Rapport de laboratoire du TP2

Présenté à

Alexandre Piché

Par

Justine Pepin 1789244

Gr. 1 de laboratoire

Dans le cadre du cours de I.A. : tech. probabilistes et d'apprentissage

INF8225

18 février 2018

PARTIE I

a) Pseudocode de l'algorithme de rétropropagation pour le calcul du gradient des paramètres de toutes les couches d'un réseau de neurones avec un exemple d'entraînement

Caractéristiques du réseau :

- 1 couche en entrée D = 100 unités
- L couches cachées D = 100 unités
- 1 sortie continue comprise entre 0 et 1
- i = 1..N exemples dans l'ensemble d'apprentissage
- y_i = scalaire continu cible compris entre 0 et 1 pour l'exemple i
- $\mathbf{x}_i$ = le vecteur de D réels en entrée pour l'exemple i
- La fonction $f(x)$ d'activation de la couche finale est de forme sigmoïde
- La fonction $h^{(l)}(x)$ d'activation d'une couche cachée l est également de forme sigmoïde
- La fonction $a^{(l)}(x)$ de pré-activation de forme linéaire

Pour un exemple d'entraînement

// Évaluer la sortie de l'exemple
Pour chaque couche l de L couches cachées, faire :

$$In[l] = W[l] \times \boldsymbol{x}_i = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1D} \\ w_{21} & w_{22} & \cdots & w_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D1} & w_{D2} & \cdots & w_{DD} \end{bmatrix} \times \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iD} \end{bmatrix}$$

$$A[l] = h^{(l)}(In[l]) = \begin{bmatrix} 1/1 + e^{-ln_1} \\ 1/1 + e^{-ln_2} \\ \vdots \\ 1/1 + e^{-ln_D} \end{bmatrix}$$

// Avec In[l] qui représente un vecteur des entrées de toutes les unités d'une couche cachée. In[l] est de même taille que le vecteur d'entrée $\mathbf{x}_i$, lui-même d'une longueur D.

// Avec W[l] également qui représente les poids associés aux transitions entre 2 couches (entrée – cachée ou cachée – cachée). W est de la forme $L_X D_X D$, puisqu'il y a évidemment L de ces transitions. La dernière transition entre la dernière couche cachée et la sortie est faite avec un vecteur de poids de taille D et non pas une matrice puisqu'il n'y a qu'une sortie. Nous appellerons ce vecteur W_{sortie}.

// W est initialisé avec des poids quelconques qui se trouvent sous une courbe normale centrée à 0 comme dans le TP1.

// On peut également ajouter un biais pour calculer les entrées des unités en modifiant le vecteur d'entrée x_i afin de concaténer 1 à la suite du vecteur. Il faut aussi modifier la matrice W[l] des poids afin d'obtenir une matrice θ[l] qui s'écrit comme suit :

// $In[l] = \theta[l] \times \boldsymbol{x}_i = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{D1} \\ \vdots & \vdots & \dots & \vdots \\ w_{1D} & w_{2D} & \dots & w_{DD} \\ b_1 & b_2 & \dots & b_D \end{bmatrix}^T \times \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \\ 1 \end{bmatrix}$

// In[l] ne sera aucunement modifié par cet ajout. Il faut ensuite calculer A[l] le vecteur des sorties de chaque couche L. On doit se servir de la fonction sigmoïde d'activation $h^{(l)}(x)$ et l'appliquer de façon distincte sur chaque élément $a_{1..D}$ du vecteur A[l]. Le vecteur A[l] devient ensuite le vecteur d'entrée $\mathbf{x}_i$ de la couche suivante pour l'exécution de la boucle.

// Pour parvenir à la couche de sortie, utiliser W_{sortie} :

$$In_{sortie} = W_{sortie} \times \boldsymbol{x}_i = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{bmatrix}^T \times \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iD} \end{bmatrix}$$

// Comparer y_i la sortie cible de l'exemple i à $f(In_{sortie})$ la prévision du réseau jusqu'à maintenant.

$$\begin{aligned} \Delta_i = & \ f'(In_{sortie}) * \left(y_i - f(In_{sortie})\right) \\ & = f(In_{sortie}) * \left(1 - f(In_{sortie})\right) * \left(y_i - f(In_{sortie})\right) \end{aligned}$$

// Rétropropager le Δ à la couche précédente (la dernière couche cachée l = L). Ce calcul consiste à multiplier tous les éléments du vecteur de poids W_{sortie} par le Δ, puis par la dérivée de la fonction d'activation effectuée sur la couche précédente. Les multiplications sont bien évidemment faites élément par élément. Ensuite, réactualiser les poids de W_{sortie} en multipliant les sorties de la couche précédente contenues dans A[L – 1] par les quantités scalaires Δ et α et en additionnant le tout à W_{sortie}.

$$\Delta_j = \ A[L-1] * (1 - A[L-1]) * \Delta_i W_{sortie}$$

$$W_{sortie} = \ W_{sortie} + \alpha * \Delta_i A[L-1]$$

// Rétropropager aux autres couches.

