuracy, ce qui diffère complètement de notre précédente étude avec le grid-search. Les hyper-paramètres sont à corréler entre eux, ainsi la vue en 2 dimensions permet de bien comprendre l'influence des paramètres entre eux :



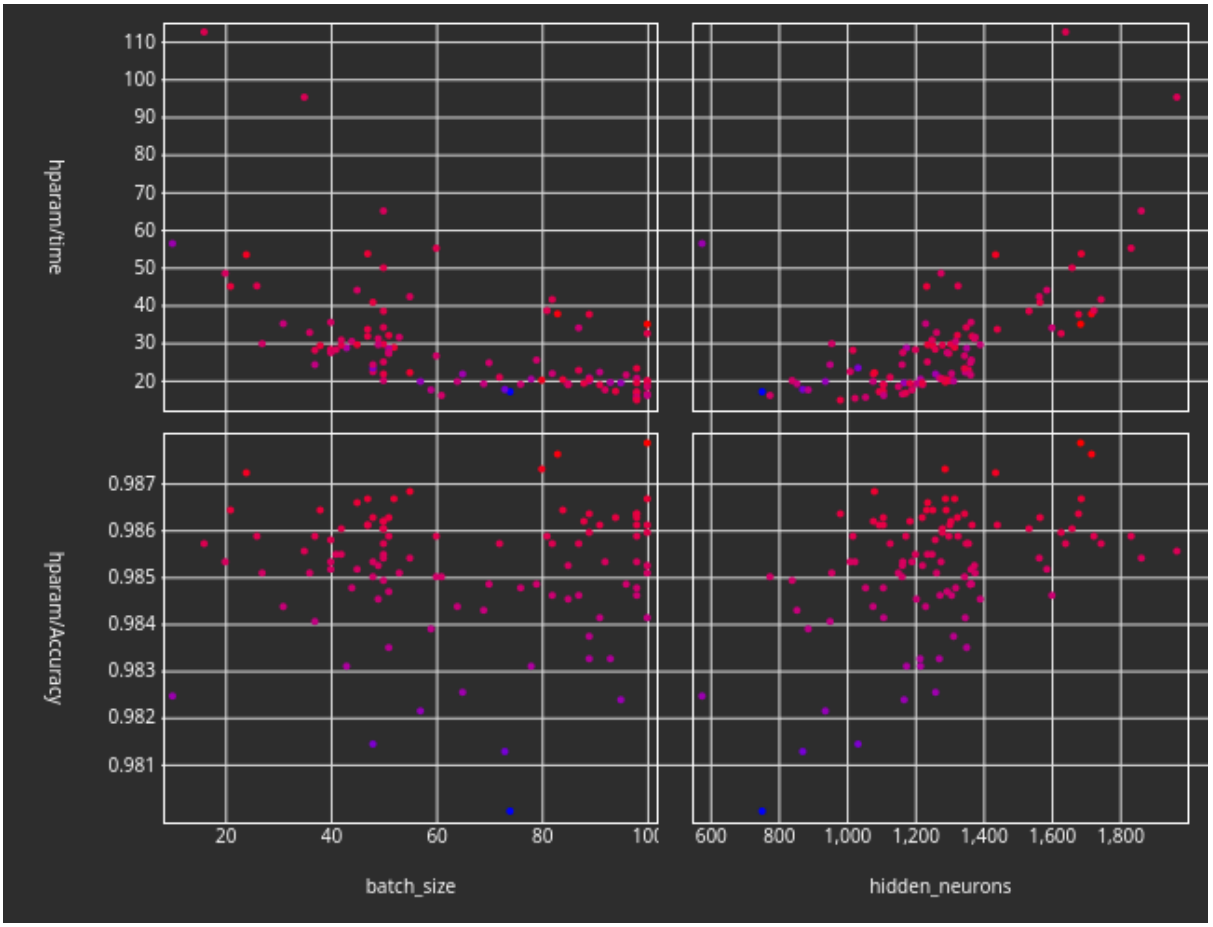
Coloration par accuracy, sélection des trials > à 0.98 d'accuracy

