POLYTECHNIQUE MONTRÉAL

TP2
Intelligence artificielle : techniques probabilistes et d'apprentissage
INF8225

Groupe de laboratoire 01
Olivia Gélinas 1784417
Vincent Proulx-Meunier 1399037

École Polytechnique de Montréal
18 février 2018

Partie 1

a) (6 points) Donnez quelque pseudocode incluant des calculs matriciels — vectoriels détaillés pour l'algorithme de rétropropagation pour calculer le gradient pour les paramètres de chaque couche étant donné un exemple d'entraînement.

```
// Où X = [X_1, X_2, ... X_n, 1]
// W est une matrice de poids avec une colonne ajoutée à la fin pour représenter les biais
// W_l représente donc la matrice de poids associée à chaque L layer
// Il y a L layers et la couche de sortie est L+1
// alpha est le taux d'apprentissage
// in_l est la préactivation du layer l
// A_l est l'activation du layer l
// le réseau contient L couches cachées

// l'entrée est chargée dans le vecteur A d'entrée

A_l-1 <- X

POUR l=1 à L+1 FAIRE

       in_l <- (W_l)(A_l-1)
       A_l <- 1 / (1 + exp(-in_l))

// Soit Y la sortie désirée pour et A la sortie obtenue

delta <- ((1 / (1 + exp(-in_L+1)))*( 1 - 1 / (1 + exp(-in_L+1))))  * (Y-A_L+1)

POUR l=L à 1 FAIRE

       delta_l <-((1 / (1 + exp(-in_l)))*( 1 - 1 / (1 + exp(-in_l)))) * (W_l)(delta)
       W_l <- W_l+ (alpha)(A_l)(delta_l+1)
```

b) (4 points) Imaginez que vous avez maintenant un jeu de données avec N = 500 000 exemples. Expliquez comment vous utiliserez votre pseudocode pour optimiser ce réseau neuronal dans le contexte d'une expérience d'apprentissage machine correctement effectuée.

Tout d'abord, le pseudocode ci-dessus représente la portion « training » de l'algorithme et il est conçu pour un seul exemple et une seule epoch. Pour considérer 500 000 d'exemple, on pourrait soit exécuter une boucle du pseudocode pour tous les exemples, soit procéder avec des mini-batchs, ce qui ajouterait une dimension aux matrices (pour traiter plusieurs exemples à la fois). De plus, on voudra ajouter une boucle supplémentaire autour du pseudocode afin d'effectuer plus d'une epoch pour le training.

En considérant plusieurs exemples, il faudra en plus du training, ajouter une portion validation à notre expérience d'apprentissage machine. Une bonne méthode est de séparer les exemples en trois groupes. Par exemple, 50 000 exemples pour faire la validation (10%), 25 000 exemples pour les tests (5%) et 425 000 pour le training (85%). L'important est de conserver l'ensemble de test jusqu'à la toute fin, pour qu'on s'en serve pour faire une exécution finale sur des données non touchées pour une évaluation du modèle obtenu.

Une fois la phase training terminée, nous allons être en mesure de trouver les meilleurs paramètres (W et b) possibles en exécutant la phase de validation. L'idée est de prendre les meilleurs paramètres possibles et éviter le sur-apprentissage. Pour ce faire, il suffit de mesurer la meilleure précision sur les données de validation. Une façon additionnelle d'éviter le sur-apprentissage sera de faire du cross-validation sur nos données entre chaque epoch en partitionnant les données des ensembles de training et de validation.

Partie 2

Pour faire la partie 2, notre but était d'utiliser Pytorch afin de se familiariser avec cette librairie ainsi qu'avec des concepts de réseaux neuronaux. Nous avons principalement implémenté deux types de réseaux de neurones, des réseaux feedforward et des réseaux convolutionnels. Nous utilisions les données fournies par le fashion MNIST et nous avons essayé d'obtenir la précision la plus haute possible sur celles-ci. Par contre, le plus important n'était pas la précision obtenue mais plutôt la nature explorative du TP, donc nous présenterons plutôt les différents essais que nous avons effectués dans les lignes qui suivent. Nous avons débuté avec des essais sur des réseaux feedforward, en variant le nombre de neurones par couches, le nombre de couches cachées, ainsi que le taux d'apprentissage et le nombre d'epochs utilisé. Puis nous avons fait des essais avec des réseaux convolutionnels, en vérifiant les impacts de différents paramètres, comme la taille du kernel et le nombre de canaux. Nous présentons ici sommairement nos résultats. Les données brutes ainsi que le code utilisé sont disponibles dans le répertoire de projet.

Comparaisons initiales dans le réseau feedforward

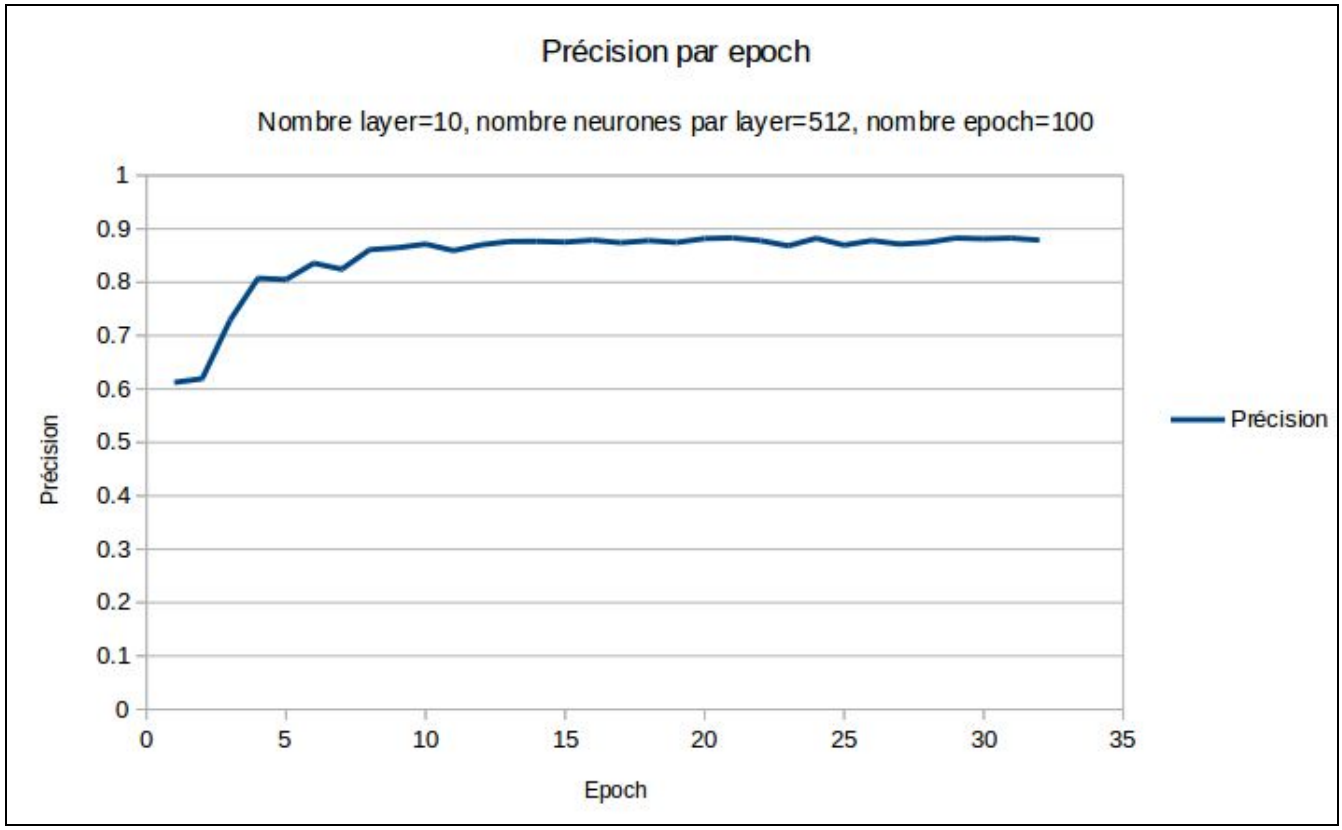
Tout d'abord, nous voulions tester nos intuitions que d'incrémenter le nombre d'epoch et le nombre de couches et de varier le taux d'apprentissage améliorerait nos résultats. Nous avons fait ces premières expériences en utilisant une fonction d'activation sigmoïde, tel que dans le modèle fourni en exemple. Nous avons créé des paramètres pour changer le nombre de layers et pour faire varier le nombre de neurones dans chaque layer, tel que dans la portion de code dans la figure qui suit.

```python
class NLayerSigmoidModel(nn.Module):
    def __init__(self, nNeurones, nLayer):
        super().__init__()
        self.nLayer = nLayer
        self.fc1 = nn.Linear(28 * 28, nNeurones)
        self.fcX = nn.Linear(nNeurones, nNeurones)
        self.fc2 = nn.Linear(nNeurones, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image.view(batch_size, -1)
        x = F.sigmoid(self.fc1(x))
        for i in range(0, self.nLayer-1):
            x = F.sigmoid(self.fcX(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x
```