Pour chaque couche l de L en commençant par L-2 jusqu'à 0, faire :

$$\Delta_k = \ A[l] * (1 - A[l]) * \left(W[l+1] \times \Delta_j\right)$$

$$W[l+1] = \ W[l+1] + \alpha * A[l] \times \Delta_j{}^T$$

$$\Delta_j = \ \Delta_k$$

// On devrait ainsi pouvoir reprendre le vecteur d'entrées $\mathbf{x}_i$ et évaluer la sortie du prochain exemple.

// Dans la dernière boucle pour le calcul de Δ, on utilise encore la dérivée de la fonction sigmoïde $h'(x) = h(x) * (1 – h(x))$. On fait également une multiplication matricielle entre

W[l + 1] et le Δ, désormais un vecteur car lorsque calculé une seconde fois pour passer de Δ_i à Δ_j, le Δ devient vectoriel (un élément de Δ par nœud d'une couche).

// Dans la dernière boucle pour le calcul de W[l + 1], on fait un produit matriciel extérieur entre A[l] et Δ qu'on transpose pour l'occasion. Finalement, l'addition entre le vecteur A[l] et la matrice W[l + 1] peut se faire élément par élément.

En résumé, sans les explications pour le calcul matriciel, on devrait se retrouver avec le pseudocode suivant pour un exemple :

$\boldsymbol{x}_{entrée} = \boldsymbol{x}_i$
Pour chaque couche l de L couches cachées, faire :
$In[l] = W[l] \times \boldsymbol{x}_{entrée}$
$A[l] = h^{(l)}(In[l])$
$x_{entrée} = A[l]$
$In_{sortie} = W_{sortie} \times \boldsymbol{x}_{entrée}$

// Commencer la rétropropagation

$\Delta_i = f(In_{sortie}) * \left(1 - f(In_{sortie})\right) * \left(y_i - f(In_{sortie})\right)$

$\Delta_j = A[L-1] * (1 - A[L-1]) * \Delta_i W_{sortie}$

$W_{sortie} = W_{sortie} + \alpha * \Delta_i A[L-1]$

Pour chaque couche l de L en commançant par L-2 jusqu'à 0, faire :

$\Delta_k = A[l] * (1 - A[l]) * (W[l+1] \times \Delta_j)$

$W[l+1] = W[l+1] + \alpha * A[l] \times \Delta_j{}^T$

$\Delta_j = \Delta_k$

// On devrait ainsi pouvoir reprendre le vecteur d'entrées $\mathbf{x}_i$ et évaluer la sortie du prochain exemple.

b) Optimisation du pseudocode pour un jeu de données de N = 500000 exemples.

Une bonne façon d'optimiser l'expérience d'apprentissage machine avec ce réseau de neurones serait de faire la descente de gradient stochastique avec des mini-batches comme dans le premier TP du cours. Pour ce faire, il faudrait former quelques batches en séparant les N exemples. Ensuite, il faudrait appliquer le pseudocode vu en a) et donc produire des sorties et rétropropager la différence sur la cible pour tous les exemples d'une batch. Au lieu de modifier les paramètres (poids et biais s'il y a lieu) sur-le-champ, il faudrait accumuler les erreurs et puis performer l'ajustement des paramètres à l'aide de la moyenne obtenue sur la batch. Ensuite seulement peut-on passer à la prochaine batch et répéter ces étapes.

PARTIE II

Note générale : La fonction d'aplanissement utilisée dans les 3 derniers essais est montrée dans l'extrait de code ci-dessous et a été empruntée à un tutoriel pour débutants de pytorch trouvé sur : http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