Les meilleurs paramètres que nous avons eus sont :

- taille du lot 100
- nombre de neurones couche cachée 1685
- taux d'apprentissage 0.0005
- optimiser ADAM

Le tout pour un score de 0.9878 soit 98,78%.
Ce nouvel hyper paramétrage est entièrement différent de ceux trouvés avec le grid search. La taille du lot est à 100 et le taux d'apprentissage à 0.0005, ce qui est très faible. Cela nous montre

que plusieurs combinaisons sont possibles, et que chaque paramètre est à corréler avec les autres. On ne peut pas dire de façon certaine que notre hyper paramétrage est le plus optimal, ni même dire que tel paramètre doit être fixé à telle valeur pour que ça fonctionne bien. De plus, nous avons précédemment déterminé que monter le nombre de neurones au-dessus de 600 était peu pertinent au vu de la durée d'entraînement par rapport au gain. Cependant :



Comme on peut le voir, le temps est corrélé aussi bien avec le nombre de neurones (plus il est grand, plus c'est long) qu'avec la taille du lot (plus elle est petite, plus c'est long). Nous n'avions pas eu l'idée de tester cette combinaison gagnante : grande taille de lot, beaucoup de neurones. L'interaction entre le taux d'apprentissage et le nombre d'époques est également important. Un taux d'apprentissage élevé (ex. 0.1) permet de faire des progrès rapides, mais avec un nombre d'époques réduit. En revanche, avec un taux d'apprentissage plus faible, plus d'époques sont nécessaires pour atteindre des résultats satisfaisants

4. Deep network

4.1. Le Modèle

Le Deep network est un perceptron multicouches avec plusieurs couches et une sortie linéaire. Notre classe est étendue de nn.Module , lors de l'initialisation de notre classe, nous définissons le nombre de données en entrée, le nombre de couches, le nombre de neurones par couche et la sortie. On peut donc initialiser notre classe avec $\mathbb{M}$ couche et $\mathbb{N}$ neurones, notre entrée est de 784 et une sortie de 10.

Dans un premier temps, dans notre implémentation, chaque couche avait le même nombre de neurones (sauf l'entrée et la sortie), par la suite, nous avons modifié cela pour que chaque couche puisse posséder des nombres de neurones différents.
Ensuite, dans notre méthode forward, nous appelons chaque couche une à une et appliquons une fonction d'activation (Relu) et finalement, on passe par la sortie.

```
class Mlp(nn.Module):

	def __init__(self, input_nbr, hidden_layers_nbr, hidden_neuron_nbr,
output_nbr):
	super(Mlp, self).__init__()
	self.hidden_layers = nn.ModuleList([nn.Linear(input_nbr,
hidden_neuron_nbr)])
	for n in range(hidden_layers_nbr-1):
		self.hidden_layers.append(nn.Linear(hidden_neuron_nbr,
hidden_neuron_nbr))
	self.output = nn.Linear(hidden_neuron_nbr, output_nbr)

	def forward(self, x):
	for layer in self.hidden_layers:
		x = layer(x)
		x = F.relu(x)
	x = self.output(x)
	return x
```

4.2. Hyper paramétrage naïf + Early-stopping

Dans un premier temps, nous avons voulu fait un test avec les paramètres suivants :

- taille du lot 1, 3 ou 9
- nombre de couche 1, 5, 10 ou 20
- nombre de neurones dans la couche cachée 250, 350, 500 ou 6000
- taux d'apprentissage 0.05, 0.01 ou 0.1