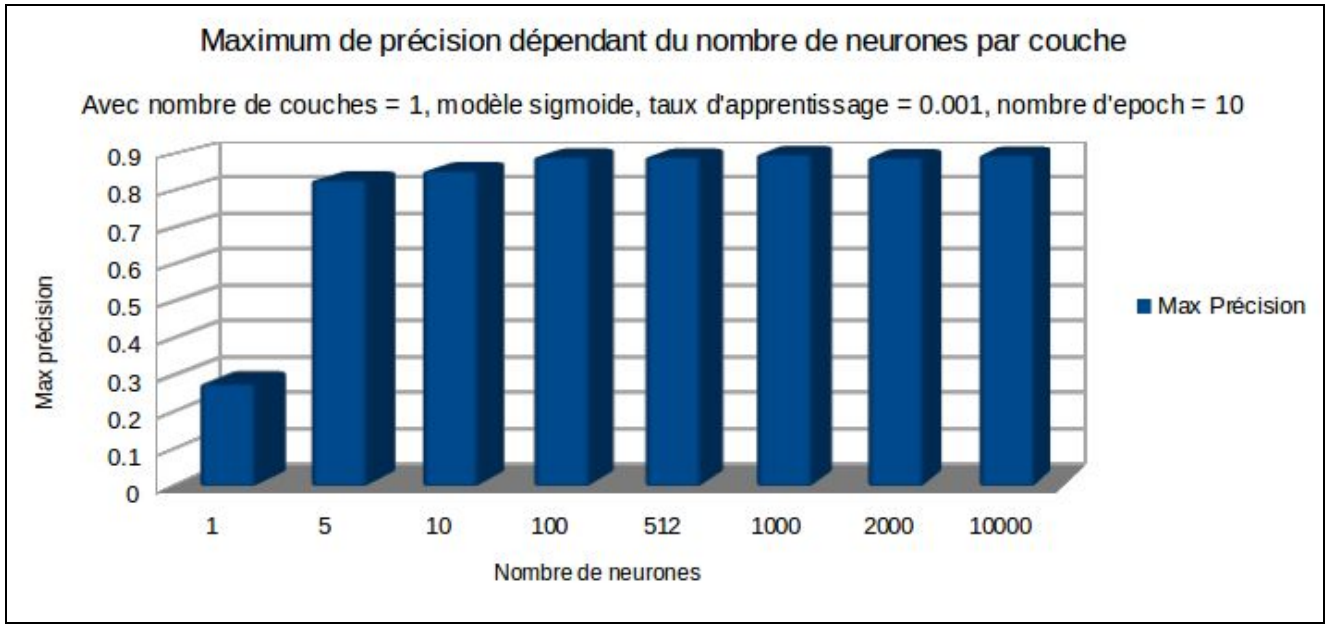
Pour utiliser ce modèle et les autres que nous avons créés pour les expériences, nous faisons comme dans la figure suivante, avec par exemple 5 layers de 512 neurones, 20 epochs et 0.001 de taux d'apprentissage.

```python
best_precision = 0
nNeurones = 512

for model in [NLayerSigmoidModel(nNeurones, 5)]:
    print('\n' + "DEBUT DES TEST POUR LE MODEL:")
    model = model.cuda()
    model, precision = experiment(model, 20, 0.001)
    if precision > best_precision:
        best_precision = precision
        best_model = model

test(best_model, test_loader)
```

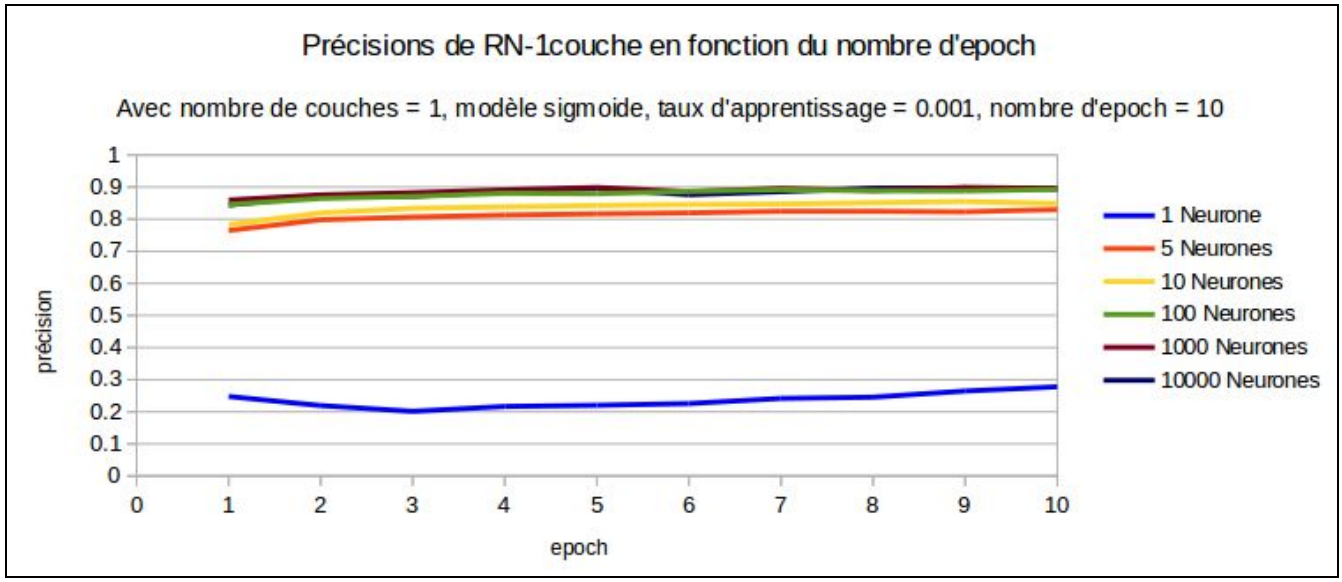
Ainsi, à travers nos expériences, nous avons trouvé que d'augmenter le nombre d'epochs augmente généralement la précision, mais que ce n'est pas nécessairement le cas pour le nombre de couches ou le taux d'apprentissage. Par exemple, nous avons débuté avec le modèle fourni en exemple, en ajoutant 10 layers utilisant une fonction d'activation sigmoïde, et avons vérifié sa performance à travers 30 epochs. Nous avons trouvé le résultat dans le graphe ci-dessous. On voit que dans les premières epochs, la précision augmente rapidement, puis que l'augmentation ralentit par la suite. Nous avons aussi observé cette tendance à travers nos autres expériences.