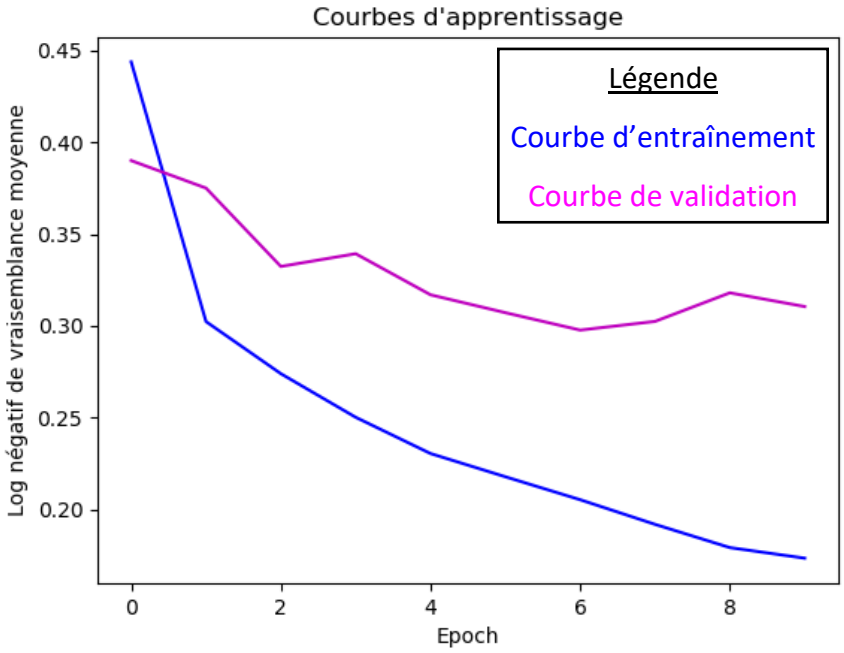
```
    # Fonction empruntee a :
http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.ht
ml
    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch
dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

II – Expérience 1

Pour la première expérience, l'architecture qui a été essayée se compose de 4 couches complètement connectées (*fully connected*) qui sont activées par des fonctions ReLUs, ce qui fait que les nœuds les employant sont dits *Rectified Linear Units*. Les couches sont progressivement réduites en taille, ce qui permet de doucement converger vers le nombre de classes (10) du Fashion MNIST. Voici l'implémentation utilisée :

```
class FcMultipleLayersNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 784)
        self.fc2 = nn.Linear(784, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image.view(batch_size, -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.log_softmax(self.fc4(x), dim=1)
        return x
```

Bien que cette implémentation ne diffère pas vraiment de l'expérience proposée dans le tutoriel du TP2, je voulais voir si plus de couches du même type changeait de façon positive le résultat et si la ReLU avait un gros impact sur notre entraînement. Les courbes d'apprentissage pour cet essai sont montrées dans le diagramme suivant :



On remarque que l'ensemble de validation peine à suivre l'ensemble d'entraînement en ce qui concerne les pertes. Le réseau est rapidement surentraîné sur l'ensemble d'entraînement et n'est pas aussi efficace lorsqu'il a affaire à la validation ou aux tests. Rien n'a vraiment changé en ce qui concerne la précision, évidemment :

```
valid set: Average loss: 0.3750, Accuracy: 5164/6000 (86%)

valid set: Average loss: 0.3324, Accuracy: 5245/6000 (87%)

valid set: Average loss: 0.3393, Accuracy: 5258/6000 (88%)

valid set: Average loss: 0.3169, Accuracy: 5315/6000 (89%)

valid set: Average loss: 0.3072, Accuracy: 5316/6000 (89%)

valid set: Average loss: 0.2977, Accuracy: 5323/6000 (89%)

valid set: Average loss: 0.3025, Accuracy: 5332/6000 (89%)

valid set: Average loss: 0.3180, Accuracy: 5332/6000 (89%)

valid set: Average loss: 0.3105, Accuracy: 5355/6000 (89%)

test set: Average loss: 0.3374, Accuracy: 8860/10000 (89%)
```

Afin d'avoir le cœur net à propos du fait d'augmenter le nombre de couches du même type, cet essai a été répété mais avec sept couches au lieu de quatre, dont voici le résultat :

```
valid set: Average loss: 1.9389, Accuracy: 1206/6000 (20%)

valid set: Average loss: 1.7189, Accuracy: 1245/6000 (21%)

valid set: Average loss: 1.7018, Accuracy: 1143/6000 (19%)

valid set: Average loss: 1.6989, Accuracy: 1174/6000 (20%)

valid set: Average loss: 1.7158, Accuracy: 1210/6000 (20%)

valid set: Average loss: 1.7005, Accuracy: 1207/6000 (20%)

valid set: Average loss: 1.7035, Accuracy: 1272/6000 (21%)

valid set: Average loss: 1.7187, Accuracy: 1132/6000 (19%)

valid set: Average loss: 1.5591, Accuracy: 1611/6000 (27%)

valid set: Average loss: 1.3524, Accuracy: 2066/6000 (34%)