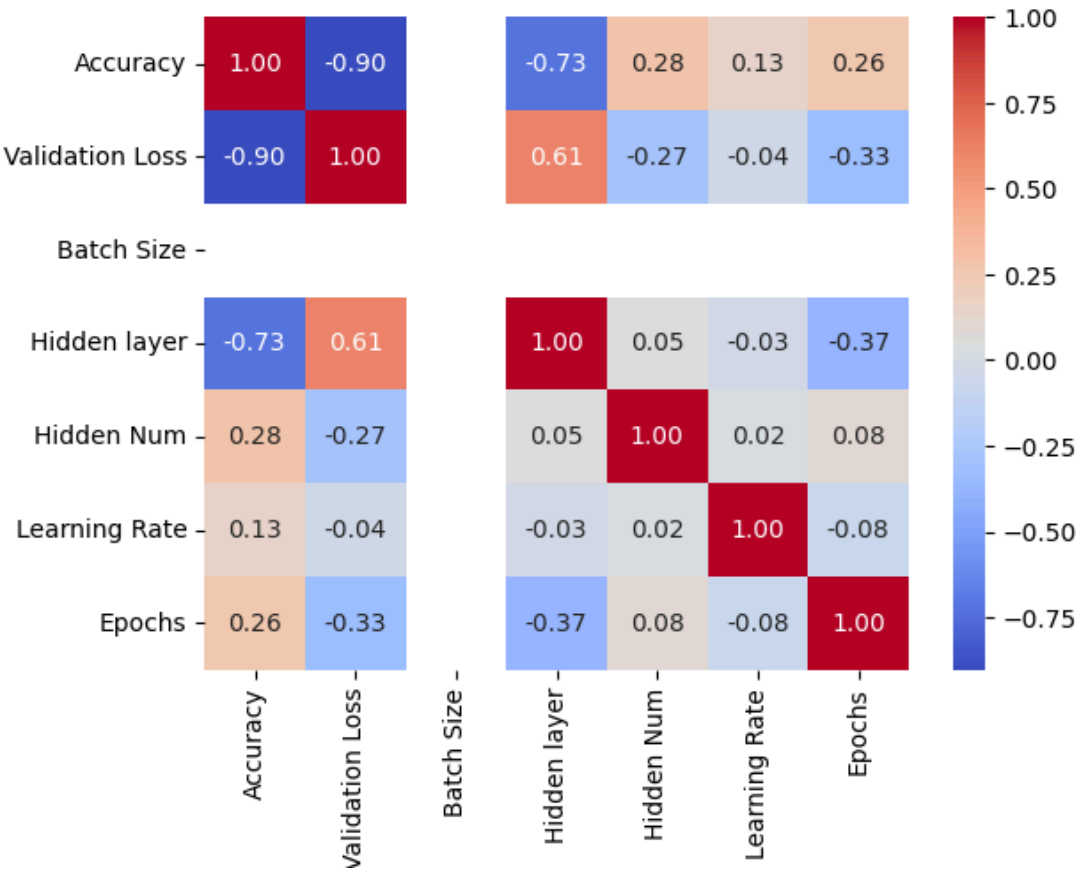
Mais face au temps de calcul très long, nous avons limité la taille du lot à 1

Les meilleurs paramètres que nous avons eus sont :

- taille du lot 1
- nombre de couches 5
- nombre de neurones couche cachée 600
- taux d'apprentissage 0.01

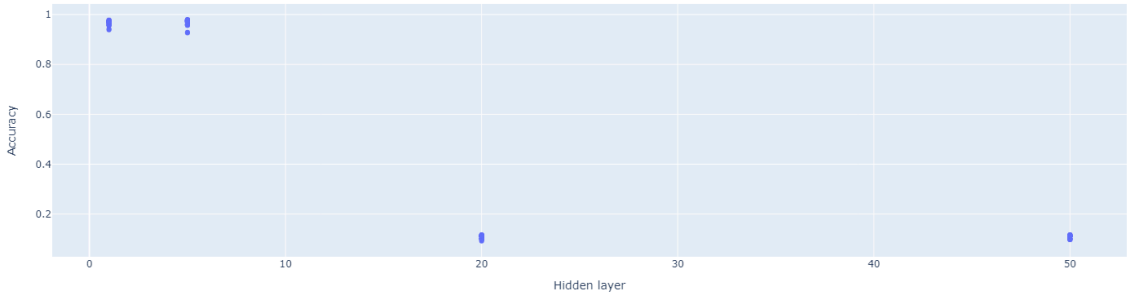
Le tout pour un score de 0.9864 soit 98,64%

Pour se rendre compte de l'influence de chaque paramètre, nous avons calculé la corrélation entre chaque paramètre.