Puis, nous avons testé si le nombre de neurones par couche impactait la précision du modèle. Pour ce faire, nous avons utiliser un modèle Sigmoid à une seule couche et nous avons évalué la précision maximale pour différents nombres de neurones dans la couche. Nous avons gardé constants le nombre d'epochs (20) et le taux d'apprentissage (0.001). Vous trouverez les résultats obtenus dans le graphe ci-dessous. On s'aperçoit qu'il y a une augmentation relativement rapide de la précision lorsqu'on passe de 1 neurone à 100 neurones pour la couche cachée. Par la suite, la précision semble se stabiliser malgré les différences du nombre de neurones. Ainsi, pour la suite de l'analyse pour isoler cette variable, nous avons toujours pris un nombre de neurones égal à 512. Nous trouvions que ce nombre permettait une bonne précision sans que l'exécution du code ne prenne trop de temps, étant donné que notre but était plutôt exploratif.



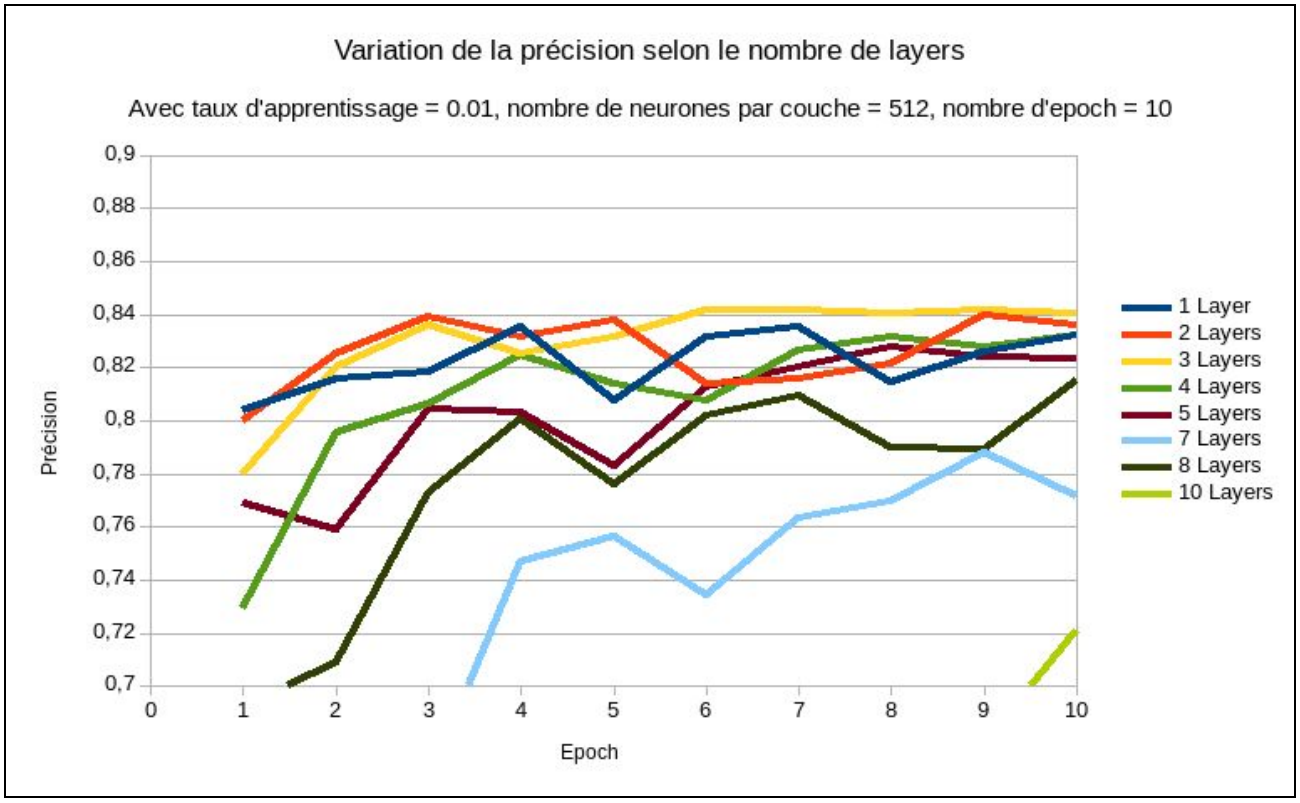
La prochaine figure confirme nos résultats présentés précédemment. Lorsqu'on regarde comment la précision évolue à travers les epochs selon le nombre de neurones, on voit que le pire résultat est obtenu avec 1 neurone, mais que la distinction entre 5, 10, 100, 1000 et 10 000 neurones n'est pas toujours aussi claire, même si en général la précision augmente avec le nombre d'epochs.

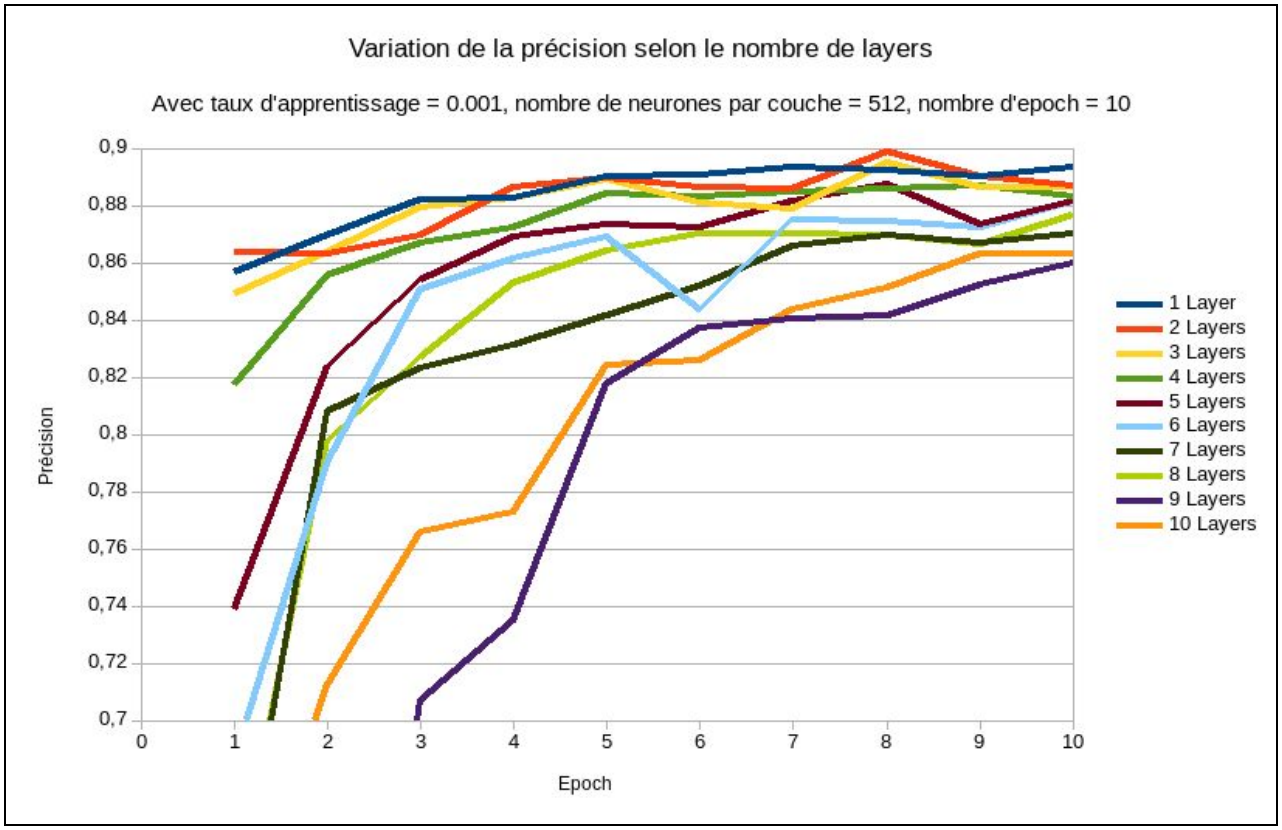


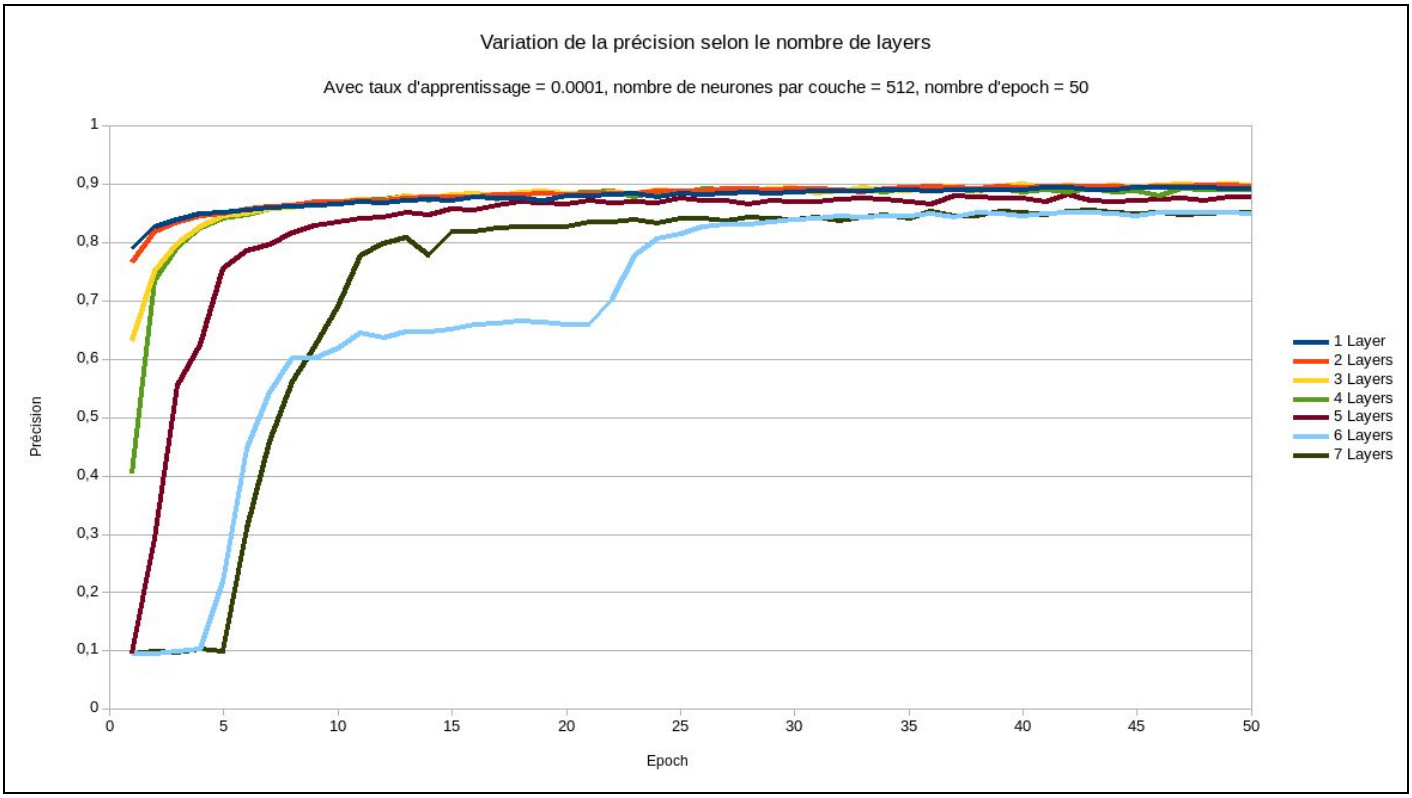
Une fois le nombre de neurones fixé, nous avons testé l'effet que peut avoir le taux d'apprentissage et le nombre de couches pour un même modèle (N-Layer-Sigmoid). Nous avons donc mesuré la précision pour des nombres de couches allant de 1 à 7. Les trois graphes ci-dessous illustrent respectivement les résultats obtenus pour des taux d'apprentissage de 0.01, 0.001 et 0.0001. Le quatrième graphe est seulement un zoom du troisième graphe afin de mieux voir les différentes courbes.

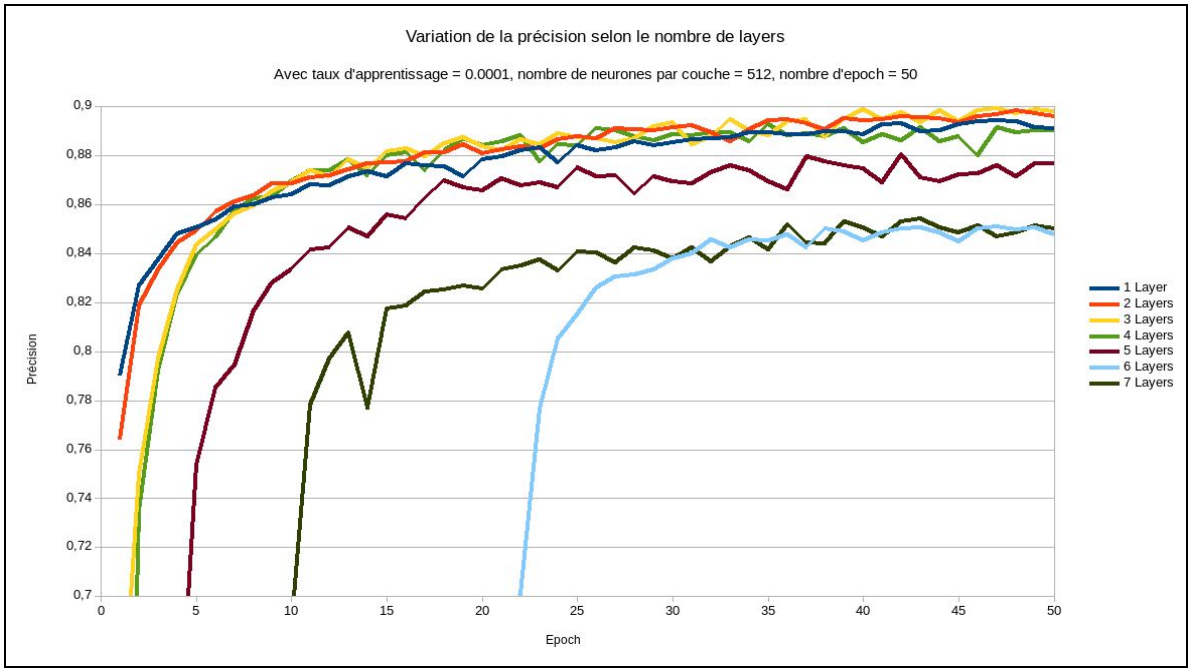
Concernant les taux d'apprentissage, on peut constater que le taux de 0.01 est moins efficace que les deux autres. Pour les taux de 0.001 et 0.0001, on obtient des résultat assez similaire, mais le taux de 0.0001 prend plus d'epochs pour converger, tandis qu'il semble obtenir des précisions légèrement supérieures. Pour le reste de nos tests, nous avons utilisé un taux de 0.001 afin de diminuer le temps d'exécution des tests. De plus, la différence avec le taux 0.0001 est assez négligeable pour la suite de nos tests.

Concernant le nombre de couches (1 à 7), nous remarquons que les modèle ayant 1 à 4 couches obtiennent des meilleurs résultats. Nous pensions que l'ajout de layers aurait mené vers des meilleurs résultats, mais nos démarches ont montré que ce n'est pas nécessairement le cas. Il semble y avoir une plage optimale de nombre de layers, et qu'à l'extérieur de cette plage de valeurs les résultats deviennent moins précis. Nous avons donc continué nos tests avec un nombre de couches entre 1 à 4, car nos observations suggéraient que cela apportait de meilleurs résultats et parce que moins de couches permettait un temps d'exécution plus raisonnable étant donné la portée du TP.