test set: Average loss: 1.3547, Accuracy: 3463/10000 (35%)
```

On peut maintenant observer la baisse importante de précision sur les ensembles de validation et de test. Ce problème vient du fait que chaque nœud d'une couche est connecté à tous les nœuds de la couche suivante; ainsi, on est moins efficace avec les données structurées en plusieurs dimensions comme les images car on ne prend pas en compte que certaines caractéristiques peuvent se montrer uniquement en certains points de l'image. Un autre point négatif avec les couches complètement connectées est qu'on a énormément de poids à traîner pour tout le réseau (soit le nombre d'entrée au carré entre deux couches).

II – Expérience 2

Dans l'objectif de faire des essais plus fructueux ainsi que d'essayer plusieurs types de couches cachées, j'ai décidé d'implémenter une architecture existante qui performe bien sur le Fashion MNIST.

Cet essai a été inspiré par Xfan1025 (https://github.com/Xfan1025/Fashion-MNIST/blob/master/fashion-mnist.ipynb) qui a proposé une architecture pour atteindre 93% de précision sur le Fashion MNIST. Le modèle qu'il utilise va comme suit :

```
class FcMultipleLayersNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.norm1 = nn.BatchNorm2d(1)
        self.conv1 = nn.Conv2d(1, 64, 5)
        self.conv2 = nn.Conv2d(64, 512, 5)
        self.fc1 = nn.Linear(8192, 128)
        self.drp1 = nn.Dropout()
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image  # .view(batch_size, -1)
        x = F.relu(self.norm1(x))
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = self.drp1(F.relu(self.fc1(x)))
        x = self.drp1(F.relu(self.fc2(x)))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x
```

Dans son architecture, Xfan1025 utilise plusieurs types de couches. Tout d'abord, il normalise les données entrées. Normaliser les données permet que chaque caractéristique ait une influence équivalente sur le résultat final, malgré les éventuelles différences d'échelles de mesure pour les caractéristiques des données entrantes (http://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html).

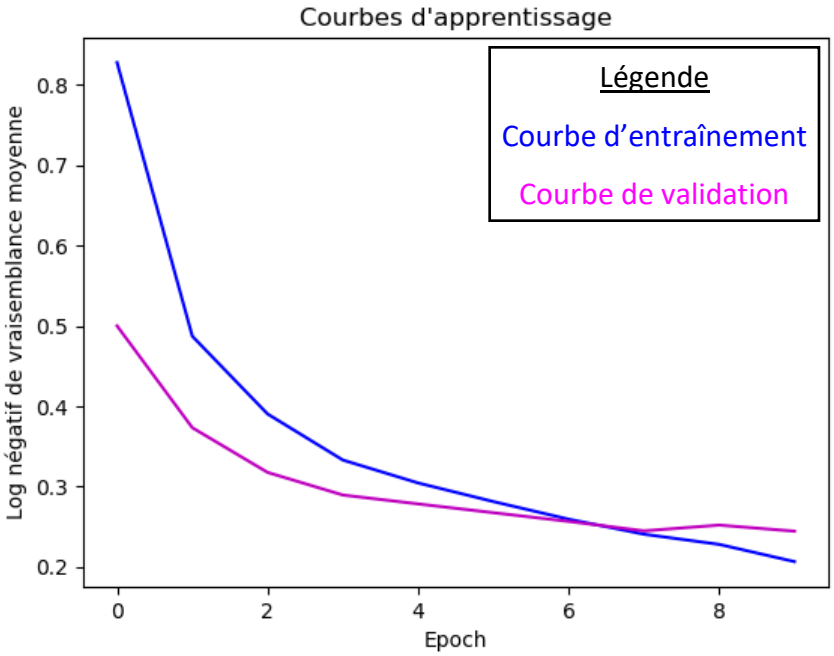
Ensuite, il les fait passer à travers deux couches convolutives, toutes deux immédiatement suivies d'une couche de mise en commun (*pooling*). Les couches convolutives permettent d'associer les caractéristiques de l'image à une région de l'image en rendant les nœuds d'une couche sensibles à une section spécifique des données provenant de la couche précédente. On suit souvent les couches convolutives par des couches de *pooling* afin de prévenir le surentraînement qui ferait qu'on aurait une excellente performance sur l'ensemble d'entraînement, mais un résultat nettement moins bon sur l'ensemble de validation ou de test. Faire la mise en commun (*pooling*) de certains pixels équivaut à brouiller l'image de la couche précédente.

Finalement, il regroupe progressivement les données à l'aide de 2 couches complètement connectées (*fully connected*) afin de faire une fonction de *softmax()* sur 10 groupes. Ces deux couches complètement connectées sont suivies de fonctions de *Dropout()* qui laissent tomber certains nœuds afin de prévenir le réseau de se surspécifier sur son ensemble d'entraînement.

La différence entre l'implémentation de Xfan1025 et la mienne consiste en le nombre d'epochs utilisé (10 pour moi contre 30 pour lui) ainsi que la taille de l'ensemble d'entraînement (54000 pour moi contre 51000 pour lui). Autre fait notable, Xfan1025 utilise TensorFlow avec Keras, tandis que j'ai emprunté l'implémentation modèle du TP (https://github.com/AlexPiche/INF8225/blob/master/tp2/Pytorch%20tutorial.ipynb).

J'ai décidé d'utiliser un plus petit nombre d'epochs que ce qui était montré dans l'implémentation de Xfan1025 à cause de la durée prolongée (> 1 heure) de l'exécution du script. Au lieu d'obtenir un résultat de 93%, j'ai plutôt obtenu une précision de 91% sur l'ensemble de test.

Le diagramme suivant montre les pertes sur l'ensemble d'entraînement et de validation durant l'apprentissage.



On peut voir dans ce diagramme que la validation avait tout d'abord moins de pertes que l'entraînement, ce qui s'est inversé durant les 3 dernières epochs. Une explication plausible à ce phénomène est que beaucoup de couches visant à rendre le réseau résistant au changement dans les données d'entrée, comme la normalisation ou encore les fonctions de *Dropout()* ont été appliquées. Conséquemment, les poids ont mis plus de temps à s'ajuster correctement à l'ensemble d'entraînement, ce qui a pesé lourd dans la balance pour les vérifications de perte durant les premières epochs.

La capture d'écran suivante montre le résultat de précision affiché dans la console suite à l'exécution de tout le programme :