Graphe de corrélation

Comme on peut le voir, ce qui est le plus corrélé est le nombre de couches.

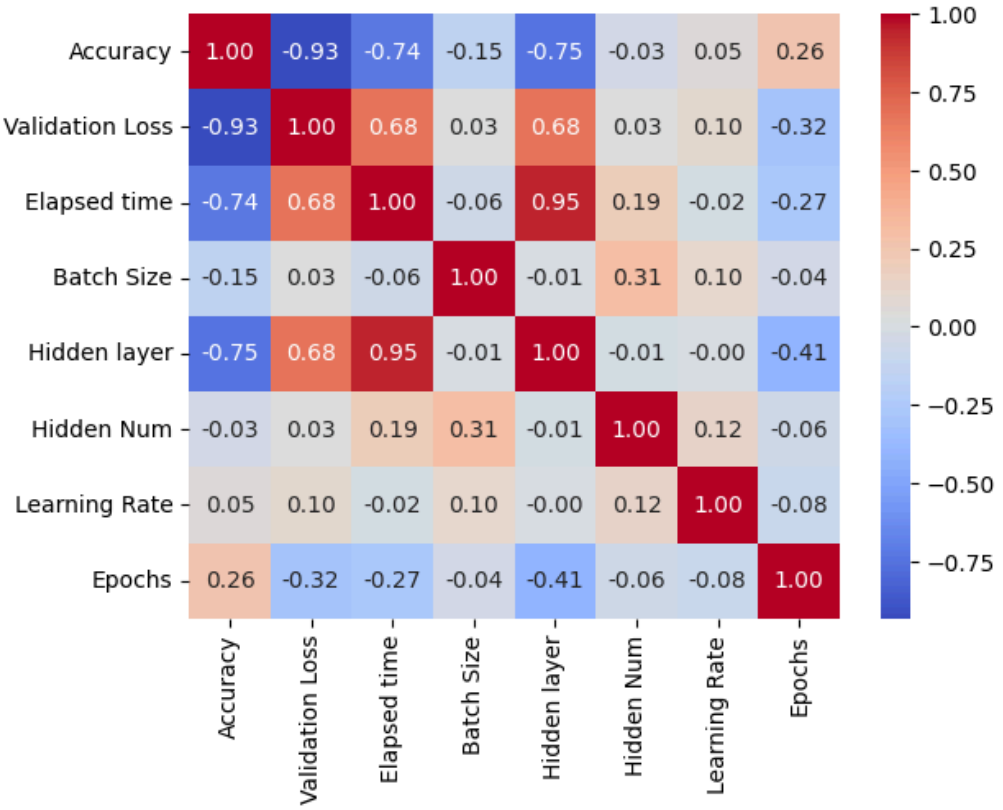


Accuracy en fonction du nombre de couches

Comme on le voit sur ce graphique, les meilleurs résultats en termes d'accuracy sont lorsqu'on possède un nombre de couches compris entre 1 et 5.

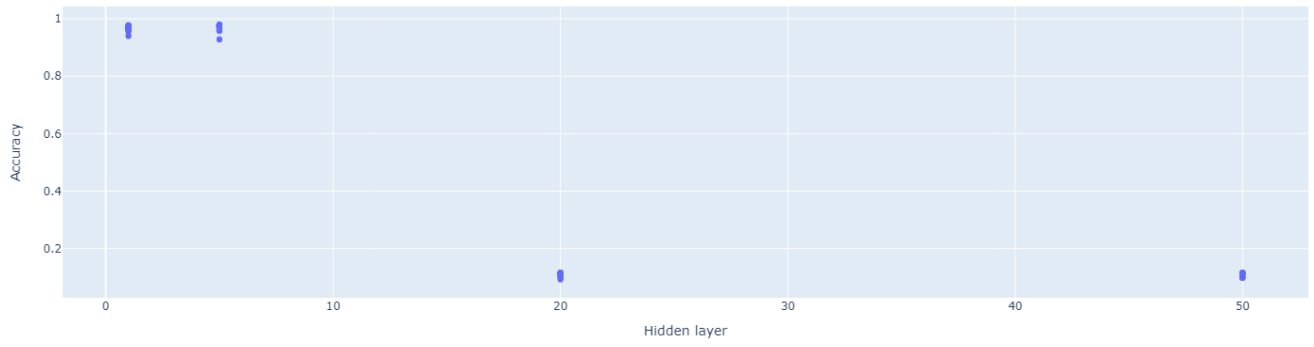
Par la suite, nous avons réalisé de nouveau test sur ce jeu de données :