Variation de la précision selon le nombre de layers
Avec taux d'apprentissage = 0.001, nombre de neurones par couche = 512, nombre d'epoch = 10
Précision
Epoch
1 Layer
2 Layers
3 Layers
4 Layers
5 Layers
6 Layers
7 Layers
8 Layers
9 Layers
10 Layers


Variation de la précision selon le nombre de layers
Avec taux d'apprentissage = 0.0001, nombre de neurones par couche = 512, nombre d'epoch = 50
Précision
Epoch
1 Layer
2 Layers
3 Layers
4 Layers
5 Layers
6 Layers
7 Layers



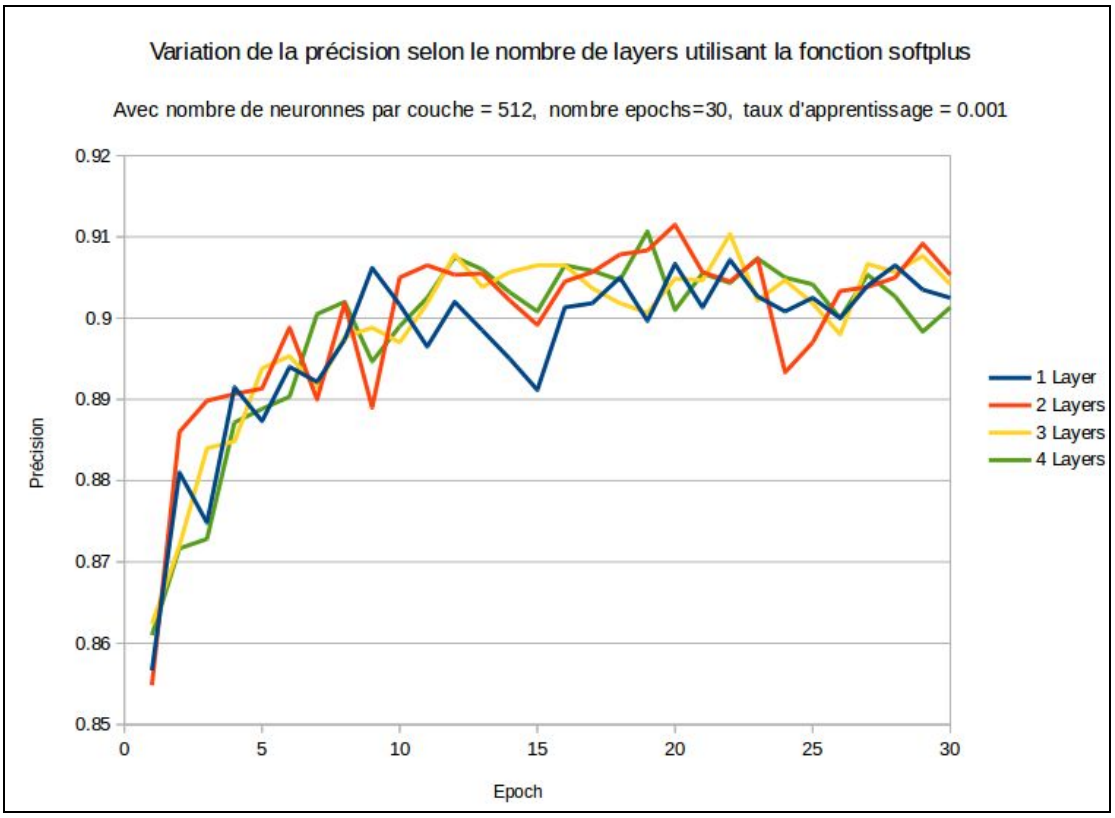
Comparaison des fonctions d'activation

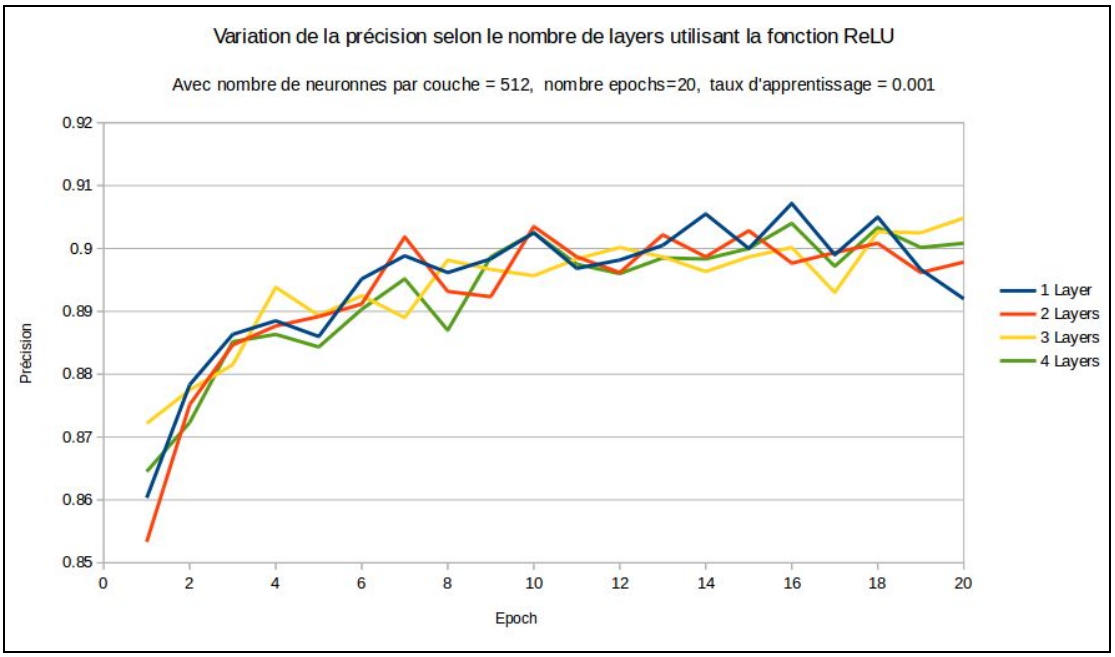
Après avoir testé différentes variables (le taux d'apprentissage, le nombre de couches et le nombre de neurones par couche), nous avons testé différentes fonctions d'activations, en fixant les autres variables. Dans la figure qui suit, nous montrons un exemple du code permettant d'utiliser la fonction tanh au lieu de sigmode. Le code est semblable pour les autres fonctions d'activation aussi.

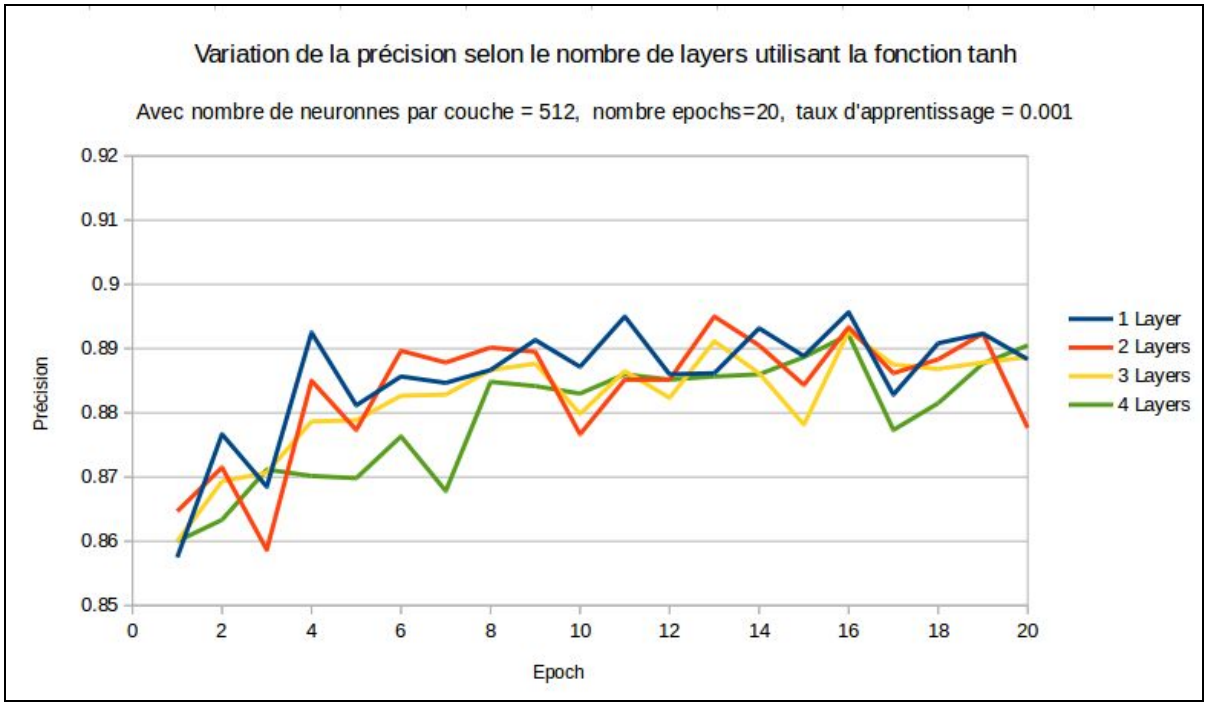
```
class NLayerTanhModel(nn.Module):
    def __init__(self, nNeurones, nLayer):
        super().__init__()
        self.nLayer = nLayer
        self.fc1 = nn.Linear(28 * 28, nNeurones)
        self.fcX = nn.Linear(nNeurones, nNeurones)
        self.fc2 = nn.Linear(nNeurones, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image.view(batch_size, -1)
        x = F.tanh(self.fc1(x))
        for i in range(0, self.nLayer-1):
            x = F.tanh(self.fcX(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x
```