```
valid set: Average loss: 0.5002, Accuracy: 4844/6000 (81%)

valid set: Average loss: 0.3735, Accuracy: 5186/6000 (86%)

valid set: Average loss: 0.3178, Accuracy: 5296/6000 (88%)

valid set: Average loss: 0.2896, Accuracy: 5338/6000 (89%)

valid set: Average loss: 0.2786, Accuracy: 5393/6000 (90%)

valid set: Average loss: 0.2676, Accuracy: 5421/6000 (90%)

valid set: Average loss: 0.2568, Accuracy: 5429/6000 (90%)

valid set: Average loss: 0.2450, Accuracy: 5465/6000 (91%)

valid set: Average loss: 0.2522, Accuracy: 5457/6000 (91%)

valid set: Average loss: 0.2446, Accuracy: 5478/6000 (91%)
```

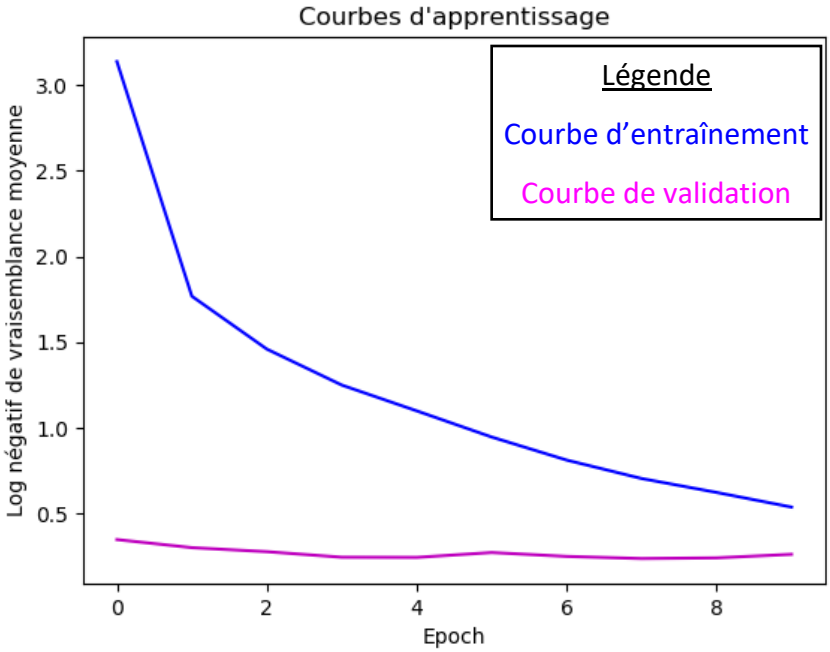
Comme mentionné plus tôt, la précision sur l'ensemble de test était ici de 91%. Bien sûr, il aurait été pertinent d'allonger un peu la durée d'exécution pour reproduire plus fidèlement l'expérience de Xfan1025, cependant j'ai préféré dans les essais suivants faire des variantes de son algorithme afin de voir l'impact, en termes de précision sur l'ensemble de test, de d'autres paramètres.

II – Expérience 3

Afin de continuer sur la lancée de l'expérience précédente, j'ai décidé de conserver la même implémentation, mais de ne pas utiliser les fonctions de *Dropout()* pour vérifier si les précautions prises pour empêcher le surentraînement ne sont pas un peu trop excessives dans notre cas. Le modèle utilisé va comme suit :

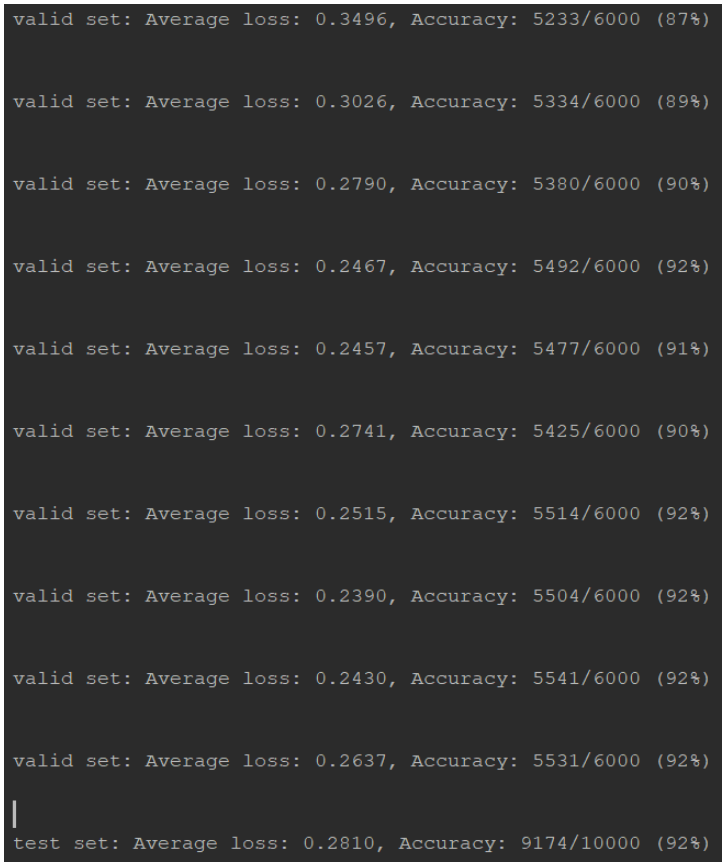
```
class FcMultipleLayersNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.norm1 = nn.BatchNorm2d(1)
        self.conv1 = nn.Conv2d(1, 64, 5)
        self.conv2 = nn.Conv2d(64, 512, 5)
        self.fc1 = nn.Linear(8192, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image  # .view(batch_size, -1)
        x = F.relu(self.norm1(x))
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x
```