- taille du lot 9, 15 ou 30
- nombre de couches 1, 5, 10 ou 20
- nombre de neurones dans la couche cachée 250, 350, 500 ou 6000
- taux d'apprentissage 0.05, 0.01 ou 0.1

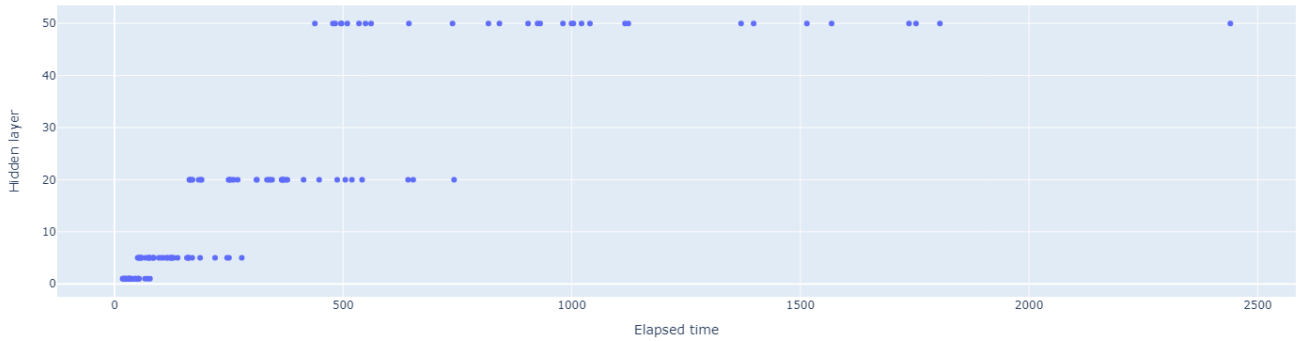


Graphe de corrélation

Comme on peut le voir, même en changeant la taille du lot, le nombre de couches reste l'argument le plus corrélé.



Accuracy en fonction du nombre de couches



Nombre de couches en fonction du temps

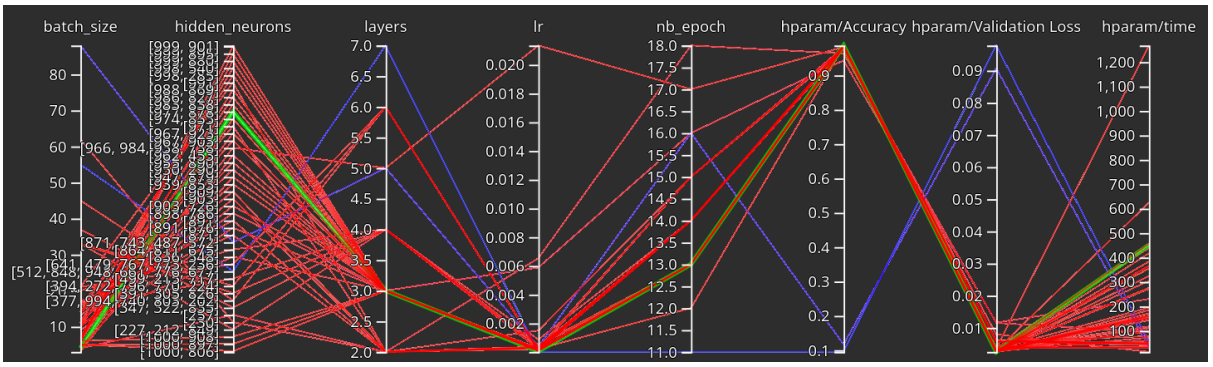
On voit donc que mettre un nombre de couches trop élevé impacte énormément le temps de calcul