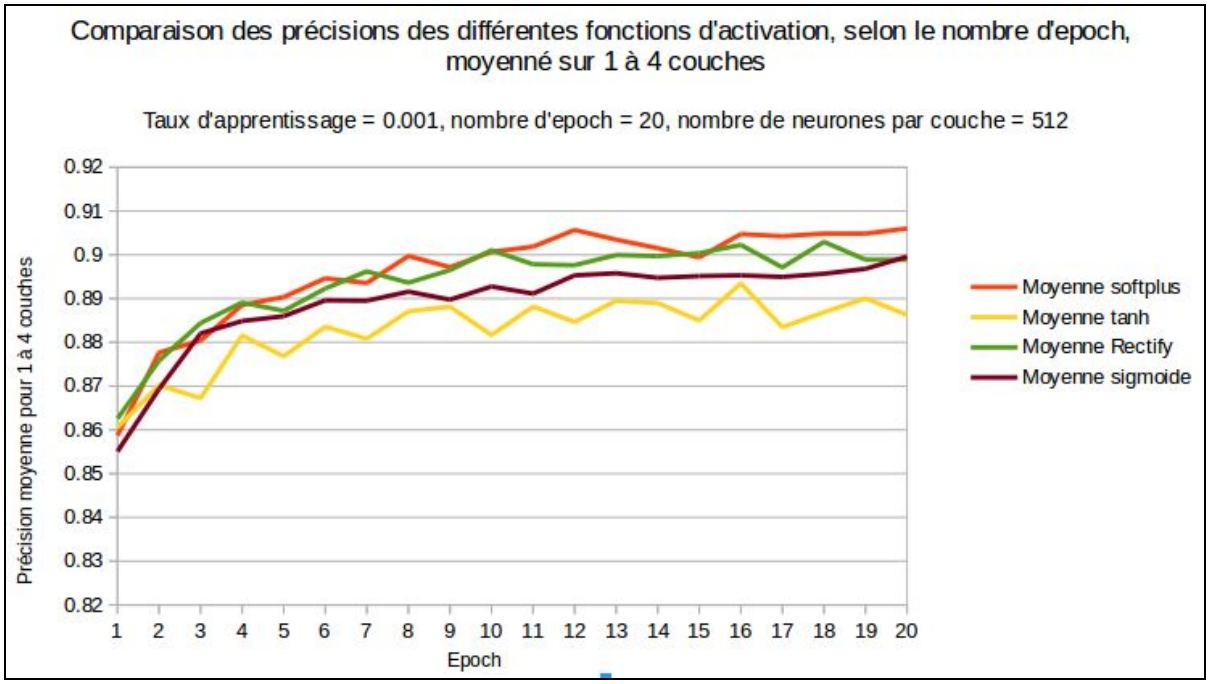
Les 3 graphes suivants représentent respectivement nos résultats pour les fonctions d'activation softplus, rectify (relu) et tanh. On ne voit pas clairement quel nombre de layers est nécessairement optimal, donc nous avons plutôt décidé de faire une moyenne sur la performance avec les différents nombres de layers pour comparer les différentes fonctions d'activation entre elles. Donc, le quatrième

graphe est une représentation moyenne des 4 fonctions d'activation testés jusqu'à maintenant, représentant leur performance moyennée avec 1, 2, 3 et 4 layers. De ces résultats, nous constatons que la fonction d'activation softplus se démarque des autres. En observant le graphe des moyennes, on constate que sa courbe de précision converge entre le 90% et 91%. Les autres se tiennent plutôt entre 88% et 90%.