Une différence d'architecture que j'ai introduite par rapport à celle de Xfan1025 est de faire de plus petites mini-batches (à 100 exemples pour moi au lieu de 256 pour lui). Dans le premier TP, les expériences avec différentes tailles de mini-batches montraient que les pertes sur l'ensemble de validation suivent moins les pertes sur l'ensemble d'entraînement lorsqu'on a de plus petites mini-batches, mais que la précision sur l'ensemble de validation augmente plus vite (car on met à jour les poids plus souvent). Cet effet a encore pu être observé, car les premières précisions sur l'ensemble de validation sont plus proches de la précision finale observée dans cet essai (87% à 92%) que dans l'essai précédent (81% à 91%). Par contre, je ne sais pas si cette meilleure précision au départ vient de la taille réduite des mini-batches ou de l'absence des fonctions de *Dropout()*. Le diagramme suivant montre les pertes sur l'ensemble d'entraînement et de validation durant l'apprentissage.



Il aurait été intéressant de prolonger quelque peu les epochs ici; en effet, on peut voir que la perte sur l'ensemble d'entraînement n'a pas encore croisée celle de l'ensemble de validation. Il aurait été intéressant de voir à quelle epoch se serait produit ce phénomène, surtout parce que dans l'essai précédent cette croisée survient avant l'epoch où Xfan1025 a atteint 93% de précision.

Les précisions sur l'ensemble de validation de cet essai sont dans la capture d'écran suivante :



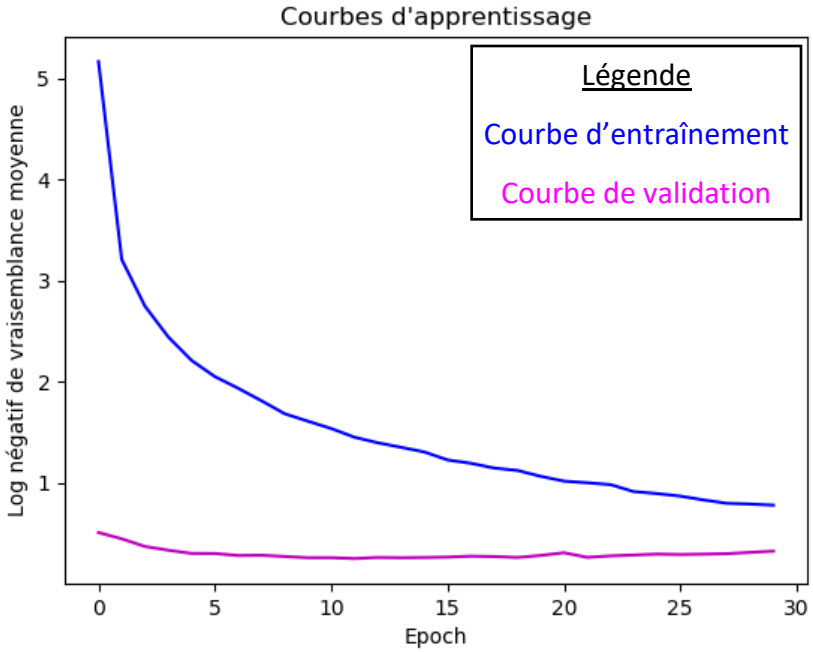
Pour cet essai, il a été possible d'obtenir 92% de précision sur l'ensemble de test. Ce chiffre est intéressant puisqu'il porte à penser qu'Xfan1025 aurait pu obtenir 93% également en faisant de plus petites mini-batches et un peu moins de 30 epochs sans utiliser les fonctions de *Dropout()*. Pour un essai ultérieur, il serait intéressant d'essayer cette stratégie et de comparer le résultat à celui d'Xfan1025.

II – Expérience 4

Pour cette dernière expérience, j'hésitais entre refaire l'expérience précédente avec plus d'epochs ou faire l'expérience d'Xfan1025 avec de plus petites mini-batches. J'ai choisi cette dernière option; cependant, ce n'est pas tout à fait l'expérience d'Xfan1025 puisque j'ai conservé la taille de l'ensemble d'entraînement à 54000 exemples. Les mini-batches sont de grandeur 100 exemples pour moi et 256 exemples pour lui.

```
class FcMultipleLayersNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.norm1 = nn.BatchNorm2d(1)
        self.conv1 = nn.Conv2d(1, 64, 5)
        self.conv2 = nn.Conv2d(64, 512, 5)
        self.fc1 = nn.Linear(8192, 128)
        self.drp1 = nn.Dropout()
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, image):
        batch_size = image.size()[0]
        x = image  # .view(batch_size, -1)
        x = F.relu(self.norm1(x))
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = self.drp1(F.relu(self.fc1(x)))
        x = self.drp1(F.relu(self.fc2(x)))
        x = F.log_softmax(self.fc3(x), dim=1)
        return x
```

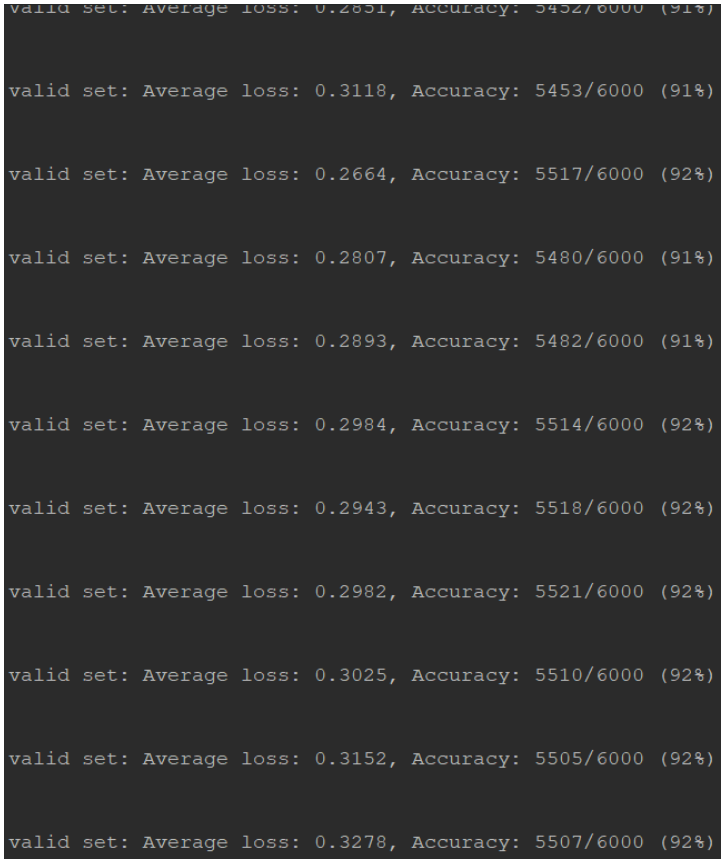
Le diagramme suivant montre les pertes sur l'ensemble d'entraînement et de validation durant l'apprentissage.



Les précisions sur l'ensemble de validation de cet essai sont dans la capture d'écran suivante :

```
valid set: Average loss: 0.5118, Accuracy: 4725/6000 (79%)

valid set: Average loss: 0.4496, Accuracy: 4964/6000 (83%)

valid set: Average loss: 0.3737, Accuracy: 5165/6000 (86%)

valid set: Average loss: 0.3368, Accuracy: 5262/6000 (88%)

valid set: Average loss: 0.3052, Accuracy: 5366/6000 (89%)

valid set: Average loss: 0.3046, Accuracy: 5335/6000 (89%)

valid set: Average loss: 0.2845, Accuracy: 5399/6000 (90%)

valid set: Average loss: 0.2871, Accuracy: 5406/6000 (90%)

valid set: Average loss: 0.2748, Accuracy: 5419/6000 (90%)

valid set: Average loss: 0.2625, Accuracy: 5450/6000 (91%)
```

```
valid set: Average loss: 0.2625, Accuracy: 5450/6000 (91%)

valid set: Average loss: 0.2627, Accuracy: 5454/6000 (91%)

valid set: Average loss: 0.2545, Accuracy: 5469/6000 (91%)

valid set: Average loss: 0.2650, Accuracy: 5451/6000 (91%)

valid set: Average loss: 0.2624, Accuracy: 5479/6000 (91%)

valid set: Average loss: 0.2648, Accuracy: 5475/6000 (91%)

valid set: Average loss: 0.2689, Accuracy: 5482/6000 (91%)

valid set: Average loss: 0.2777, Accuracy: 5484/6000 (91%)

valid set: Average loss: 0.2742, Accuracy: 5484/6000 (91%)

valid set: Average loss: 0.2654, Accuracy: 5502/6000 (92%)

valid set: Average loss: 0.2851, Accuracy: 5452/6000 (91%)
```



Pour cet essai, il a été possible d'obtenir 92% de précision sur l'ensemble de test. On peut voir que la précision sur l'ensemble de validation passe de 79% à 92%, tandis que chez Xfan1025 elle va de 86% à 93%. Peut-être qu'une mise à jour trop fréquente à cause de la taille des mini-batches handicape cette architecture, puisque les fonctions de *Dropout()* causent tout de même un fort débalancement des poids au début de l'exécution qui est lent à corriger. Donc, par rapport aux deux exemples précédent, on voit bien que de combiner les *Dropout()* à de petites mini-batches influence bel et bien à partir de quelle précision on part après une epoch en réduisant cette précision.

CONCLUSION

En conclusion, il a été pertinent de s'inspirer d'une architecture déjà existante et qui performe assez bien sur le Fashion MNIST. Ainsi, j'ai pu comprendre pourquoi les couches cachées s'ensuivent dans un certain ordre et qu'est-ce que chacune d'elle permet de faire. J'ai aussi pu tester des variantes de cette architecture afin d'observer comment chaque paramètre influence le résultat final.

J'ai pu trouver l'algorithme de Xfan1025 en cherchant tout d'abord sur le Github de ceux qui ont fait le Fashion MNIST : https://github.com/zalandoresearch/fashion-mnist. Sur leur page, on peut trouver des liens vers les comptes Github d'autres gens qui ont conçu des architectures qui performent bien sur le Fashion MNIST. Si j'ai choisi celui de Xfan1025, c'est parce que je trouvais qu'il se situait au point d'équilibre entre la présence de couches que je voulais essayer, comme les couches convolutives, la normalisation ou les *Dropout()*, et du temps d'exécution requis (taille des paramètres à traîner plus restreinte, pas trop d'epochs, pas d'augmentation).

Par contre, j'en ai profité pour regarder les autres implémentations proposées. Celles qui fonctionnent vraiment bien (autour de 95% ou plus sur l'ensemble de test du Fashion MNIST) font toutes de l'augmentation sur l'ensemble d'entraînement. Généralement, la réflexion par rapport à l'horizontale semble populaire et semble bien fonctionner. Pour une expérience éventuelle qui améliorerait la performance sur l'ensemble de test, il serait donc pertinent de tenter des augmentations, et donc de travailler sur GPU, sans quoi le temps de traitement est beaucoup trop long.