4.3. Automatique Hyper paramétrage

Nous faisons donc de nouveau test avec les paramètres suivants :

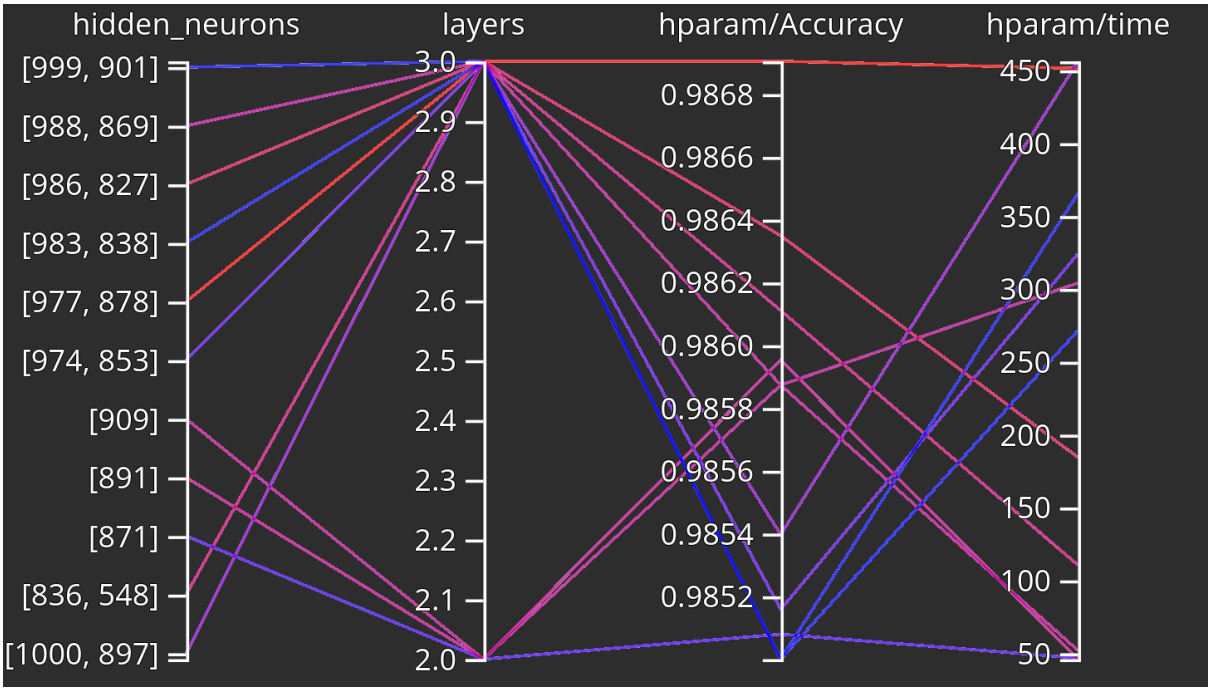
- taille batch entre 3 et 100
- nombre de couches entre 2 et 10
- nombre de neurones dans la couche cachée entre 200 et 1000
- taux d'apprentissage entre 1e-5 et 1e-1
- optimiser ADAM ou SGD
- 3H de run pour 133 trials

En raison du temps de calcul élevé, nous mettons une limite de temps de 3H plutôt qu'un nombre maximum de trials. Parmis les 133 trials seules 43 restent complets (pas prune). Voici le resultat sur TensorBoard:



Ici les batchs size privilégier par Optuna sont < 20, le taux d'apprentissage majoritaire est lui aussi très bas < 0.002. le nombre de couches qui fonctionne bien est soit 2 ou 3. Pour les distinguer, et comprendre combien de neurone, il est favorable de mettre par couche, on fait un

zoom:



Zoom pour les modèles avec plus de 0.985 d'accuracy

On le voit clairement, les modèles à trois couches sont plus performant que ceux à deux couches Le bon nombre de neurones pour la première couche est plus élevé que pour la deuxième couche (au moins 99 neurones d'écart).