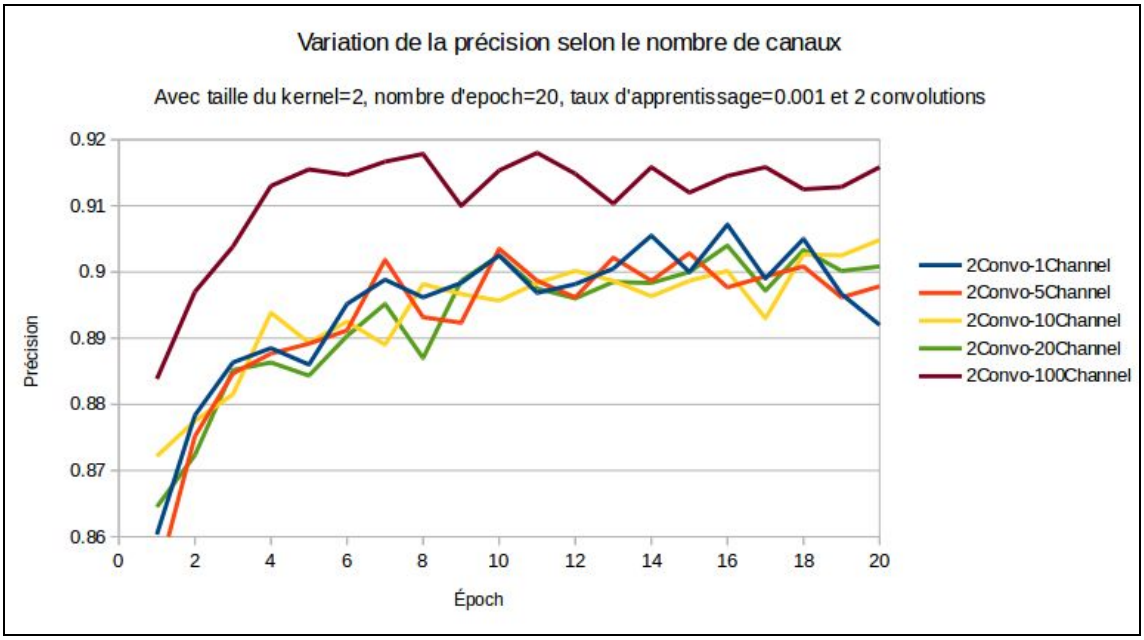


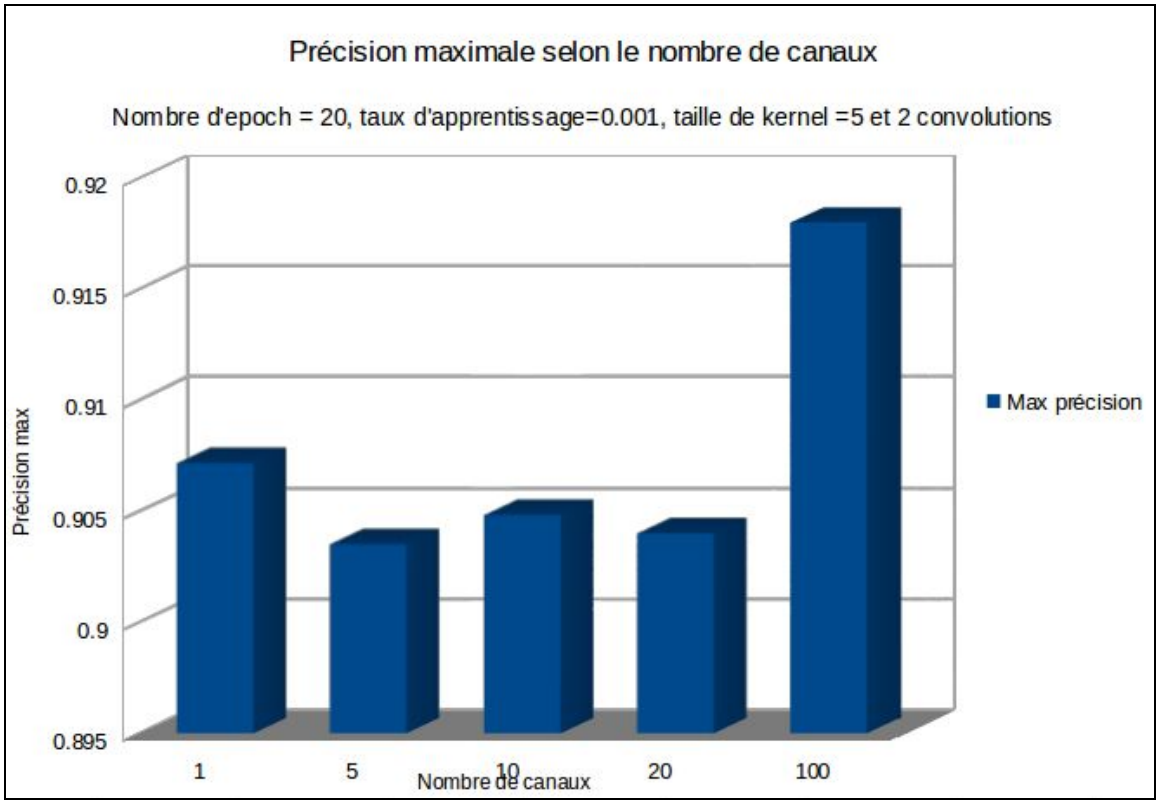
Comparaison de paramètres des réseaux convolutionnels

Après nos premiers essais, nous avons continué en testant différents paramètres de réseaux convolutionnels. Le code utilisé ressemblait à la portion du code qui suit, nous permettant de changer des paramètres comme le nombre de canaux sortant et la taille du kernel.

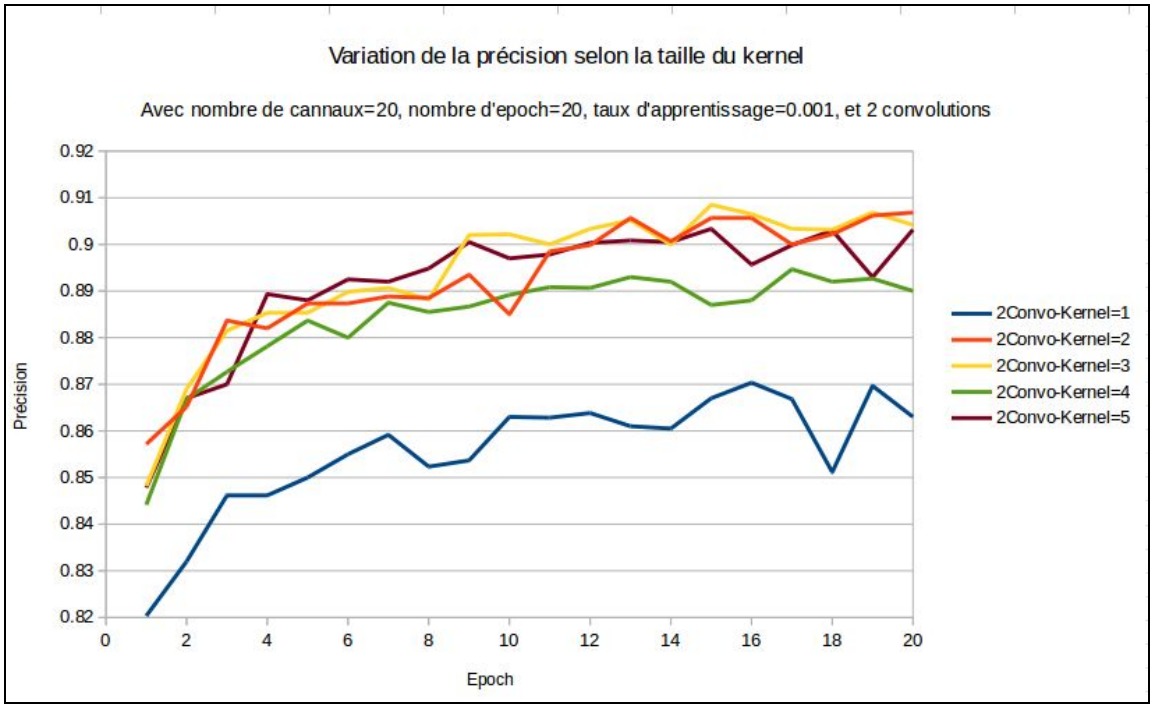
```python
# Conv
class ConvoModel(nn.Module):
    def __init__(self, nChannel, kernel_size):
        super(ConvoModel, self).__init__()
        self.conv1 = nn.Conv2d(1, nChannel, kernel_size)
        self.conv2 = nn.Conv2d(nChannel, nChannel, kernel_size)
        self.mp = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(2500, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1)
        x = F.log_softmax(self.fc1(x), dim=1)
        return x
```

Nous avons commencé en faisant des essais avec différents nombre de canaux. Puisque les images sont en en noir et blanc, nous avons toujours mis un seul canal entrant, puis nous avons varié le nombre de canaux sortant. Nous avons trouvé qu'en général, augmenter le nombre de canaux augmentait la performance du modèle, comme l'on voit dans les graphiques qui suivent. Le premier montre la variation de la précision selon le nombre de canaux à travers 20 epochs. Le deuxième montre la précision maximale atteinte sous les mêmes conditions.

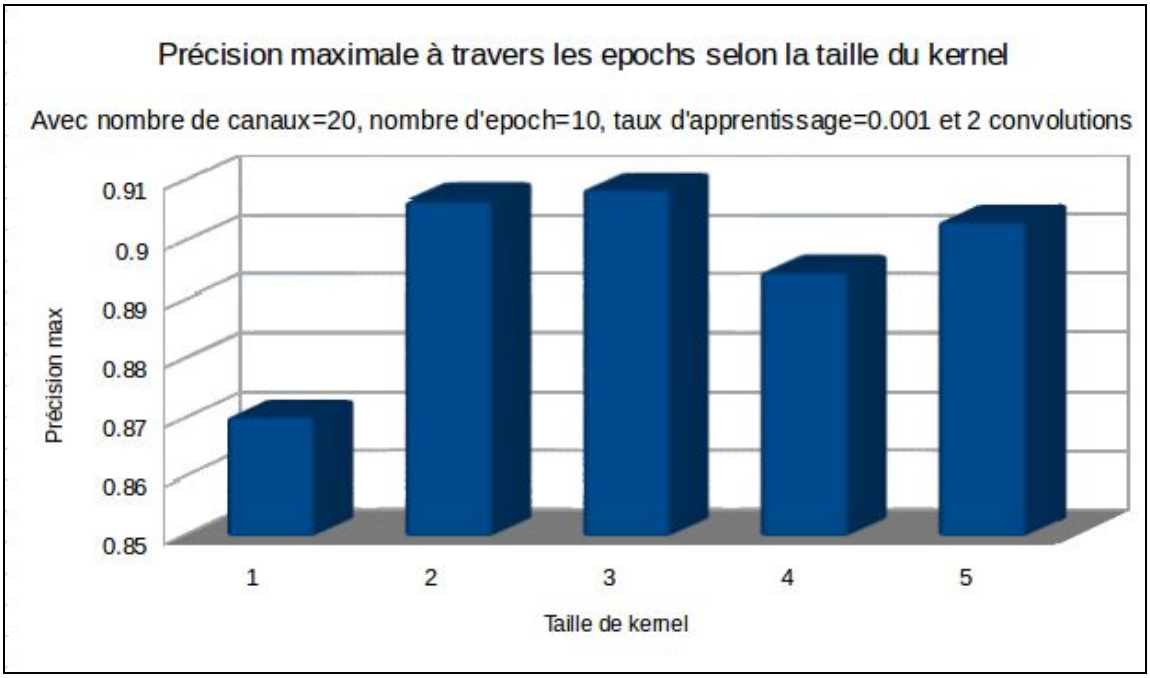




Par la suite, nous avons effectué des essais sur la taille du kernel utilisé pour la convolution. Nous avons donc fixé les autres variables, comme le nombre de canaux, le nombre d'epochs, le taux d'apprentissage et le nombre de convolutions. On voit sur les graphiques ci-dessous qu'un plus grand kernel ne mène pas toujours vers de meilleurs résultats. Par exemple, avec les paramètres dans le graphe ci-dessous, la taille de kernel de 3 a donné le plus de précision.



On voit une tendance semblable dans le graphe suivant, qui montre le maximum de précision atteint en 10 epochs, avec les valeurs de 2 et 3 suggérant les meilleures performances.



Résumé des meilleurs résultats sur l'ensemble test

Bref, avec les modèles feed forwards, nous avons généralement obtenu de meilleurs résultats avec la fonction d'activation softplus, avec un plus grand nombre d'epoch, et avec un nombre balancé de layers. Par contre, nous avons généralement obtenu des encore meilleurs résultats avec les réseaux convolutionnels. Pour cette raison, nous avons fait nos

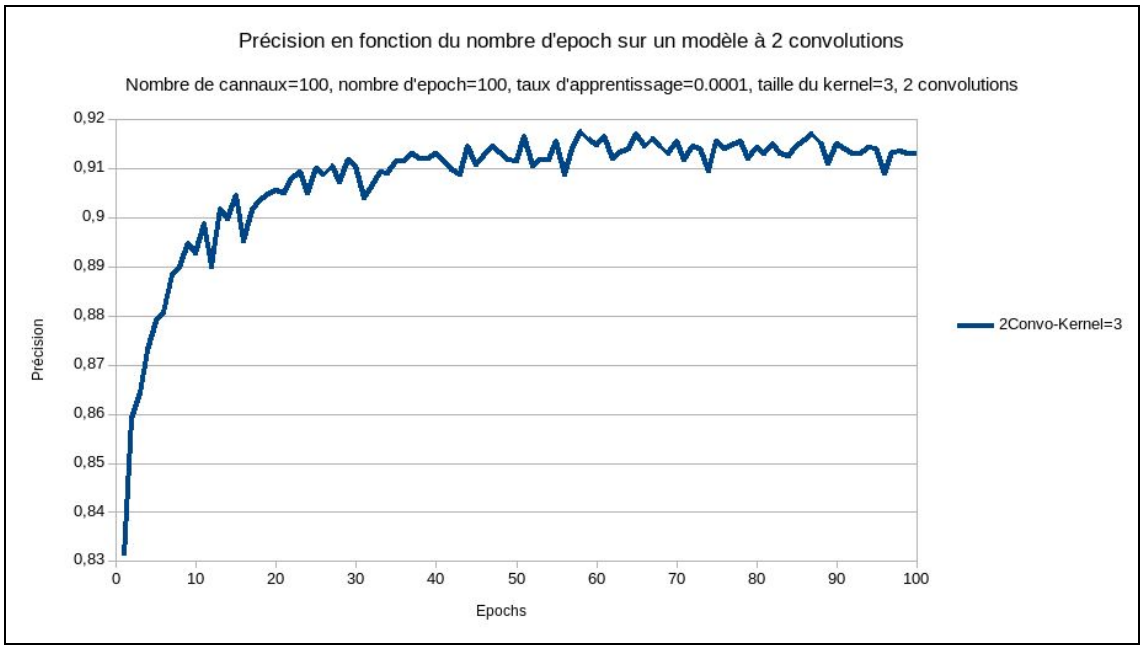
essais finaux avec différents paramètres dans des réseaux avec fonction d'activation softplus et avec convolutions. Nous consacrons cette section pour présenter les différents résultats de ces exécutions sur les ensembles de test. On en voit un apperçu dans le tableau suivant.

Résumé des résultats:

Paramètres	Performance sur l'ensemble de test
Modèle feedforward, fonction d'activation softplus, 2 layers, 512 neurones par couche, taux d'apprentissage à 0.001 et 30 epochs	Average loss: 0.6771, Accuracy: 8909/10000 (89%)
Modèle convolutionnel, 2 convolutions, 100 canaux, kernel de taille 3, taux d'apprentissage à 0.0001 et 100 epochs	Average loss: 0.3495, Accuracy: 9094/10000 (91%)
Modèle convolutionnel, 2 convolution,100 canaux, kernel de taille 5, taux d'apprentissage à 0.001 et 20 epochs	Average loss: 0.4112, Accuracy: 9102/10000 (91%)

Initialement, nous pensions que nous obtiendrions notre meilleur résultat avec un réseau convolutionnel avec un très grand nombre de canaux et un petit taux d'apprentissage avec beaucoup d'epochs, tel que dans la deuxième ligne du tableau. Par contre, en comparant les résultats dans le tableau, on voit que ce n'était pas le cas. Nous avons obtenu les meilleurs résultats avec les paramètres décrits dans la troisième ligne du tableau, qui avait le même nombre de canaux, mais moins d'epochs, un plus grand kernel et un plus grand taux d'apprentissage que dans la deuxième ligne. On voit qu'avec le premier essai avec un réseau convolutionnel on obtient un accuracy de 9094/10000, mais que le accuracy est légèrement meilleur dans le deuxième essai, c'est-à-dire 9094/10000. On remarque par contre que le loss est plus bas dans le premier essai. Dans la première rangée, on peut comparer ces résultats avec la performance du meilleur réseau feedforward conçu. Par contre, on voit que son loss et son accuracy sont pires que dans les meilleurs réseaux convolutionnels.

On voit à partir de ces résultats qu'il est difficile de prédire les résultats de paramètres choisis, même avec toutes nos observations empiriques. Dans le graphique qui suit, on présente le résultat sur l'exécution à 100 canaux sur 100 epochs, comme dans la deuxième rangée du tableau ci-dessus. On voit que la précision augmente rapidement dans les premières epochs, puis se stabilise plus tard. On pensait qu'avec un si grand nombre d'epochs, la précision serait plus élevée, mais elle semble se stabiliser autour d'environ 91-92%.



Remarques finales et notes sur les travaux futurs

Ce travail nous a permis d'explorer les réseaux neuronaux en utilisant Pytorch. Par manque de temps et de ressources computationnelles, il nous reste encore certains paramètres que nous voudrions tester dans un travail futur. Par exemple, il reste encore à explorer les effets du padding et de la modification du stride pour le modèle convolutionnel, ainsi que l'ajout au nombre de convolutions (pour l'instant nous nous sommes limités à deux convolutions). Il serait aussi intéressant d'explorer en plus de détail les effets des méthodes de preprocessing des données, comme la normalization et la translation ou la rotation des images utilisées. De plus, nous avons surtout considéré la qualité de nos modèles selon la précision qu'ils donnaient, mais dans le futur nous voudrions aussi examiner le loss en plus de détail, potentiellement en s'en servant pour des méthodes de early stopping pour éviter encore mieux le overfitting.