Ainsi pour le meilleur modèle, on obtient :

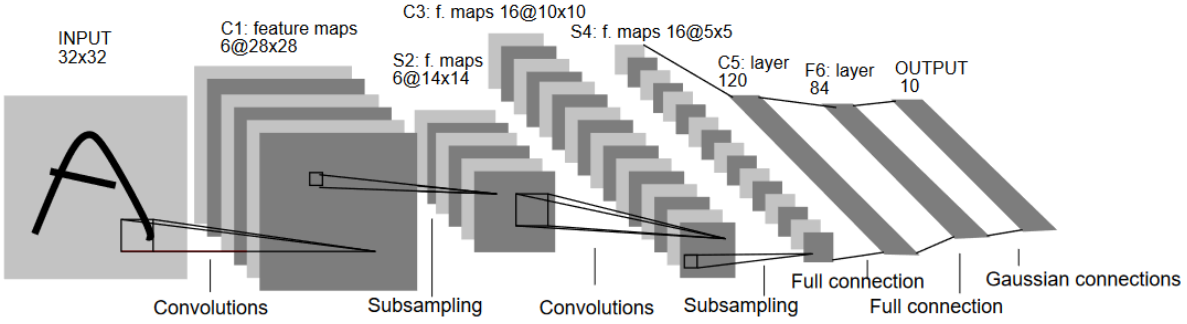
- taille du lot 4
- nombre de couches 3
- nombre de neurones première couche cachée 977
- nombre de neurones deuxième couche cachée 878
- taux d'apprentissage 0.000049640
- optimiser Adam

Le tout pour un score de 0.9869 soit 98,69%.

5. CNN

5.1. Le modèle

Le CNN (Convolutional Neural Network) est un réseau de neurones qui utilise des couches de convolution pour extraire automatiquement des caractéristiques pertinentes des données, souvent utilisées pour l'analyse d'images en raison de leur capacité à extraire des caractéristiques via les couches de convolution. Pour réaliser notre modèle, nous nous sommes basés sur LeNet5.



Architecture LeNet5

Notre architecture possède donc deux couches conventionnelles, deux couches entièrement connectées (full connection) et une sortie. On utilise également Relu comme fonction d'activation. Ensuite dans notre méthode forward, nous commençons par transformer notre vecteur de taille 784 en une image 28x28 puis nous appelons chaque couche dans cet ordre là : Convolution puis Relu, Maxpool, Convolution puis Relu, puis nouvelle transformation sur la donnée pour la rendre utilisable par les deux couches entièrement connecté.

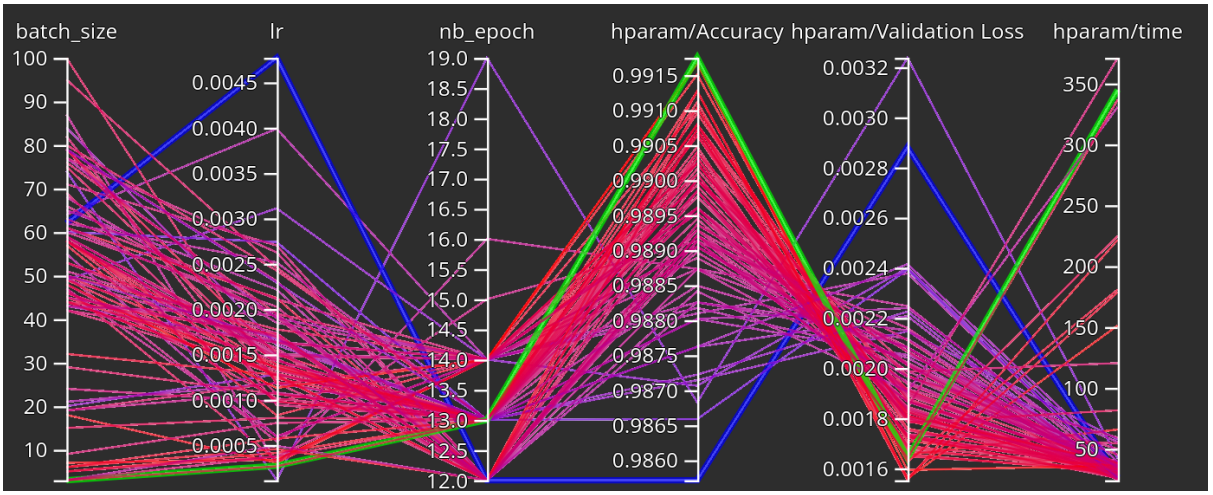
```
class Cnn(Module):

      def __init__(self, output_nbr: int):

      super(Cnn, self).__init__()
      self.conv1 = Conv2d(1, 6, kernel_size=(5, 5))
      self.conv2 = Conv2d(6, 16, kernel_size=(5, 5))

      self.fc1 = Linear(16 * 4 * 4, 120)
      self.fc2 = Linear(120, 84)

      self.output = Linear(84, output_nbr)

      def forward(self, x: Tensor) -> Tensor:
      x = x.view(-1, 1, 28, 28)

      x = F.relu(self.conv1(x))
      x = F.max_pool2d(x, kernel_size=(2, 2))
      x = F.relu(self.conv2(x))
      x = F.max_pool2d(x, kernel_size=(2, 2))
      x = x.view(-1, 16 * 4 * 4)
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
      output = self.output(x)

      return output
```

5.2. Automatique Hyper paramétrage

Nous faisons un run avec les paramètres suivants :

- taille du lot entre 3 et 100
- taux d'apprentissage entre 1e-5 et 1e-1
- optimiser ADAM ou SGD
- Trials 956 pour 3H de run

Ce coup-ci, on a prune 871 trials ce qui nous laisse 85 trials complets. On a réalisé beaucoup plus de trials en 3H que le MLP, on en déduit que le CNN est globalement plus rapide que le Mlp



Filtrer >0.98 d'accuracy pour y voir plus claire

Les meilleurs paramètres que nous avons eus sont :

- taille du lot 3
- taux d'apprentissage 0.0002
- optimiser ADAM

Le tout pour un score de 0.9917 soit 99,17%

6. Conclusion

En conclusion, ce projet a permis de comparer et d'analyser plusieurs modèles de réseaux de neurones, notamment le perceptron, le shallow network, le deep network (MLP), et le réseau de neurones convolutifs (CNN), en utilisant le jeu de données MNIST. À travers l'implémentation de ces différents modèles, nous avons pu mieux comprendre leur fonctionnement ainsi que l'influence des hyper-paramètres sur la performance.

Les tests d'hyper-paramétrage, que ce soit par grid-search ou par Optuna, ont montré l'importance cruciale du réglage des paramètres. Ce n'est pas une science exacte et trouver la bonne combinaison d'hyper paramètre est complexe. Les résultats ont démontré que les performances des modèles peuvent varier de manière significative en fonction de ces ajustements.

Pour approfondir, il serait intéressant de continuer cette étude comparative sur d'autres jeux de données afin de confirmer l'efficacité ou non d'un modèle. On peut aussi réfléchir à l'utilisation de ces modèles pré-entraîner sur MNIST et à les tester sur d'autres jeux de données (Domaine adaptation).

7. Sources

Nous sommes partis du cours, et avons utilisé la documentation de pytorch tout au long du projet :

1. https://pytorch.org/docs/stable/index.html

Pour Tensorboard :

2. https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams?hl=fr
3. https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

notre recherche sur optuna se base sur ses sources :

4. https://araffin.github.io/post/hyperparam-tuning/
5. https://github.com/optuna/optuna-examples/blob/main/pytorch/pytorch_simple.py
6. https://optuna.readthedocs.io/en/stable/
7. Optuna: A Next-generation Hyperparameter Optimization Framework
8. Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance