INF8225 TP1 H25 (v2.0)

Alexandre - GÉLINAS / Matricule 2083465

Partie 3 réalisée: [seul(e)]

Date limite :

20h30 le 6 février 2025 (Partie 1 et 2)

20h30 le 20 février 2025 (Partie 3)

Remettez votre fichier Colab sur Moodle en 2 formats: **.pdf** ET **.ipynb**

Comment utiliser:

Il faut copier ce notebook dans vos dossiers pour avoir une version que vous pouvez modifier, voici deux façons de le faire:

- File / Save a copy in Drive ...
- File / Download .ipynb

Pour utiliser un GPU

Runtime / Change Runtime Type / Hardware Accelerator / GPU

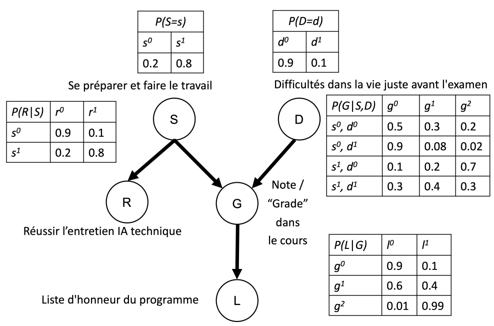
Partie 1 (16 points)

Objectif

L'objectif de la Partie 1 du travail pratique est de permettre à l'étudiant de se familiariser avec les réseaux Bayésiens et la librairie Numpy.

Problème

Considérons le réseau Bayésien ci-dessous.



Ceci représente un modèle simple pour les notes à un examen (G) et sa relation avec les étudiants qui se préparent aux examens et font correctement le travail pour les devoirs (S), les étudiants qui ont des difficultés dans la vie juste avant l'examen final (D), les étudiants qui réussissent bien à un entretien technique pour un emploi axé sur le sujet du cours (R), et des étudiants qui se retrouvent sur une sorte de palmarès de leur programme (L).

Trucs et astuces

Nous utiliserons des vecteurs multidimensionnels `5d-arrays` dont les `axes` représentent:

```
axe 0 : Se préparer (S)
axe 1 : Difficultés avant l'exam (D)
axe 2 : Réussir l'entretien technique (R)
axe 3 : Note dans le cours (Grade) (G)
axe 4 : Liste d'honneur (L)
```

Chaque `axe` serait de dimension `2` ou `3`:

```
Exemple pour S:
0 : s0
1 : s1

Exemple pour G:
0 : g0
1 : g1
2 : g2
```

Quelques point à garder en tête:

- Utiliser la jointe comme point de départ pour vos calculs (ne pas développer tous les termes à la main).
- Attention à l'effet du do-operator sur le graphe.
- L'argument "keepdims=True" de "np.sum()" vous permet conserver les mêmes indices.
- Pour un rappel sur les probabilités conditionelles, voir: https://www.probabilitycourse.com/chapter1/1_4_0_conditional_probability.php

1. Complétez les tables de probabilités ci-dessous

```
import numpy as np
np.set_printoptions(precision=5)

# Les tableaux sont bâtis avec les dimensions (S, D, R, G, L)
# et chaque dimension avec les probablités associées aux 2 ou 3 valeurs possibles ({0, 1} ou {0, 1, 2})

Pr_S = np.array([0.2, 0.8]).reshape(2, 1, 1, 1, 1) # Donné en exemple
Pr_D = np.array([0.9, 0.1]).reshape(1, 2, 1, 1, 1)
Pr_R_given_S = np.array([0.9, 0.1, 0.2, 0.8]).reshape(2, 1, 2, 1, 1)
Pr_G_given_SD = np.array([0.5, 0.3, 0.2, 0.9, 0.08, 0.02, 0.1, 0.2, 0.7, 0.3, 0.4, 0.3]).reshape(2, 2, 1, 3, 1)
Pr_L_given_G = np.array([0.9, 0.1, 0.6, 0.4, 0.01, 0.99]).reshape(1, 1, 1, 3, 2)
```

```
print (f"Pr(S)=\n{np.squeeze(Pr_S)}\n")
print (f"Pr(D)=\n{np.squeeze(Pr_D)}\n")
print (f"Pr(R|S)=\n{np.squeeze(Pr_R_given_S)}\n")
print (f"Pr(G|S,D)=\n{np.squeeze(Pr_G_given_SD)}\n")
print (f"Pr(L|G)=\n{np.squeeze(Pr_L_given_G)}\n")
```

```
Pr(S)=
[0.2 0.8]

Pr(D)=
[0.9 0.1]

Pr(R|S)=
[[0.9 0.1]
 [0.2 0.8]]

Pr(G|S,D)=
[[[0.5  0.3  0.2 ]
  [0.9  0.08 0.02]]

 [[0.1  0.2  0.7 ]
  [0.3  0.4  0.3 ]]]

Pr(L|G)=
[[0.9  0.1 ]
 [0.6  0.4 ]
 [0.01 0.99]]
```

2. À l'aide de ces tables de probabilité conditionnelles, calculez les requêtes ci-dessous. Dans les cas où l'on compare un calcul non interventionnel à un calcul interventionnel, commentez sur l'interprétation physique des deux situations et les résultats obtenus à partir de vos modèles.

a) $Pr(G) = [P (G = g^0), P (G = g^1), P (G = g^2)]$$Pr(G) = [P (G = g^0), P (G = g^1), P (G = g^2)]$

```
all =  Pr_S * Pr_D * Pr_R_given_S * Pr_G_given_SD * Pr_L_given_G

answer_a = all.sum(axis=(0, 1, 2, 4))
print(f"Pr(G)={answer_a}")
```

```
Pr(G)=[0.204  0.2316 0.5644]
```

b) $Pr(G|R = r^1)$$Pr(G|R = r^1)$

```
answer_b = all[:, :, 1, :, :].reshape(2, 2, 1, 3, 2).sum(axis=(0, 1, 2, 4)) / all[:, :, 1, :, :].sum()
print(f"Pr(G|R=r1)={answer_b}")
```

```
Pr(G|R=r1)=[0.13273 0.22176 0.64552]
```

c) $Pr(G|R = r^0)$$Pr(G|R = r^0)$

```
answer_c = all[:, :, 0, :, :].reshape(2, 2, 1, 3, 2).sum(axis=(0, 1, 2, 4)) / all[:, :, 0, :, :].sum()
print(f"Pr(G|R=r0)={answer_c}")
```

```
Pr(G|R=r0)=[0.34235 0.25071 0.40694]
```

d) $Pr(G|R=r^1, S=s^0)$$Pr(G|R=r^1, S=s^0)$

```
answer_d = all[0, :, 1, :, :].reshape(1, 2, 1, 3, 2).sum(axis=(0, 1, 2, 4)) / all[0, :, 1, :, :].sum()
print(f"Pr(G|R=r1, S=s0)={answer_d}")
```

```
Pr(G|R=r1, S=s0)=[0.54  0.278 0.182]
```

e) $Pr(G|R=r^0, S=s^0)$$Pr(G|R=r^0, S=s^0)$

```
answer_e = all[0, :, 0, :, :].reshape(1, 2, 1, 3, 2).sum(axis=(0, 1, 2, 4)) / all[0, :, 0, :, :].sum()
print(f"Pr(G|R=r0, S=s0)={answer_e}")
```

```
Pr(G|R=r0, S=s0)=[0.54  0.278 0.182]
```

f) $Pr(R|D=d^1)$$Pr(R|D=d^1)$

```
answer_f = all[:, 1, :, :, :].reshape(2, 1, 2, 3, 2).sum(axis=(0, 1, 3, 4)) / all[:, 1, :, :, :].sum()
print(f"Pr(R|D=d1)={answer_f}")
```

```
Pr(R|D=d1)=[0.34 0.66]
```

g) $Pr(R|D=d^0)$$Pr(R|D=d^0)$

```
answer_g = all[:, 0, :, :, :].reshape(2, 1, 2, 3, 2).sum(axis=(0, 1, 3, 4)) / all[:, 0, :, :, :].sum()
print(f"Pr(R|D=d0)={answer_g}")
```

```
Pr(R|D=d0)=[0.34 0.66]
```

h) $Pr(R|D=d^1, G=g^2)$$Pr(R|D=d^1, G=g^2)$

```
answer_h = all[:, 1, :, 2, :].reshape(2, 1, 2, 1, 2).sum(axis=(0, 1, 3, 4)) / all[:, 1, :, 2, :].sum()
print(f"Pr(R|D=d1, G=g2)={answer_h}")
```

```
Pr(R|D=d1, G=g2)=[0.21148 0.78852]
```

i) $Pr(R|D=d^0, G=g^2)$$Pr(R|D=d^0, G=g^2)$

```
answer_i = all[:, 0, :, 2, :].reshape(2, 1, 2, 1, 2).sum(axis=(0, 1, 3, 4)) / all[:, 0, :, 2, :].sum()
print(f"Pr(R|D=d0, G=g2)={answer_i}")
```

```
Pr(R|D=d0, G=g2)=[0.24667 0.75333]
```

j) $Pr(R|D=d^1, L=l^1)$$Pr(R|D=d^1, L=l^1)$

```
answer_j = all[:, 1, :, :, 1].reshape(2, 1, 2, 3, 1).sum(axis=(0, 1, 3, 4)) / all[:, 1, :, :, 1].sum()
print(f"Pr(R|D=d1, L=l1)={answer_j}")
```

```
Pr(R|D=d1, L=l1)=[0.2475 0.7525]
```

k) $Pr(R|D=d^0, L=l^1)$$Pr(R|D=d^0, L=l^1)$

```
answer_k = all[:, 0, :, :, 1].reshape(2, 1, 2, 3, 1).sum(axis=(0, 1, 3, 4)) / all[:, 0, :, :, 1].sum()
print(f"Pr(R|D=d1, L=l1)={answer_k}")
```

```
Pr(R|D=d1, L=l1)=[0.2736 0.7264]
```

l) $Pr(R|do(G=g^2))$$Pr(R|do(G=g^2))$

```
all_without_G =  Pr_S * Pr_D * Pr_R_given_S * Pr_L_given_G
answer_l = all_without_G[:, :, :, 2, :].reshape(2, 2, 2, 1, 2).sum(axis=(0, 1, 3, 4)) / all_without_G[:, :, :, 2, :].sum()
print(f"Pr(R|do(G=g2))={answer_l}")
```

```
Pr(R|do(G=g2))=[0.34 0.66]
```

m) $Pr(R|G=g^2)$$Pr(R|G=g^2)$

```
answer_m = all[:, :, :, 2, :].reshape(2, 2, 2, 1, 2).sum(axis=(0, 1, 3, 4)) / all[:, :, :, 2, :].sum()
print(f"Pr(R|G=g2)={answer_m}")
```

```
Pr(R|G=g2)=[0.24515 0.75485]
```

n) $Pr(R)$$Pr(R)$

```
answer_n = all.sum(axis=(0, 1, 3, 4))
print(f"Pr(R={answer_n}")
```

```
Pr(R=[0.34 0.66]
```

o) $Pr(G|do(L=l^1))$$Pr(G|do(L=l^1))$

```
all_without_L =  (Pr_S * Pr_D * Pr_R_given_S * Pr_G_given_SD).reshape(2, 2, 2, 3, 1)
answer_o = all_without_L[:, :, :, :, 0].reshape(2, 2, 2, 3, 1).sum(axis=(0, 1, 2, 4)) / all_without_L[:, :, :, :, 0].sum()
print(f"Pr(G|do(L=l1))={answer_o}")
```

```
Pr(G|do(L=l1))=[0.204  0.2316 0.5644]
```

p) $Pr(G=g^1|L=l^1)$$Pr(G=g^1|L=l^1)$

```
answer_p = all[:, :, :, 1, 1].reshape(2, 2, 2, 1, 1).sum() / all[:, :, :, :, 1].sum()
print(f"Pr(G=1|L=l1)={answer_p}")
```

```
Pr(G=1|L=l1)=0.13789900505510602
```

Réponse:

Partie 2 (20 points)

Objectif

L'objectif de la partie 2 du travail pratique est de permettre à l'étudiant de se familiariser avec l'apprentissage automatique via la régression logistique. Nous allons donc résoudre un problème de classification d'images en utilisant l'approche de descente du gradient (gradient descent) pour optimiser la log-vraisemblance négative (negative log-likelihood) comme fonction de perte.

L'algorithme à implémenter est une variation de descente de gradient qui s'appelle l'algorithme de descente de gradient stochastique par mini-ensemble (mini-batch stochastic gradient descent). Votre objectif est d'écrire un programme en Python pour optimiser les paramètres d'un modèle étant donné un ensemble de données d'apprentissage, en utilisant un ensemble de validation pour déterminer quand arrêter l'optimisation, et finalement de montrer la performance sur l'ensemble du test.

Théorie: la régression logistique et le calcul du gradient

Il est possible d'encoder l'information concernant l'étiquetage avec des vecteurs multinomiaux (one-hot vectors), c.-à-d. un vecteur de zéros avec un seul 1 pour indiquer quand la classe $C=k$$C=k$ dans la dimension $k$$k$. Par exemple, le vecteur $\mathbf{y}=[0, 1, 0, \cdots, 0]^T$$\mathbf{y}=[0, 1, 0, \cdots, 0]^T$ représente la deuxième classe. Les caractéristiques (features) sont données par des vecteurs $\mathbf{x}_i \in \mathbb{R}^{D}$$\mathbf{x}_i \in \mathbb{R}^{D}$. En définissant les paramètres de notre modèle comme : $\mathbf{W}=[\mathbf{w}_1, \cdots, \mathbf{w}_K]^T$$\mathbf{W}=[\mathbf{w}_1, \cdots, \mathbf{w}_K]^T$ et $\mathbf{b}=[b_1, b_2, \cdots b_K]^T$$\mathbf{b}=[b_1, b_2, \cdots b_K]^T$ et la fonction softmax comme fonction de sortie, on peut exprimer notre modèle sous la forme : \begin{eqnarray} p(\mathbf{y}|\mathbf{x}) &=& \frac{\exp(\mathbf{y}^T \mathbf{W} \mathbf{x} + \mathbf{y}^T \mathbf{b})}{\sum*{\mathbf{y}_k \in \mathscr{Y}} \exp(\mathbf{y}_k^T \mathbf{W} \mathbf{x} + \mathbf{y}_k^T \mathbf{b})} \end{eqnarray}\begin{eqnarray} p(\mathbf{y}|\mathbf{x}) &=& \frac{\exp(\mathbf{y}^T \mathbf{W} \mathbf{x} + \mathbf{y}^T \mathbf{b})}{\sum*{\mathbf{y}_k \in \mathscr{Y}} \exp(\mathbf{y}_k^T \mathbf{W} \mathbf{x} + \mathbf{y}_k^T \mathbf{b})} \end{eqnarray} L'ensemble de données consiste de $n$$n$ paires (label, input) de la forme $\mathscr{D}:=(\mathbf{\tilde{y}}\_i, \mathbf{\tilde{x}}\_i)*{i=1}^n$$\mathscr{D}:=(\mathbf{\tilde{y}}_i, \mathbf{\tilde{x}}_i)*{i=1}^n$, où nous utilisons l'astuce de redéfinir $\mathbf{\tilde{x}}_i = [\mathbf{\tilde{x}}_i^T 1]^T$$\mathbf{\tilde{x}}_i = [\mathbf{\tilde{x}}_i^T 1]^T$ et nous redéfinissions la matrice de paramètres $\boldsymbol{\theta} \in \mathbb{R}^{K\times(D+1)}$$\boldsymbol{\theta} \in \mathbb{R}^{K\times(D+1)}$ (voir des notes de cours pour la relation entre $\boldsymbol{\theta}$$\boldsymbol{\theta}$ et $\mathbf{W}$$\mathbf{W}$). Notre fonction de perte, la log-vraisemblance négative des données selon notre modèle est définie comme: \begin{equation} \mathscr{L}\big(\boldsymbol{\theta}, \mathscr{D} \big) := -\log \prod*{i=1}^N P(\mathbf{\tilde{y}}_i|\mathbf{\tilde{x}}_i; \boldsymbol{\theta}) \end{equation}\begin{equation} \mathscr{L}\big(\boldsymbol{\theta}, \mathscr{D} \big) := -\log \prod*{i=1}^N P(\mathbf{\tilde{y}}_i|\mathbf{\tilde{x}}_i; \boldsymbol{\theta}) \end{equation} Pour cette partie du TP, nous avons calculé pour vous le gradient de la fonction de perte par rapport par rapport aux paramètres du modèle: \begin{eqnarray} \frac{\partial}{\partial \boldsymbol{\theta}} \mathscr{L}\big(\boldsymbol{\theta}, \mathscr{D} \big) &=& -\sum*{i=1}^N \frac{\partial}{\partial \boldsymbol{\theta}} \Bigg\{\log \Bigg(\frac{\exp(\mathbf{\tilde{y}}_i^T \boldsymbol{\theta} \mathbf{\tilde{x}}_i)}{\sum_{\mathbf{y}_k \in \mathscr{Y}} \exp(\mathbf{y}_k^T \boldsymbol{\theta} \mathbf{\tilde{x}}_i)} \Bigg) \Bigg\} \\ &=& -\sum_{i=1}^N \left(\mathbf{\tilde{y}}_i \mathbf{\tilde{x}}^T_i- \sum_{\mathbf{y}_k \in \mathscr{Y}} P(\mathbf{y}_k|\mathbf{\tilde{x}}_i,\boldsymbol{\theta}) \mathbf{y}_k \mathbf{\tilde{x}}^T_i \right) \\ &=& \sum_{i=1}^N \mathbf{\hat{p}}_i \mathbf{\tilde{x}}^T_i - \sum_{i=1}^N \mathbf{\tilde{y}}_i \mathbf{\tilde{x}}^T*i \end{eqnarray}\begin{eqnarray} \frac{\partial}{\partial \boldsymbol{\theta}} \mathscr{L}\big(\boldsymbol{\theta}, \mathscr{D} \big) &=& -\sum*{i=1}^N \frac{\partial}{\partial \boldsymbol{\theta}} \Bigg\{\log \Bigg(\frac{\exp(\mathbf{\tilde{y}}_i^T \boldsymbol{\theta} \mathbf{\tilde{x}}_i)}{\sum_{\mathbf{y}_k \in \mathscr{Y}} \exp(\mathbf{y}_k^T \boldsymbol{\theta} \mathbf{\tilde{x}}_i)} \Bigg) \Bigg\} \\ &=& -\sum_{i=1}^N \left(\mathbf{\tilde{y}}_i \mathbf{\tilde{x}}^T_i- \sum_{\mathbf{y}_k \in \mathscr{Y}} P(\mathbf{y}_k|\mathbf{\tilde{x}}_i,\boldsymbol{\theta}) \mathbf{y}_k \mathbf{\tilde{x}}^T_i \right) \\ &=& \sum_{i=1}^N \mathbf{\hat{p}}_i \mathbf{\tilde{x}}^T_i - \sum_{i=1}^N \mathbf{\tilde{y}}_i \mathbf{\tilde{x}}^T*i \end{eqnarray} où $\mathbf{\hat{p}}_i$$\mathbf{\hat{p}}_i$ est un vecteur de probabilités produit par le modèle pour l'exemple $\mathbf{\tilde{x}}_i$$\mathbf{\tilde{x}}_i$ et $\mathbf{\tilde{y}}_i$$\mathbf{\tilde{y}}_i$ est le vrai _label* pour ce même exemple.

Finalement, il reste à discuter de l'évaluation du modèle. Pour la tâche d'intérêt, qui est une instance du problème de classification, il existe plusieurs métriques pour mesurer les performances du modèle la précision de classification, l'erreur de classification, le taux de faux/vrai positifs/négatifs, etc. Habituellement dans le contexte de l'apprentissage automatique, la précision est la plus commune.

La précision est définie comme le rapport du nombre d'échantillons bien classés sur le nombre total d'échantillons à classer:

$$ \tau_{acc} := \frac{|\mathscr{C}|}{|\mathscr{D}|} $$$$ \tau_{acc} := \frac{|\mathscr{C}|}{|\mathscr{D}|} $$

où l'ensemble des échantillons bien classés $\mathscr{C}$$\mathscr{C}$ est:

$$ \mathscr{C} := \lbrace (\mathbf{x}, \mathbf{y}) \in \mathscr{D} \, | \, \underset{k}{\arg\max} \, \, P(\cdot|\mathbf{\tilde{x}}_i; \boldsymbol{\theta})_k = \underset{k}{\arg\max} \, \, \tilde{y}_{i,k} \rbrace $$$$ \mathscr{C} := \lbrace (\mathbf{x}, \mathbf{y}) \in \mathscr{D} \, | \, \underset{k}{\arg\max} \, \, P(\cdot|\mathbf{\tilde{x}}_i; \boldsymbol{\theta})_k = \underset{k}{\arg\max} \, \, \tilde{y}_{i,k} \rbrace $$

En mots, il s'agit du sous-ensemble d'échantillons pour lesquels la classe la plus probable selon notre modèle correspond à la vraie classe.

Double-cliquez (ou appuyez sur Entrée) pour modifier

Description des tâches

1. Code à compléter

On vous demande de compléter l'extrait de code ci-dessous pour résoudre ce problème. Vous devez utiliser la librairie PyTorch cette partie du TP: https://pytorch.org/docs/stable/index.html. Mettez à jour les paramètres de votre modèle avec la descente par *mini-batch*. Exécutez des expériences avec trois différents ensembles: un ensemble d'apprentissages avec 90% des exemples (choisis au hasard), un ensemble de validation avec 10%. Utilisez uniquement l'ensemble de test pour obtenir votre meilleur résultat une fois que vous pensez avoir obtenu votre meilleure stratégie pour entraîner le modèle.

2. Rapport à rédiger

Présentez vos résultats dans un rapport. Ce rapport devrait inclure:

- **Recherche d'hyperparamètres:** Faites une recherche d'hyperparamètres pour différents taux d'apprentissage, e.g. 0.1, 0.01, 0.001, et différentes tailles de mini-batch, e.g. 1, 20, 200, 1000 pour des modèles entrainés avec SGD. Présentez dans un tableau la précision finale du modèle, sur l'*ensemble de validation*, pour ces différentes combinaisons d'hyperparamètres.
- **Analyse du meilleur modèle:** Pour votre meilleur modèle, présentez deux figures montrant la progression de son apprentissage sur l'*ensemble d'entrainement et l'ensemble de validation*. La première figure montrant les courbes de log-vraisemblance négative moyenne après chaque epoch, la deuxième montrant la précision du modèle après chaque epoch. Finalement donnez la précision finale sur l'ensemble de test.
- **Lire l'article de recherche - Adam**: a method for stochastic optimization. Kingma, D., & Ba, J. (2015). International Conference on Learning Representation (ICLR). https://arxiv.org/pdf/1412.6980.pdf. Implémentez Adam, répétez les deux étapes précédentes (recherche d'hyperparamètres et analyse du meilleur modèle) cette fois en utilisat Adam, et comparez les performances finales avec votre meilleur modèle SGD.

IMPORTANT

L'objectif du TP est de vous faire implémenter la rétropropagation à la main. **Il est donc interdit d'utiliser les capacités de construction de modèles ou de différentiation automatique de pytorch -- par exemple, aucun appels à torch.nn, torch.autograd ou à la méthode .backward().** L'objectif est d'implémenter un modèle de classification logistique ainsi que son entainement en utilisant uniquement des opérations matricielles de base fournies par PyTorch e.g. torch.sum(), torch.matmul(), etc.

Fonctions fournies

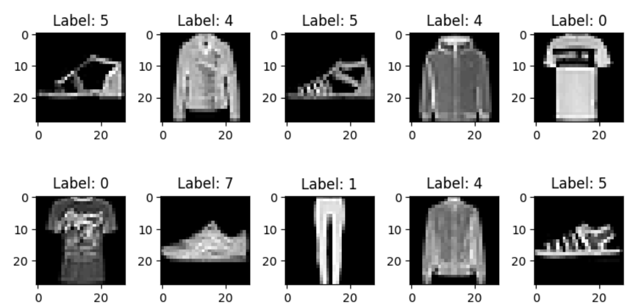
```
# fonctions pour charger les ensembles de donnees
from torchvision.datasets import FashionMNIST
from torchvision import transforms
import torch
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm
import matplotlib.pyplot as plt

def get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=1):
  dataset = FashionMNIST("./dataset", train=True,  download=True, transform=transforms.Compose([transforms.ToTensor()]))
  dataset_test = FashionMNIST("./dataset", train=False,  download=True, transform=transforms.Compose([transforms.ToTensor()]))
  len_train = int(len(dataset) * (1.-val_percentage))
  len_val = len(dataset) - len_train
  dataset_train, dataset_val = random_split(dataset, [len_train, len_val])
  data_loader_train = DataLoader(dataset_train, batch_size=batch_size,shuffle=True,num_workers=4)
  data_loader_val   = DataLoader(dataset_val, batch_size=batch_size,shuffle=True,num_workers=4)
  data_loader_test  = DataLoader(dataset_test, batch_size=batch_size,shuffle=True,num_workers=4)
  return data_loader_train, data_loader_val, data_loader_test

def reshape_input(x, y):
    x = x.view(-1, 784)
    y = torch.FloatTensor(len(y), 10).zero_().scatter_(1,y.view(-1,1),1)
    return x, y


# call this once first to download the datasets
_ = get_fashion_mnist_dataloaders()
```

```
# simple logger to track progress during training
class Logger:
    def __init__(self):
        self.losses_train = []
        self.losses_valid = []
        self.accuracies_train = []
        self.accuracies_valid = []

    def log(self, accuracy_train=0, loss_train=0, accuracy_valid=0, loss_valid=0):
        self.losses_train.append(loss_train)
        self.accuracies_train.append(accuracy_train)
        self.losses_valid.append(loss_valid)
        self.accuracies_valid.append(accuracy_valid)

    def plot_loss_and_accuracy(self, train=True, valid=True):

        assert train and valid, "Cannot plot accuracy because neither train nor valid."

        figure, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,
                                          figsize=(12, 6))

        if train:
            ax1.plot(self.losses_train, label="Training")
            ax2.plot(self.accuracies_train, label="Training")
```

```
        if valid:
            ax1.plot(self.losses_valid, label="Validation")
            ax1.set_title("CrossEntropy Loss")
            ax2.plot(self.accuracies_valid, label="Validation")
            ax2.set_title("Accuracy")

        for ax in figure.axes:
            ax.set_xlabel("Epoch")
            ax.legend(loc='best')
            ax.set_axisbelow(True)
            ax.minorticks_on()
            ax.grid(True, which="major", linestyle='-')
            ax.grid(True, which="minor", linestyle='--', color='lightgrey', alpha=.4)

    def print_last(self):
        print(f"Epoch {len(self.losses_train):2d}, \
                Train:loss={self.losses_train[-1]:.3f}, accuracy={self.accuracies_train[-1]*100:.1f}%, \
                Valid: loss={self.losses_valid[-1]:.3f}, accuracy={self.losses_valid[-1]*100:.1f}%", flush=True)
```

Aperçu de l'ensemble de données FashionMnist

```
def plot_samples():
  a, _, _ = get_fashion_mnist_dataloaders()
  num_row = 2
  num_col = 5# plot images
  num_images = num_row * num_col
  fig, axes = plt.subplots(num_row, num_col, figsize=(1.5*num_col,2*num_row))
  for i, (x,y) in enumerate(a):
      if i >= num_images:
        break
      ax = axes[i//num_col, i%num_col]
      x = (x.numpy().squeeze() * 255).astype(int)
      y = y.numpy()[0]
      ax.imshow(x, cmap='gray')
      ax.set_title(f"Label: {y}")

  plt.tight_layout()
  plt.show()
plot_samples()
```



Fonctions à compléter

```
def accuracy(y, y_pred) :
  # nombre d'éléments à classifier.
  card_D = torch.tensor(y.shape[0])

  # calcul du nombre d'éléments bien classifiés.
  card_C = torch.sum(torch.argmax(y_pred, dim=1) == torch.argmax(y, dim=1))

  # calcul de la précision de classification.
  acc = torch.abs(card_C) / torch.abs(card_D)

  return acc, (card_C, card_D)

def accuracy_and_loss_whole_dataset(data_loader, model):
  cardinal = 0
  loss     = 0.
  n_accurate_preds  = 0.

  for x, y in data_loader:
    x, y = reshape_input(x, y)
    y_pred                = model.forward(x)
    xentrp                = cross_entropy(y, y_pred)
    _, (n_acc, n_samples) = accuracy(y, y_pred)

    cardinal = cardinal + n_samples
    loss     = loss + xentrp
    n_accurate_preds  = n_accurate_preds + n_acc

  loss = loss / float(cardinal)
  acc  = n_accurate_preds / float(cardinal)

  return acc, loss

def cross_entropy(y, y_pred):
  # calcul de la valeur d'entropie croisée.
  loss = -torch.sum(y * torch.log(torch.clamp(y_pred, 1e-12)))
  return loss

def softmax(x, axis=-1):
  # assurez vous que la fonction est numeriquement stable
  # e.g. softmax(torch.tensor([[1000, 10000, 100000]]))
  # calcul des valeurs de softmax(x)
  x = torch.exp(x - torch.max(x, dim=axis, keepdim=True).values)
  values = x / (torch.sum(x, dim=axis, keepdim=True) + 1e-12)
  return values

def inputs_tilde(x, axis=-1):
  # augments the inputs `x` with ones along `axis`
  x_tilde = torch.cat((x, torch.ones(x.shape[0], 1)), dim=axis)
  return x_tilde
```

```
class LinearModel:
    def __init__(self, num_features, num_classes):
      self.params = torch.normal(0, 0.01, (num_features + 1, num_classes))

```

```
      self.t = 0
      self.m_t = 0 # pour Adam: moyennes mobiles du gradient
      self.v_t = 0 # pour Adam: moyennes mobiles du carré du gradient

    def forward(self, x):
      # implémenter calcul des outputs en fonction des inputs `x`.
      inputs = inputs_tilde(x)
      outputs = softmax(torch.matmul(inputs, self.params), axis=-1)
      return outputs

    def get_grads(self, y, y_pred, X):
      # implémenter calcul des gradients.
      grads = torch.matmul(inputs_tilde(X).T, y_pred - y)
      return grads

    def sgd_update(self, lr, grads):
      # implémenter mise à jour des paramètres ici.
      self.params -= lr * grads

    def adam_update(self, lr, grads):
      # implémenter mise à jour des paramètres ici.
      beta_1 = 0.9
      beta_2 = 0.999
      epsilon = 1e-8
      self.t += 1

      self.m_t = beta_1 * self.m_t + (1 - beta_1) * grads
      self.v_t = beta_2 * self.v_t + (1 - beta_2) * grads ** 2

      m_t_hat = self.m_t / (1 - beta_1 ** self.t)
      v_t_hat = self.v_t / (1 - beta_2 ** self.t)

      self.params -= lr * m_t_hat / (torch.sqrt(v_t_hat) + epsilon)

def train(model, lr=0.1, nb_epochs=10, sgd=True, data_loader_train=None, data_loader_val=None):
    best_model = None
    best_val_accuracy = 0
    logger = Logger()

    for epoch in range(nb_epochs+1):
        # at epoch 0 evaluate random initial model
        #   then for subsequent epochs, do optimize before evaluation.
        if epoch > 0:
          for x, y in data_loader_train:
              x, y = reshape_input(x, y)
              y_pred = model.forward(x)
              loss = cross_entropy(y, y_pred)
              grads = model.get_grads(y, y_pred, x)
              if sgd:
                model.sgd_update(lr, grads)
              else:
                model.adam_update(lr, grads)

        accuracy_train, loss_train = accuracy_and_loss_whole_dataset(data_loader_train, model)
        accuracy_val, loss_val = accuracy_and_loss_whole_dataset(data_loader_val, model)

        if accuracy_val > best_val_accuracy:
          # record the best model parameters and best validation accuracy
          best_model = model
          best_val_accuracy = accuracy_val

        logger.log(accuracy_train, loss_train, accuracy_val, loss_val)
        print(f"Epoch {epoch:2d}, \
                Train: loss={loss_train.item():.3f}, accuracy={accuracy_train.item()*100:.1f}%, \
                Valid: loss={loss_val.item():.3f}, accuracy={accuracy_val.item()*100:.1f}%", flush=True)

    return best_model, best_val_accuracy, logger
```

Évaluation

SGD: Recherche d'hyperparamètres

```
# SGD
# Montrez les résultats pour différents taux d'apprentissage, e.g. 0.1, 0.01, 0.001, et différentes tailles de mini-batch, e.g. 1, 20, 200, 1000.
batch_size_list = [1, 20, 200, 1000]    # Define ranges in a list
lr_list = [0.1, 0.01, 0.001]            # Define ranges in a list

with torch.no_grad():
  for lr in lr_list:
    for batch_size in batch_size_list:
      print("----------------------------------------------------------------")
      print("Training model with a learning rate of {0} and a batch size of {1}".format(lr, batch_size))
      data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

      model = LinearModel(num_features=784, num_classes=10)
      _, val_accuracy, _ = train(model,lr=lr, nb_epochs=5, sgd=True, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
      print(f"validation accuracy = {val_accuracy*100:.3f}")
```

```
----------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 1
Epoch  0,                Train: loss=2.307, accuracy=11.0%,                Valid: loss=2.307, accuracy=10.7%
Epoch  1,                Train: loss=2.376, accuracy=78.8%,                Valid: loss=2.487, accuracy=78.4%
Epoch  2,                Train: loss=2.353, accuracy=82.7%,                Valid: loss=2.553, accuracy=81.4%
Epoch  3,                Train: loss=3.367, accuracy=77.8%,                Valid: loss=3.583, accuracy=77.0%
Epoch  4,                Train: loss=2.873, accuracy=78.9%,                Valid: loss=3.212, accuracy=77.2%
Epoch  5,                Train: loss=2.379, accuracy=80.2%,                Valid: loss=2.737, accuracy=78.2%
validation accuracy = 81.433
----------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 20
Epoch  0,                Train: loss=2.300, accuracy=20.1%,                Valid: loss=2.298, accuracy=20.3%
Epoch  1,                Train: loss=3.436, accuracy=78.3%,                Valid: loss=3.411, accuracy=78.5%
Epoch  2,                Train: loss=2.922, accuracy=82.0%,                Valid: loss=2.975, accuracy=81.8%
Epoch  3,                Train: loss=2.878, accuracy=83.1%,                Valid: loss=2.967, accuracy=82.7%
Epoch  4,                Train: loss=2.593, accuracy=83.6%,                Valid: loss=2.744, accuracy=83.1%
Epoch  5,                Train: loss=3.279, accuracy=81.4%,                Valid: loss=3.490, accuracy=80.5%
validation accuracy = 83.083
----------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 200
Epoch  0,                Train: loss=2.296, accuracy=8.8%,                Valid: loss=2.295, accuracy=8.7%
Epoch  1,                Train: loss=6.226, accuracy=75.1%,                Valid: loss=6.201, accuracy=75.1%
Epoch  2,                Train: loss=5.264, accuracy=78.7%,                Valid: loss=5.321, accuracy=78.6%
Epoch  3,                Train: loss=7.079, accuracy=71.9%,                Valid: loss=6.947, accuracy=72.2%
Epoch  4,                Train: loss=5.105, accuracy=79.0%,                Valid: loss=5.124, accuracy=78.6%
Epoch  5,                Train: loss=6.629, accuracy=72.6%,                Valid: loss=6.560, accuracy=72.6%
validation accuracy = 78.633
----------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 1000
Epoch  0,                Train: loss=2.292, accuracy=9.4%,                Valid: loss=2.289, accuracy=10.2%
Epoch  1,                Train: loss=6.921, accuracy=74.3%,                Valid: loss=7.066, accuracy=73.7%
```

```
Epoch  2,               Train: loss=7.858, accuracy=70.8%,               Valid: loss=7.720, accuracy=71.3%
Epoch  3,               Train: loss=5.097, accuracy=80.9%,               Valid: loss=5.178, accuracy=80.5%
Epoch  4,               Train: loss=6.278, accuracy=76.4%,               Valid: loss=6.186, accuracy=76.7%
Epoch  5,               Train: loss=6.073, accuracy=77.4%,               Valid: loss=6.096, accuracy=77.2%
validation accuracy = 80.517
-----------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 1
Epoch  0,               Train: loss=2.332, accuracy=12.3%,               Valid: loss=2.335, accuracy=11.9%
Epoch  1,               Train: loss=0.541, accuracy=81.9%,               Valid: loss=0.542, accuracy=81.6%
Epoch  2,               Train: loss=0.461, accuracy=84.6%,               Valid: loss=0.470, accuracy=83.7%
Epoch  3,               Train: loss=0.559, accuracy=83.0%,               Valid: loss=0.550, accuracy=83.3%
Epoch  4,               Train: loss=0.436, accuracy=85.7%,               Valid: loss=0.447, accuracy=85.7%
Epoch  5,               Train: loss=0.425, accuracy=86.0%,               Valid: loss=0.445, accuracy=85.1%
validation accuracy = 85.733
-----------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 20
Epoch  0,               Train: loss=2.307, accuracy=11.5%,               Valid: loss=2.308, accuracy=12.2%
Epoch  1,               Train: loss=0.719, accuracy=79.0%,               Valid: loss=0.752, accuracy=79.0%
Epoch  2,               Train: loss=0.628, accuracy=83.2%,               Valid: loss=0.661, accuracy=82.0%
Epoch  3,               Train: loss=0.569, accuracy=84.1%,               Valid: loss=0.630, accuracy=82.7%
Epoch  4,               Train: loss=0.556, accuracy=84.4%,               Valid: loss=0.613, accuracy=83.5%
Epoch  5,               Train: loss=0.519, accuracy=84.2%,               Valid: loss=0.578, accuracy=83.1%
validation accuracy = 83.533
-----------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 200
Epoch  0,               Train: loss=2.291, accuracy=4.5%,               Valid: loss=2.293, accuracy=4.3%
```

Tableau pour la précision sur l'ensemble de validation

N.B. que les lignes correspondent aux valeurs du taux d'apprentisage et les colonnes correspondent au valeur du batch size. Les valeurs ci-dessous sont donné comme exemples; remplacez-les par les valeurs que vous avez utilisées pour votre recherche d'hyperparamètres.

learning rate\batch_size	1	20	200	1000
0.1	81.433	83.433	82.150	80.517
0.01	85.733	83.533	80.900	77.433
0.001	84.150	84.667	84.633	76.683

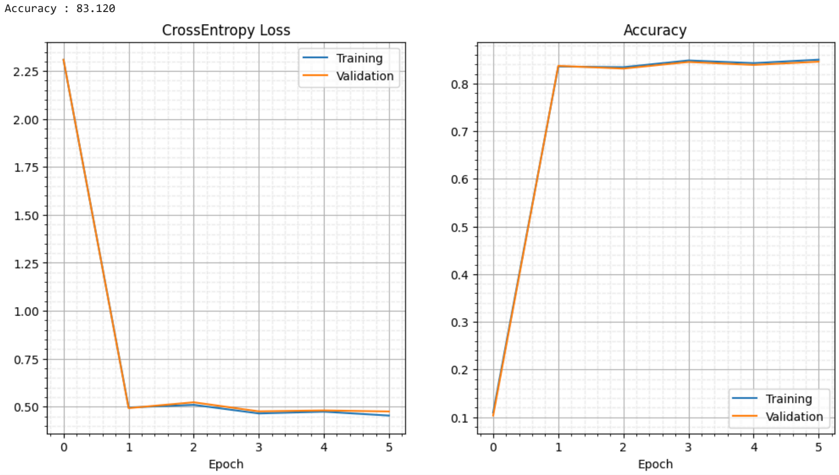
En ce qui concerne nos hyperparamètres pour SGD, il semble y avoir une augmentation de la qualité de notre modèle en ayant un nombre petit comme **batch_size** et un **learning_rate** plus petit. Cependant, l'apprentissage ce modèle ayant un **batch_size** petit prend beaucoup plus de temps que ceux ayant une grande valeur.

La meilleure valeur de précision que j'ai obtenu se retrouve pour un **batch_size** de **1** et un **learning_rate** de **0.01** où l'on obtient une précision d'environ **85.733**. Ce sont ces valeurs que nous prendrons pour l'analyse du modèle.

SGD: Analyse du meilleur modèle

```
# SGD
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
batch_size = 1    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.01         # Vous devez modifier cette valeur avec la meilleur que vous avez eu.

with torch.no_grad():
  data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

  model = LinearModel(num_features=784, num_classes=10)
  best_model, best_val_accuracy, logger = train(model,lr=lr, nb_epochs=5, sgd=True,
                                               data_loader_train=data_loader_train, data_loader_val=data_loader_val)
  logger.plot_loss_and_accuracy()
  print(f"Best validation accuracy = {best_val_accuracy*100:.3f}")

  accuracy_test, loss_test = accuracy_and_loss_whole_dataset(data_loader_test, best_model)
print("Evaluation of the best training model over test set")
print("------")
print(f"Loss : {loss_test:.3f}")
print(f"Accuracy : {accuracy_test*100.:.3f}")
```

```
Epoch  0,               Train: loss=2.308, accuracy=10.9%,               Valid: loss=2.309, accuracy=10.3%
Epoch  1,               Train: loss=0.496, accuracy=83.5%,               Valid: loss=0.492, accuracy=83.7%
Epoch  2,               Train: loss=0.509, accuracy=83.4%,               Valid: loss=0.523, accuracy=83.1%
Epoch  3,               Train: loss=0.465, accuracy=84.8%,               Valid: loss=0.475, accuracy=84.5%
Epoch  4,               Train: loss=0.474, accuracy=84.3%,               Valid: loss=0.480, accuracy=83.9%
Epoch  5,               Train: loss=0.453, accuracy=85.0%,               Valid: loss=0.475, accuracy=84.6%
Best validation accuracy = 84.567
Evaluation of the best training model over test set
------
Loss : 0.531
Accuracy : 83.120
```



On peut voir dans le graphique que notre entrainement est extrêment efficace dans la première époque, mais se stabilise à partir de cette première époque jusqu'à la fin. Il y a donc un plateau à partir de l'époque 1 et il arrête d'optimiser, soit car il arrête d'apprendre ou qu'il a trouver un minimum valide.

De plus, comme la validation et l'entraînement sont très similaire, il n'y a pas de sur-apprentissage (overfitting) ni de sous-apprentissage (underfitting) pour le moment. Cependant, on peut voir qu'à l'époque 5, il commence à avoir une légère diférence entre l'entraînement et la validation, et pourrait ainsi nous indiquer que si nous augmentons le nombre d'époque, que l'algorithme va faire une surapprentissage (overfitting) dans ces prochaines époques.

Pour ce qui est des résultats, on obtient une **valeur de perte finale** de **0.531** et une **valeur de précision finale** de **83.120**

Adam: Recherche d'hyperparamètres

Implémentez Adam, répétez les deux étapes précédentes (recherche d'hyperparamètres et analyse du meilleur modèle) cette fois en utilisat Adam, et comparez les performances finales avec votre meilleur modèle SGD.

```
# ADAM
# Montrez les résultats pour différents taux d'apprentissage, e.g. 0.1, 0.01, 0.001, et différentes tailles de mini-batch, e.g. 1, 20, 200, 1000.
batch_size_list = [1, 20, 200, 1000]      # Define ranges in a list
lr_list = [0.1, 0.01, 0.001]              # Define ranges in a list

with torch.no_grad():
  for lr in lr_list:
    for batch_size in batch_size_list:
      print("------------------------------------------------------------------")
      print("Training model with a learning rate of {0} and a batch size of {1}".format(lr, batch_size))
      data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

      model = LinearModel(num_features=784, num_classes=10)
      _, val_accuracy, _ = train(model,lr=lr, nb_epochs=5, sgd=False, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
      print(f"validation accuracy = {val_accuracy*100:.3f}")
```

```
------------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 1
Epoch  0,               Train: loss=2.300, accuracy=10.7%,               Valid: loss=2.300, accuracy=10.9%
Epoch  1,               Train: loss=5.821, accuracy=77.0%,               Valid: loss=5.820, accuracy=77.1%
Epoch  2,               Train: loss=4.387, accuracy=82.0%,               Valid: loss=4.834, accuracy=80.6%
Epoch  3,               Train: loss=4.694, accuracy=81.1%,               Valid: loss=4.901, accuracy=80.4%
Epoch  4,               Train: loss=4.329, accuracy=82.6%,               Valid: loss=4.835, accuracy=80.6%
Epoch  5,               Train: loss=7.544, accuracy=70.6%,               Valid: loss=7.638, accuracy=70.1%
validation accuracy = 80.617
------------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 20
Epoch  0,               Train: loss=2.304, accuracy=6.4%,               Valid: loss=2.306, accuracy=6.6%
Epoch  1,               Train: loss=3.983, accuracy=75.8%,               Valid: loss=4.133, accuracy=74.8%
Epoch  2,               Train: loss=2.518, accuracy=83.4%,               Valid: loss=2.815, accuracy=82.0%
Epoch  3,               Train: loss=2.901, accuracy=81.4%,               Valid: loss=3.161, accuracy=80.0%
Epoch  4,               Train: loss=2.384, accuracy=84.9%,               Valid: loss=2.802, accuracy=83.1%
Epoch  5,               Train: loss=2.885, accuracy=82.1%,               Valid: loss=3.280, accuracy=80.3%
validation accuracy = 83.133
------------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 200
Epoch  0,               Train: loss=2.303, accuracy=13.3%,               Valid: loss=2.302, accuracy=13.8%
Epoch  1,               Train: loss=0.734, accuracy=82.5%,               Valid: loss=0.830, accuracy=81.5%
Epoch  2,               Train: loss=1.342, accuracy=77.2%,               Valid: loss=1.484, accuracy=75.9%
Epoch  3,               Train: loss=0.962, accuracy=83.3%,               Valid: loss=1.123, accuracy=81.7%
Epoch  4,               Train: loss=1.341, accuracy=80.6%,               Valid: loss=1.491, accuracy=79.7%
Epoch  5,               Train: loss=0.603, accuracy=85.7%,               Valid: loss=0.812, accuracy=83.6%
validation accuracy = 83.617
------------------------------------------------------------------
Training model with a learning rate of 0.1 and a batch size of 1000
Epoch  0,               Train: loss=2.297, accuracy=12.3%,               Valid: loss=2.299, accuracy=11.4%
Epoch  1,               Train: loss=0.859, accuracy=81.6%,               Valid: loss=0.855, accuracy=82.2%
Epoch  2,               Train: loss=0.511, accuracy=83.8%,               Valid: loss=0.536, accuracy=83.8%
Epoch  3,               Train: loss=0.483, accuracy=84.2%,               Valid: loss=0.516, accuracy=84.0%
Epoch  4,               Train: loss=0.846, accuracy=77.4%,               Valid: loss=0.910, accuracy=76.7%
Epoch  5,               Train: loss=0.498, accuracy=84.1%,               Valid: loss=0.541, accuracy=83.9%
validation accuracy = 84.050
------------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 1
Epoch  0,               Train: loss=2.327, accuracy=4.5%,               Valid: loss=2.327, accuracy=4.1%
Epoch  1,               Train: loss=2.188, accuracy=78.6%,               Valid: loss=2.203, accuracy=78.8%
Epoch  2,               Train: loss=2.907, accuracy=78.1%,               Valid: loss=3.026, accuracy=77.9%
Epoch  3,               Train: loss=2.540, accuracy=79.8%,               Valid: loss=2.693, accuracy=79.4%
Epoch  4,               Train: loss=1.963, accuracy=81.0%,               Valid: loss=2.214, accuracy=80.0%
Epoch  5,               Train: loss=2.219, accuracy=79.3%,               Valid: loss=2.401, accuracy=78.8%
validation accuracy = 80.017
------------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 20
Epoch  0,               Train: loss=2.343, accuracy=5.8%,               Valid: loss=2.344, accuracy=6.0%
Epoch  1,               Train: loss=0.845, accuracy=80.9%,               Valid: loss=0.911, accuracy=79.8%
Epoch  2,               Train: loss=0.592, accuracy=82.4%,               Valid: loss=0.633, accuracy=82.3%
Epoch  3,               Train: loss=0.602, accuracy=83.7%,               Valid: loss=0.686, accuracy=82.5%
Epoch  4,               Train: loss=0.744, accuracy=80.3%,               Valid: loss=0.851, accuracy=78.8%
Epoch  5,               Train: loss=0.561, accuracy=84.8%,               Valid: loss=0.672, accuracy=83.3%
validation accuracy = 83.317
------------------------------------------------------------------
Training model with a learning rate of 0.01 and a batch size of 200
Epoch  0,               Train: loss=2.348, accuracy=1.9%,               Valid: loss=2.346, accuracy=1.7%
Epoch  1,               Train: loss=0.445, accuracy=84.8%,               Valid: loss=0.468, accuracy=83.6%
```

Tableau pour la précision sur l'ensemble de validation

N.B. que les lignes correspondent aux valeurs du taux d'apprentissage et les colonnes correspondent au valeur du batch size. Les valeurs ci-dessous sont donné comme exemples; remplacez-les par les valeurs que vous avez utilisées pour votre recherche d'hyperparamètres.

learning rate\batch_size	1	20	200	1000
0.1	80.617	83.133	83.617	84.050
0.01	80.017	83.317	83.917	84.800
0.001	83.817	85.400	85.133	81.117

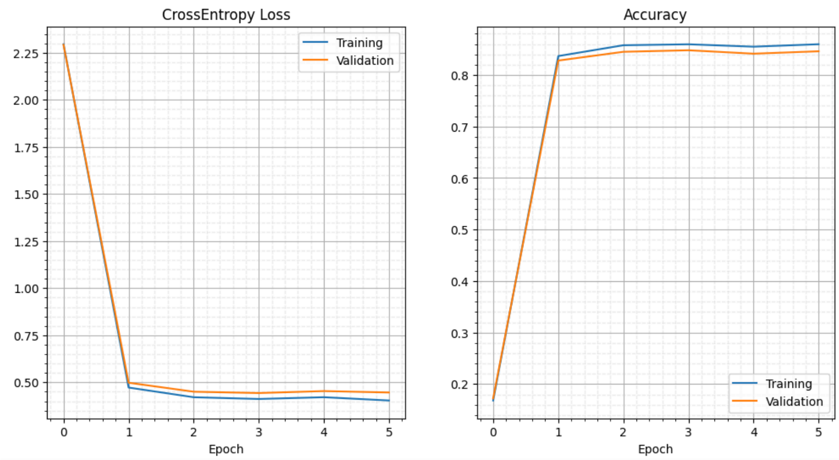
En ce qui concerne nos hyperparamètres pour Adam, il semble y avoir une augmentation de la qualité de notre modèle en ayant un nombre petit comme **learning_rate**. Pour le **bach_size**, ils semble y avoir une amélioration de la précision lorsque la valeur est plus grande, mais les résultats ne semble pas nécessairement assez concluant pour vraiment assumer que cela se produit. Il faudrait faire d'autres expériences sur le sujet, mais les expériences m'ont pris un temps assez énorme que j'ai pris la décision de ne pas en ajouté (environ 3 heures pour l'ensemble de la partie 2). Cependant, il est vrai que cela pourrait se faire un peu plus rapidement sachant qu'un **batch_size** moins élevé prendrais moins de temps pour l'algorithme.

La meilleure valeur de précision que j'ai obtenu se retrouve pour un **batch_size** de **20** et un **learning_rate** de **0.001** où l'on obtient une précision d'environ **85.400**. Ce sont ces valeurs que nous prendrons pour l'analyse du modèle Adam.

Adam: Analyse du meilleur modèle

```
# ADAM
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
batch_size = 20     # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001          # Vous devez modifier cette valeur avec la meilleur que vous avez eu.

with torch.no_grad():
  data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

  model = LinearModel(num_features=784, num_classes=10)
  best_model, best_val_accuracy, logger = train(model,lr=lr, nb_epochs=5, sgd=False,
                                                data_loader_train=data_loader_train, data_loader_val=data_loader_val)
  logger.plot_loss_and_accuracy()
  print(f"Best validation accuracy = {best_val_accuracy*100:.3f}")

  accuracy_test, loss_test = accuracy_and_loss_whole_dataset(data_loader_test, best_model)
print("Evaluation of the best training model over test set")
print("------")
print(f"Loss : {loss_test:.3f}")
print(f"Accuracy : {accuracy_test*100.:.3f}")
```

```
Epoch  0,              Train: loss=2.295, accuracy=16.8%,              Valid: loss=2.292, accuracy=17.2%
Epoch  1,              Train: loss=0.472, accuracy=83.7%,              Valid: loss=0.498, accuracy=82.8%
Epoch  2,              Train: loss=0.421, accuracy=85.8%,              Valid: loss=0.450, accuracy=84.5%
Epoch  3,              Train: loss=0.412, accuracy=86.0%,              Valid: loss=0.443, accuracy=84.8%
Epoch  4,              Train: loss=0.421, accuracy=85.5%,              Valid: loss=0.454, accuracy=84.1%
Epoch  5,              Train: loss=0.404, accuracy=86.0%,              Valid: loss=0.446, accuracy=84.6%
Best validation accuracy = 84.800
Evaluation of the best training model over test set
------
Loss : 0.464
Accuracy : 83.870
```



On peut voir dans le graphique que notre entrainement est extrêment efficace dans la première époque, mais se stabilise à partir de cette première époque jusqu'à la fin. Il y a donc un plateau à partir de l'époque 1 et il arrête d'optimiser, soit car il arrête d'apprendre ou qu'il a trouver un minimum valide.

Cependant, notre résultat d'entrainement est plus précis que celle de notre validation. On peut ainsi dire qu'il y a un sur-apprentissage (overfitting) du modèle car celui-ci se comporte mieux sur des données d'entraînement que sur les données de validations. Il aura donc une plus grande difficulté par la suite à apprendre de nouvelle valeurs car il y aura un fort poids déjà présent par l'entrainement qu'il a eu. Dans le même ordre d'idée, l'algorithme n'est assurément pas en train de faire de sous-apprentissage (underfitting) étant dans un contexte inverse actuellement.

Pour ce qui est des résultats, on obtient une **valeur de perte finale** de **0.464** et une **valeur de précision finale** de **83.870**

Analyse des Résultats

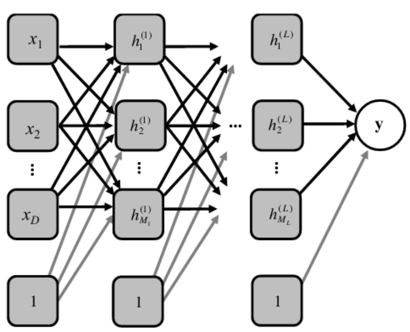
(Voir l'ensemble des textes en bleu plus haut. Ceux-ci font parti de mon analyse, mais je trouvais plus simple de les mettre par section et de résumé par la suite pour les deux algorithmes)

En somme, nos deux algorithmes d'apprentissages sont très similaire bien que celui de Adam nous donne un meilleur résultat lorsque nous prenons les meilleures valeurs de **batch_size** ains que de **learning_rate**. Comme Adam semble faire plus rapidement du sur-apprentissage (overfitting), on peut ainsi déduire qu'il apprend beaucoup plus rapidement que celui de SGD. Cependant, comme leurs valeurs sont sensiblement très similaire, on ne peut pas affirmer que l'un des algorithme est meilleur que l'autre avec uniquement les expériences que j'ai fait.

Partie 3 (20 points)

Pour cette partie, vous pouvez travailler en groupes de 2, mais il faut écrire sa propre dérivation et soumettre son propre rapport. Si vous travaillez avec un partenaire, il faut indiquer leur nom dans votre rapport.

Problème



Considérons maintenant un réseau de neurones avec une couche d'entrée avec $D=784$$D=784$ unités, $L$$L$ couches cachées, chacune avec 300 unités et un vecteur de sortie $\mathbf{y}$$\mathbf{y}$ de dimension $K$$K$. Vous avez $i = 1, .., N$$i = 1, .., N$ exemples dans un ensemble d'apprentissage, où chaque ${\bf x}_i \in \mathbb{R}^{784}$${\bf x}_i \in \mathbb{R}^{784}$ est un vecteur de caractéristiques (features). $\mathbf{y}$$\mathbf{y}$ est un vecteur du type *one-hot* -- un vecteur de zéros avec un seul 1 pour indiquer que la classe $C=k$$C=k$ dans la dimension $k$$k$. Par exemple, le vecteur $\mathbf{y}=[0, 1, 0, \cdots, 0]^T$$\mathbf{y}=[0, 1, 0, \cdots, 0]^T$ représente la deuxième classe. La fonction de perte est donnée par \begin{equation} \mathscr{L} = -\sum*{i=1}^{N}\sum*{k=1}^{K}y_{k,i}\log (f_k({\bf x}_i)) \end{equation}\begin{equation} \mathscr{L} = -\sum*{i=1}^{N}\sum*{k=1}^{K}y_{k,i}\log (f_k({\bf x}_i)) \end{equation}

La fonction d'activation de la couche finale a la forme ${\bf f} = [f_1, ..., f_K]$${\bf f} = [f_1, ..., f_K]$ donné par la fonction d'activation softmax: \begin{equation} f*k({\bf a}^{(L+1)}({\bf x}_i)) = \frac{\exp(a_k^{(L+1)})}{\sum*{c=1}^{K}\exp(a_c^{(L+1)})}, \;\;\;\; \nonumber \end{equation}\begin{equation} f*k({\bf a}^{(L+1)}({\bf x}_i)) = \frac{\exp(a_k^{(L+1)})}{\sum*{c=1}^{K}\exp(a_c^{(L+1)})}, \;\;\;\; \nonumber \end{equation}

et les couches cachées utilisent une fonction d'activation de type ReLU: \begin{equation} {\bf h}^{(l)}({\bf a}^{(l)}({\bf x}_i)) = \text{ReLU}({\bf a}^{(l)}({\bf x}_i) = \max\Big(0, \, \, {\bf a}^{(l)}({\bf x}_i)\Big) \end{equation}\begin{equation} {\bf h}^{(l)}({\bf a}^{(l)}({\bf x}_i)) = \text{ReLU}({\bf a}^{(l)}({\bf x}_i) = \max\Big(0, \, \, {\bf a}^{(l)}({\bf x}_i)\Big) \end{equation} où ${\bf a}^{(l)}$${\bf a}^{(l)}$ est le vecteur résultant du calcul de la préactivation habituelle ${\bf a}^{(l)}={\bf W}^{(l)}{\bf h}^{(l-1)} + {\bf b}^{(l)}$${\bf a}^{(l)}={\bf W}^{(l)}{\bf h}^{(l-1)} + {\bf b}^{(l)}$, qui pourrait être simplifiée à ${\boldsymbol \theta}^{(l)}\tilde{\bf h}^{(l-1)}$${\boldsymbol \theta}^{(l)}\tilde{\bf h}^{(l-1)}$ en utilisant l'astuce de définir $\tilde{\bf h}$$\tilde{\bf h}$ comme ${\bf h}$${\bf h}$ avec un 1 concaténé à la fin du vecteur.

Questions

- a) (10 points) Donnez le 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**.

- b) (15 points) Implémentez l'optimisation basée sur le gradient de ce réseau en Pytorch. Utilisez le code squelette ci-dessous comme point de départ et implémentez les mathématiques de l'algorithme de rétropropagation que vous avez décrit à la question précédente. Comparez vos gradients et votre optimisation avec le même modèle optimisé avec Autograd. Lequel est le plus rapide ? Proposez quelques expériences. Utilisez encore l'ensemble de données de Fashion MNIST (voir Partie 2). **Comparez différents modèles ayant différentes largeurs (nombre d'unités) et profondeurs (nombre de couches)**. Ici encore, n'utilisez l'ensemble de test que pour votre expérience finale lorsque vous pensez avoir obtenu votre meilleur modèle.

IMPORTANT

L'objectif du TP est de vous faire implémenter la rétropropagation à la main. L'objectif est d'implémenter un modèle de classification logistique ainsi que son entainement en utilisant uniquement des opérations matricielles de base fournies par PyTorch e.g. torch.sum(), torch.matmul(), etc. **Une fois que vous avez implémenté votre modèle, vous devez le comparer avec un modèle construit en utilisant les capacités de pytorch qui permettent une différenciation automatique. Autrement dit, pour la deuxième implémentation, vous pouvez utilisertorch.nn, torch.autograd ou à la méthode .backward().** Vous pouvez utiliser l'implémentation de votre choix pour explorer différentes architectures de modèles.

Votre pseudocode:

Algorithme de rétropopagation dans un réseau de neurones pour un exemple $\tilde{x}_i$$\tilde{x}_i$:

1. function backPropagation()
2. $\Delta$$\Delta$ = {}
3. $\frac{\partial{L}}{\partial{\theta}}$$\frac{\partial{L}}{\partial{\theta}}$ = {}
4. $\Delta^{(L+1)} = \tilde{y}_i - forward(\tilde{x}_i) $$\Delta^{(L+1)} = \tilde{y}_i - forward(\tilde{x}_i) $
5. $\frac{\partial{L}}{\partial{\theta^{(L+1)}}} = -\Delta^{(L+1)} {\tilde{h}^{T}}_{(L)}$$\frac{\partial{L}}{\partial{\theta^{(L+1)}}} = -\Delta^{(L+1)} {\tilde{h}^{T}}_{(L)}$
6. for l = L-1 down to 1
7. $\Delta^{(l)} = \frac{\partial{h^{(l)}}}{\partial{a^{(l)}}} \theta^{T(l+1)} \Delta^{(l+1)}$$\Delta^{(l)} = \frac{\partial{h^{(l)}}}{\partial{a^{(l)}}} \theta^{T(l+1)} \Delta^{(l+1)}$
8. $\frac{\partial{L}}{\partial{\theta^{(l)}}} = -\Delta^{(l)} {\hat{h}^{T}}_{(l-1)}$$\frac{\partial{L}}{\partial{\theta^{(l)}}} = -\Delta^{(l)} {\hat{h}^{T}}_{(l-1)}$
9. endfor
10. return $\frac{\partial{L}}{\partial{\theta}}$$\frac{\partial{L}}{\partial{\theta}}$

Fonctions à compléter

```
''' Les fonctions dans cette cellule peuvent avoir les mêmes déclarations que celles de la partie 2'''
def accuracy(y, y_pred) :
    # nombre d'éléments à classifier.
    card_D = torch.tensor(y.shape[0])

    # calcul du nombre d'éléments bien classifiés.
    card_C = torch.sum(torch.argmax(y_pred, dim=1) == torch.argmax(y, dim=1))

    # calcul de la précision de classification.
    acc = torch.abs(card_C) / torch.abs(card_D)

    return acc, (card_C, card_D)

def accuracy_and_loss_whole_dataset(data_loader, model):
    cardinal = 0
    loss     = 0.
    n_accurate_preds  = 0.

    for x, y in data_loader:
        x, y = reshape_input(x, y)
        y_pred                = model.forward(x)
        xentrp                = cross_entropy(y, y_pred)
        _, (n_acc, n_samples) = accuracy(y, y_pred)

        cardinal = cardinal + n_samples
        loss     = loss + xentrp
        n_accurate_preds  = n_accurate_preds + n_acc

    loss = loss / float(cardinal)
    acc  = n_accurate_preds / float(cardinal)

    return acc, loss

def inputs_tilde(x, axis=-1):
    # augments the inputs `x` with ones along `axis`
    x_tilde = torch.cat((x, torch.ones(x.shape[0], 1)), dim=axis)
    return x_tilde

def softmax(x, axis=-1):
    # assurez vous que la fonction est numeriquement stable
    # e.g. softmax(np.array([1000, 10000, 100000], ndim=2))
    # calcul des valeurs de softmax(x)
    x = torch.exp(x - torch.max(x, dim=axis, keepdim=True).values)
    values = x / (torch.sum(x, dim=axis, keepdim=True) + 1e-10)
    return values

def cross_entropy(y, y_pred):
    # calcul de la valeur d'entropie croisée.
    loss = -torch.sum(y * torch.log(torch.clamp(y_pred, 1e-10)))
    return loss

def softmax_cross_entropy_backward(y, y_pred):
    # calcul de la valeur du gradient de l'entropie croisée composée avec `softmax`
    values = y - y_pred
    return values

def relu_forward(x):
    # calcul des valeurs de relu(x)
    values = torch.max(x, torch.zeros_like(x))
```

```
    return values

def relu_backward(x):
    # calcul des valeurs du gradient de la fonction `relu`
    values = torch.where(x > 0, 1, 0)
    return values


# Model est une classe representant votre reseaux de neuronnes
class MLPModel:
    def __init__(self, n_features, n_hidden_features, n_hidden_layers, n_classes):
        self.n_features        = n_features
        self.n_hidden_features = n_hidden_features
        self.n_hidden_layers   = n_hidden_layers
        self.n_classes         = n_classes

        # initialiser la liste des paramètres Teta de l'estimateur.
        self.params = [
            torch.normal(0, 0.1, (n_features + 1, n_hidden_features)),
            *[torch.normal(0, 0.1, (n_hidden_features + 1, n_hidden_features)) for _ in range(n_hidden_layers - 1)],
            torch.normal(0, 0.1, (n_hidden_features + 1, n_classes))
        ]
        print(f"Teta params={[p.shape for p in self.params]}")

        self.a = [torch.empty(1) for _ in range(self.n_hidden_layers + 2)] # liste contenant le resultat des multiplications matricielles
        self.h = [torch.empty(1) for _ in range(self.n_hidden_layers + 2)] # liste contenant le resultat des fonctions d'activations

        self.t = 0
        self.m_t = [0 for _ in range(n_hidden_layers + 2)] # pour Adam: moyennes mobiles du gradient
        self.v_t = [0 for _ in range(n_hidden_layers + 2)] # pour Adam: moyennes mobiles du carré du gradient

    def forward(self, x):
        # implémenter calcul des outputs en fonction des inputs `x`.
        self.h[0] = inputs_tilde(x)
        for l in range(self.n_hidden_layers):
            self.a[l + 1] = self.h[l] @ self.params[l]
            self.h[l + 1] = inputs_tilde(relu_forward(self.a[l + 1]))

        self.a[self.n_hidden_layers + 1] = self.h[self.n_hidden_layers] @ self.params[self.n_hidden_layers]
        self.h[self.n_hidden_layers + 1] = softmax(self.a[self.n_hidden_layers + 1])

        return self.h[self.n_hidden_layers + 1]

    def backward(self, y, y_pred):
        # implémenter calcul des gradients.
        grads = [torch.empty(1) for _ in range(self.n_hidden_layers + 1)]

        delta = softmax_cross_entropy_backward(y, y_pred)
        grads[self.n_hidden_layers] = self.h[self.n_hidden_layers].T @ -delta

        for l in range(self.n_hidden_layers, 0, -1):
            delta =  ((delta @ self.params[l].T) * relu_backward(self.h[l]))[:, :-1]
            grads[l - 1] = self.h[l - 1].T @ -delta

        return grads

    def sgd_update(self, lr, grads):
        # implémenter mise à jour des paramètres ici.
        for l in range(self.n_hidden_layers + 1):
            self.params[l] -= lr * grads[l]

    def adam_update(self, lr, grads):
        # implémenter mise à jour des paramètres ici.
        beta_1 = 0.9
        beta_2 = 0.999
        epsilon = 1e-8
        self.t += 1

        for l in range(self.n_hidden_layers + 1):
            self.m_t[l] = beta_1 * self.m_t[l] + (1 - beta_1) * grads[l]
            self.v_t[l] = beta_2 * self.v_t[l] + (1 - beta_2) * grads[l] ** 2

            m_t_hat = self.m_t[l] / (1 - beta_1 ** self.t)
            v_t_hat = self.v_t[l] / (1 - beta_2 ** self.t)

            self.params[l] -= lr * m_t_hat / (torch.sqrt(v_t_hat) + epsilon)

def train(model, lr=0.1, nb_epochs=10, sgd=True, data_loader_train=None, data_loader_val=None):
    best_model = None
    best_val_accuracy = 0
    logger = Logger()

    for epoch in range(nb_epochs+1):
        # at epoch 0 evaluate random initial model
        #   then for subsequent epochs, do optimize before evaluation.
        if epoch > 0:
            for x, y in data_loader_train:
                x, y = reshape_input(x, y)

                y_pred = model.forward(x)
                grads  = model.backward(y, y_pred)
                if sgd:
                    model.sgd_update(lr, grads)
                else:
                    model.adam_update(lr, grads)

        accuracy_train, loss_train = accuracy_and_loss_whole_dataset(data_loader_train, model)
        accuracy_val, loss_val = accuracy_and_loss_whole_dataset(data_loader_val, model)

        if accuracy_val > best_val_accuracy:
            # record the best model parameters and best validation accuracy
            best_model = model
            best_val_accuracy = accuracy_val

        logger.log(accuracy_train, loss_train, accuracy_val, loss_val)
        print(f"Epoch {epoch:2d}, \
                Train:loss={loss_train.item():.3f}, accuracy={accuracy_train.item()*100:.1f}%, \
                Valid: loss={loss_val.item():.3f}, accuracy={accuracy_val.item()*100:.1f}%", flush=True)

    return best_model, best_val_accuracy, logger
```

Évaluation

SGD: Recherche d'hyperparamètres

```
# SGD
# Montrez les résultats pour différents nombre de couche, e.g. 1, 3, 5, et différent nombres de neurone, e.g. 25, 100, 300, 500, 1000.
depth_list = [1, 3, 5]   # Define ranges in a list
```

```
width_list = [25, 100, 300, 500, 1000]   # Define ranges in a list
lr = 0.001          # Some value
batch_size = 16     # Some value

with torch.no_grad():
  for depth in depth_list:
    for width in width_list:
      print("----------------------------------------------------------------")
      print("Training model with a depth of {0} layers and a width of {1} units".format(depth, width))
      data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

      MLP_model = MLPModel(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
      _, val_accuracy, _ = train(MLP_model,lr=lr, nb_epochs=5, sgd=True, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
      print(f"validation accuracy = {val_accuracy*100:.3f}")
```

```
----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 25 units
Teta params=[torch.Size([785, 25]), torch.Size([26, 10])]
Epoch  0,                        Train:loss=2.410, accuracy=11.4%,                        Valid: loss=2.398, accuracy=12.1%
Epoch  1,                        Train:loss=0.520, accuracy=81.6%,                        Valid: loss=0.524, accuracy=81.4%
Epoch  2,                        Train:loss=0.453, accuracy=84.3%,                        Valid: loss=0.458, accuracy=84.1%
Epoch  3,                        Train:loss=0.439, accuracy=84.2%,                        Valid: loss=0.449, accuracy=84.1%
Epoch  4,                        Train:loss=0.423, accuracy=84.8%,                        Valid: loss=0.435, accuracy=84.8%
Epoch  5,                        Train:loss=0.404, accuracy=85.7%,                        Valid: loss=0.417, accuracy=85.4%
validation accuracy = 85.433
----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 100 units
Teta params=[torch.Size([785, 100]), torch.Size([101, 10])]
Epoch  0,                        Train:loss=2.604, accuracy=4.0%,                         Valid: loss=2.597, accuracy=4.5%
Epoch  1,                        Train:loss=0.491, accuracy=82.8%,                        Valid: loss=0.508, accuracy=82.6%
Epoch  2,                        Train:loss=0.429, accuracy=85.1%,                        Valid: loss=0.444, accuracy=84.4%
Epoch  3,                        Train:loss=0.399, accuracy=86.0%,                        Valid: loss=0.422, accuracy=85.4%
Epoch  4,                        Train:loss=0.395, accuracy=85.6%,                        Valid: loss=0.419, accuracy=84.5%
Epoch  5,                        Train:loss=0.360, accuracy=87.1%,                        Valid: loss=0.392, accuracy=85.5%
validation accuracy = 85.500
----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 300 units
Teta params=[torch.Size([785, 300]), torch.Size([301, 10])]
Epoch  0,                        Train:loss=3.093, accuracy=11.4%,                        Valid: loss=3.095, accuracy=11.4%
Epoch  1,                        Train:loss=0.443, accuracy=84.9%,                        Valid: loss=0.439, accuracy=84.9%
Epoch  2,                        Train:loss=0.398, accuracy=86.2%,                        Valid: loss=0.402, accuracy=86.3%
Epoch  3,                        Train:loss=0.392, accuracy=86.3%,                        Valid: loss=0.402, accuracy=85.4%
Epoch  4,                        Train:loss=0.358, accuracy=87.4%,                        Valid: loss=0.369, accuracy=87.1%
Epoch  5,                        Train:loss=0.333, accuracy=88.2%,                        Valid: loss=0.352, accuracy=87.3%
validation accuracy = 87.283
----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 500 units
Teta params=[torch.Size([785, 500]), torch.Size([501, 10])]
Epoch  0,                        Train:loss=3.126, accuracy=10.6%,                        Valid: loss=3.137, accuracy=10.6%
Epoch  1,                        Train:loss=0.457, accuracy=84.0%,                        Valid: loss=0.461, accuracy=83.7%
Epoch  2,                        Train:loss=0.390, accuracy=86.2%,                        Valid: loss=0.404, accuracy=85.7%
Epoch  3,                        Train:loss=0.351, accuracy=87.5%,                        Valid: loss=0.371, accuracy=86.8%
Epoch  4,                        Train:loss=0.357, accuracy=87.4%,                        Valid: loss=0.382, accuracy=86.8%
Epoch  5,                        Train:loss=0.321, accuracy=88.5%,                        Valid: loss=0.349, accuracy=87.6%
validation accuracy = 87.617
----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 1000 units
Teta params=[torch.Size([785, 1000]), torch.Size([1001, 10])]
Epoch  0,                        Train:loss=3.508, accuracy=16.6%,                        Valid: loss=3.520, accuracy=16.3%
Epoch  1,                        Train:loss=0.414, accuracy=85.8%,                        Valid: loss=0.439, accuracy=84.8%
Epoch  2,                        Train:loss=0.353, accuracy=87.6%,                        Valid: loss=0.387, accuracy=86.7%
Epoch  3,                        Train:loss=0.327, accuracy=88.5%,                        Valid: loss=0.374, accuracy=87.0%
Epoch  4,                        Train:loss=0.309, accuracy=89.1%,                        Valid: loss=0.356, accuracy=87.7%
Epoch  5,                        Train:loss=0.290, accuracy=89.9%,                        Valid: loss=0.342, accuracy=88.4%
validation accuracy = 88.433
----------------------------------------------------------------
Training model with a depth of 3 layers and a width of 25 units
Teta params=[torch.Size([785, 25]), torch.Size([26, 25]), torch.Size([26, 25]), torch.Size([26, 10])]
Epoch  0,                        Train:loss=2.317, accuracy=10.0%,                        Valid: loss=2.317, accuracy=10.2%
Epoch  1,                        Train:loss=0.666, accuracy=74.2%,                        Valid: loss=0.669, accuracy=74.6%
Epoch  2,                        Train:loss=0.516, accuracy=81.9%,                        Valid: loss=0.526, accuracy=81.4%
Epoch  3,                        Train:loss=0.480, accuracy=82.6%,                        Valid: loss=0.486, accuracy=82.7%
Epoch  4,                        Train:loss=0.427, accuracy=84.7%,                        Valid: loss=0.443, accuracy=84.0%
```

Tableau pour la précision sur l'ensemble de validation

N.B. que les lignes correspondent aux nombre de couche et les colonnes correspondent au nombre de neurone dans chaque couche. Les valeurs ci-dessous sont donné comme exemples; remplacez-les par les valeurs que vous avez utilisées pour votre recherche d'hyperparamètres.

depth\width	25	100	300	500	1000
1	85.443	85.500	87.283	87.617	88.433
3	84.033	86.500	88.283	88.500	87.833
5	83.817	87.667	85.900	88.767	10.250

Comme on peut le voir dans l'expérience précédente, il y a un certains seuil à partir duquel notre gradient explose en devenant trop élevé et notre algorithme n'est tout simplement pas capable d'apprendre des données. De ce fait, on doit ralentir l'apprentissage dans ces cas. Comme le temps pour ce TP est limité, j'ai pris la décision de garder mes données actuelles afin de pouvoir continuer sur les prochaines expériences étant donnée qu'il y a eu qu'un seul cas avec ce problème.

J'ai utilisé les paramètres de **learning_rate** de **0.001** ainsi qu'un **batch_size** de **16** afin d'avoir une grande précision sans prendre trop de temps.

En ce qui concerne nos hyperparamètres pour SGD, il semble y avoir une augmentation de la qualité de notre modèle en ayant un nombre grand comme **width**, mais pour le **depth** il n'y a rien de vraiment concluant.

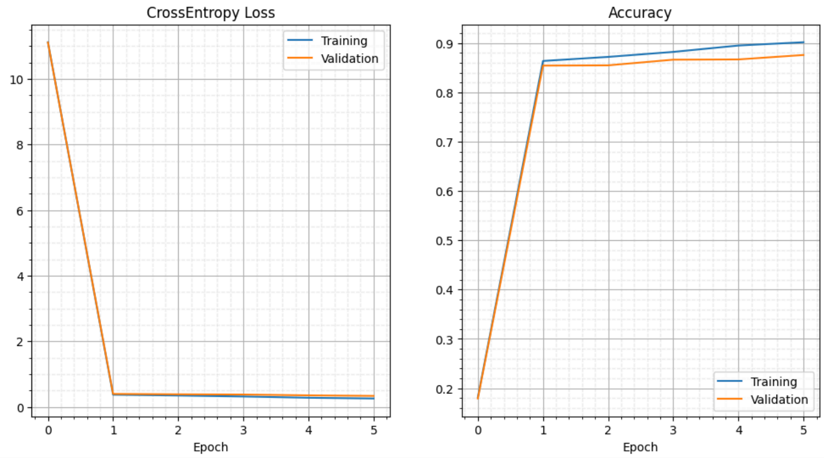
La meilleure valeur de précision que j'ai obtenu se retrouve pour un **width** de **500** et une **depth** de **5** où l'on obtient une précision d'environ **88.767**. Ce sont ces valeurs que nous prendrons pour l'analyse du modèle.

SGD: Analyse du meilleur modèle

```
# SGD
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
depth = 5    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
width = 500    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001          # Some value
batch_size = 16   # Some value

with torch.no_grad():
  data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

  MLP_model = MLPModel(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
  best_model, best_val_accuracy, logger = train(MLP_model,lr=lr, nb_epochs=5, sgd=True,
                                              data_loader_train=data_loader_train, data_loader_val=data_loader_val)
  logger.plot_loss_and_accuracy()
  print(f"Best validation accuracy = {best_val_accuracy*100:.3f}")

  accuracy_test, loss_test = accuracy_and_loss_whole_dataset(data_loader_test, best_model)
print("Evaluation of the best training model over test set")
print("------")
```

```
print(f"Loss : {loss_test:.3f}")
print(f"Accuracy : {accuracy_test*100.:.3f}")
```

```
Teta params=[torch.Size([785, 500]), torch.Size([501, 500]), torch.Size([501, 500]), torch.Size([501, 500]), torch.Size([501, 500]), torch.Size([501, 10])]
Epoch  0,                         Train:loss=11.121, accuracy=18.1%,                         Valid: loss=11.114, accuracy=17.9%
Epoch  1,                         Train:loss=0.370, accuracy=86.4%,                          Valid: loss=0.394, accuracy=85.4%
Epoch  2,                         Train:loss=0.345, accuracy=87.2%,                          Valid: loss=0.382, accuracy=85.5%
Epoch  3,                         Train:loss=0.316, accuracy=88.2%,                          Valid: loss=0.376, accuracy=86.7%
Epoch  4,                         Train:loss=0.277, accuracy=89.5%,                          Valid: loss=0.351, accuracy=86.7%
Epoch  5,                         Train:loss=0.255, accuracy=90.2%,                          Valid: loss=0.334, accuracy=87.6%
Best validation accuracy = 87.600
Evaluation of the best training model over test set
------
Loss : 0.381
Accuracy : 86.330
```



On peut voir dans le graphique que notre entrainement est extrêment efficace dans la première époque, mais se stabilise à partir de cette première époque jusqu'à la fin. Il y a donc un plateau à partir de l'époque 1 et il arrête d'optimiser, soit car il arrête d'apprendre ou qu'il a trouver un minimum valide.

Cependant, notre résultat d'entrainement est plus précis que celle de notre validation. On peut ainsi dire qu'il y a un très léger sur-apprentissage (overfitting) du modèle car celui-ci se comporte mieux sur des données d'entraînement que sur les données de validations. Il aura donc une plus grande difficulté par la suite à apprendre de nouvelle valeurs car il y aura un fort poids déjà présent par l'entrainement qu'il a eu. Dans le même ordre d'idée, l'algorithme n'est assurément pas en train de faire de sous-apprentissage (underfitting) étant dans un contexte inverse actuellement.

Pour ce qui est des résultats, on obtient une **valeur de perte finale** de **0.381** et une **valeur de précision finale** de **86.330**

Adam: Recherche d'hyperparamètres

Implémentez Adam, répétez les deux étapes précédentes (recherche d'hyperparamètres et analyse du meilleur modèle) cette fois en utilisat Adam, et comparez les performances finales avec votre meilleur modèle SGD.

```
# ADAM
# Montrez les résultats pour différents nombre de couche, e.g. 1, 3, 5, et différent nombres de neurone, e.g. 25, 100, 300, 500, 1000.
depth_list = [1, 3, 5]   # Define ranges in a list
width_list = [25, 100, 300, 500, 1000]   # Define ranges in a list
lr = 0.001           # Some value
batch_size = 16   # Some value

with torch.no_grad():
  for depth in depth_list:
    for width in width_list:
      print("-----------------------------------------------------------------")
      print("Training model with a depth of {0} layers and a width of {1} units".format(depth, width))
      data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

      MLP_model = MLPModel(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
      _, val_accuracy, _ = train(MLP_model, lr=lr, nb_epochs=5, sgd=False, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
      print(f"validation accuracy = {val_accuracy*100:.3f}")
```

```
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 25 units
Teta params=[torch.Size([785, 25]), torch.Size([26, 10])]
Epoch  0,                         Train:loss=2.334, accuracy=7.1%,                           Valid: loss=2.332, accuracy=7.5%
Epoch  1,                         Train:loss=0.438, accuracy=84.6%,                          Valid: loss=0.461, accuracy=84.1%
Epoch  2,                         Train:loss=0.388, accuracy=85.9%,                          Valid: loss=0.414, accuracy=85.2%
Epoch  3,                         Train:loss=0.358, accuracy=87.0%,                          Valid: loss=0.396, accuracy=85.3%
Epoch  4,                         Train:loss=0.336, accuracy=88.0%,                          Valid: loss=0.371, accuracy=86.7%
Epoch  5,                         Train:loss=0.319, accuracy=88.4%,                          Valid: loss=0.361, accuracy=86.8%
validation accuracy = 86.767
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 100 units
Teta params=[torch.Size([785, 100]), torch.Size([101, 10])]
Epoch  0,                         Train:loss=2.799, accuracy=3.4%,                           Valid: loss=2.796, accuracy=3.4%
Epoch  1,                         Train:loss=0.424, accuracy=83.8%,                          Valid: loss=0.433, accuracy=83.6%
Epoch  2,                         Train:loss=0.330, accuracy=88.0%,                          Valid: loss=0.353, accuracy=87.4%
Epoch  3,                         Train:loss=0.317, accuracy=88.5%,                          Valid: loss=0.344, accuracy=87.7%
Epoch  4,                         Train:loss=0.282, accuracy=89.6%,                          Valid: loss=0.330, accuracy=88.5%
Epoch  5,                         Train:loss=0.273, accuracy=89.8%,                          Valid: loss=0.325, accuracy=88.6%
validation accuracy = 88.617
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 300 units
Teta params=[torch.Size([785, 300]), torch.Size([301, 10])]
Epoch  0,                         Train:loss=3.227, accuracy=12.5%,                          Valid: loss=3.223, accuracy=12.4%
Epoch  1,                         Train:loss=0.370, accuracy=86.8%,                          Valid: loss=0.387, accuracy=85.7%
Epoch  2,                         Train:loss=0.337, accuracy=87.7%,                          Valid: loss=0.362, accuracy=86.8%
Epoch  3,                         Train:loss=0.291, accuracy=89.2%,                          Valid: loss=0.329, accuracy=87.8%
Epoch  4,                         Train:loss=0.267, accuracy=90.2%,                          Valid: loss=0.314, accuracy=88.6%
Epoch  5,                         Train:loss=0.261, accuracy=90.2%,                          Valid: loss=0.320, accuracy=88.3%
validation accuracy = 88.550
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 500 units
Teta params=[torch.Size([785, 500]), torch.Size([501, 10])]
Epoch  0,                         Train:loss=3.705, accuracy=15.8%,                          Valid: loss=3.703, accuracy=15.5%
Epoch  1,                         Train:loss=0.378, accuracy=85.8%,                          Valid: loss=0.394, accuracy=85.2%
Epoch  2,                         Train:loss=0.321, accuracy=88.1%,                          Valid: loss=0.348, accuracy=87.6%
Epoch  3,                         Train:loss=0.309, accuracy=88.4%,                          Valid: loss=0.354, accuracy=87.3%
Epoch  4,                         Train:loss=0.283, accuracy=89.5%,                          Valid: loss=0.335, accuracy=87.9%
Epoch  5,                         Train:loss=0.270, accuracy=90.0%,                          Valid: loss=0.321, accuracy=88.7%
validation accuracy = 88.683
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 1000 units
Teta params=[torch.Size([785, 1000]), torch.Size([1001, 10])]
Epoch  0,                         Train:loss=3.965, accuracy=7.5%,                           Valid: loss=3.948, accuracy=8.1%
```

```
Epoch  1,                         Train:loss=0.448, accuracy=84.7%,                                Valid: loss=0.465, accuracy=84.7%
Epoch  2,                         Train:loss=0.296, accuracy=89.3%,                                Valid: loss=0.327, accuracy=88.2%
Epoch  3,                         Train:loss=0.286, accuracy=89.5%,                                Valid: loss=0.334, accuracy=87.9%
Epoch  4,                         Train:loss=0.268, accuracy=90.4%,                                Valid: loss=0.325, accuracy=88.8%
Epoch  5,                         Train:loss=0.271, accuracy=90.1%,                                Valid: loss=0.355, accuracy=88.0%
validation accuracy = 88.800
-----------------------------------------------------------------
Training model with a depth of 3 layers and a width of 25 units
Teta params=[torch.Size([785, 25]), torch.Size([26, 25]), torch.Size([26, 10])]
Epoch  0,                         Train:loss=2.304, accuracy=14.7%,                                Valid: loss=2.305, accuracy=14.9%
Epoch  1,                         Train:loss=0.483, accuracy=82.9%,                                Valid: loss=0.500, accuracy=82.6%
Epoch  2,                         Train:loss=0.407, accuracy=85.3%,                                Valid: loss=0.436, accuracy=84.4%
Epoch  3,                         Train:loss=0.379, accuracy=85.9%,                                Valid: loss=0.408, accuracy=85.7%
Epoch  4,                         Train:loss=0.338, accuracy=87.6%,                                Valid: loss=0.377, accuracy=86.7%
```

Tableau pour la précision sur l'ensemble de validation

N.B. que les lignes correspondent aux nombre de couche et les colonnes correspondent au nombre de neurone dans chaque couche. Les valeurs ci-dessous sont donné comme exemples; remplacez-les par les valeurs que vous avez utilisées pour votre recherche d'hyperparamètres.

depth\width	25	100	300	500	1000
1	86.767	88.617	88.550	88.683	88.800
3	86.733	88.550	87.933	87.417	88.533
5	86.383	88.300	87.667	88.050	86.733

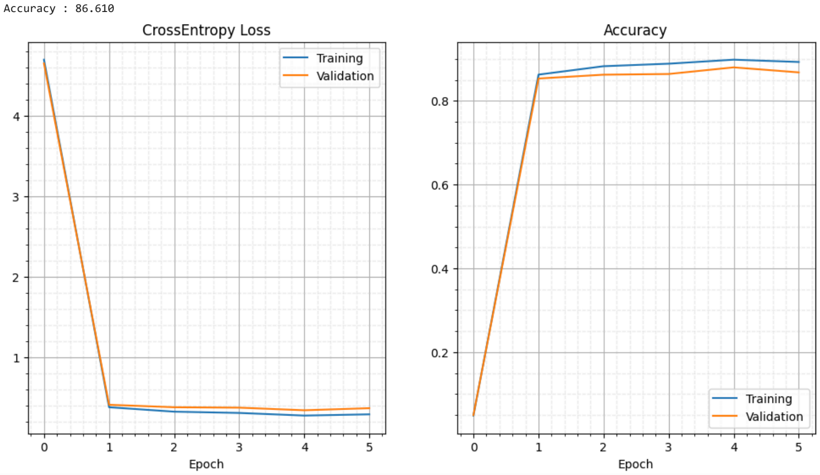
En ce qui concerne nos hyperparamètres pour Adam, il semble y avoir une seuil à partir d'une **width** de **100** et il ne semble pas y avoir de tendance pour la variable de **depth**. J'ai toujours utilisé les mêmes paramètres de **learning_rate** ainsi que de **batch_size** utilisé avec SGD afin de faire une meilleure comparaison.

La meilleure valeur de précision que j'ai obtenu se retrouve pour un **width** de **1000** et une **depth** de **1** où l'on obtient une précision d'environ **88.800**. Ce sont ces valeurs que nous prendrons pour l'analyse du modèle Adam.

Adam: Analyse du meilleur modèle

```
# ADAM
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
depth = 1    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
width = 1000    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001           # Some value
batch_size = 16   # Some value

with torch.no_grad():
  data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

  MLP_model = MLPModel(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
  best_model, best_val_accuracy, logger = train(MLP_model,lr=lr, nb_epochs=5, sgd=False,
                                               data_loader_train=data_loader_train, data_loader_val=data_loader_val)
  logger.plot_loss_and_accuracy()
  print(f"Best validation accuracy = {best_val_accuracy*100:.3f}")

  accuracy_test, loss_test = accuracy_and_loss_whole_dataset(data_loader_test, best_model)
print("Evaluation of the best training model over test set")
print("------")
print(f"Loss : {loss_test:.3f}")
print(f"Accuracy : {accuracy_test*100.:.3f}")
```

```
Teta params=[torch.Size([785, 1000]), torch.Size([1001, 10])]
Epoch  0,                         Train:loss=4.698, accuracy=4.9%,                                 Valid: loss=4.651, accuracy=5.0%
Epoch  1,                         Train:loss=0.379, accuracy=86.3%,                                Valid: loss=0.409, accuracy=85.3%
Epoch  2,                         Train:loss=0.324, accuracy=88.3%,                                Valid: loss=0.379, accuracy=86.3%
Epoch  3,                         Train:loss=0.308, accuracy=88.9%,                                Valid: loss=0.374, accuracy=86.4%
Epoch  4,                         Train:loss=0.276, accuracy=89.8%,                                Valid: loss=0.342, accuracy=88.0%
Epoch  5,                         Train:loss=0.291, accuracy=89.3%,                                Valid: loss=0.367, accuracy=86.8%
Best validation accuracy = 88.000
Evaluation of the best training model over test set
------
Loss : 0.387
Accuracy : 86.610
```



On peut voir dans le graphique que notre entrainement est extrêment efficace dans la première époque, mais se stabilise à partir de cette première époque jusqu'à la fin. Il y a donc un plateau à partir de l'époque 1 et il arrête d'optimiser, soit car il arrête d'apprendre ou qu'il a trouver un minimum valide.

Il ne semble pas non plus avoir de sur=apprentissage (overfitting) ou de sous-apprentissage (underfitting) étant donnée que la précision est sensiblement pareil pour l'entraînement et la validation.

Pour ce qui est des résultats, on obtient une **valeur de perte finale** de **0.387** et une **valeur de précision finale** de **86.610**

Analyse des Résultats

```
import torch.nn as nn
import torch.optim as optim

class MLPAutograd(nn.Module):
    def __init__(self, n_features, n_hidden_features, n_hidden_layers, n_classes):
        super(MLPAutograd, self).__init__()
        self.flatten = nn.Flatten()

        self.n_features        = n_features
```

```
        self.n_hidden_features = n_hidden_features
        self.n_hidden_layers   = n_hidden_layers
        self.n_classes         = n_classes

        layers = []
        layers.append(nn.Linear(n_features, n_hidden_features))
        layers.append(nn.ReLU())

        for _ in range(n_hidden_layers - 1):
            layers.append(nn.Linear(n_hidden_features, n_hidden_features))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(n_hidden_features, n_classes))
        layers.append(nn.Softmax(dim=1))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        x = self.flatten(x)
        return self.network(x)

def test_autograd(dataloader, model, loss_fn):
    model.eval()
    cardinal, loss, n_accurate_preds = 0, 0, 0
    with torch.no_grad():
        for x, y in dataloader:
            x, y = reshape_input(x, y)
            y_pred = model(x)
            xentrp = loss_fn(y_pred, y)
            _, (n_acc, n_samples) = accuracy(y, y_pred)

            cardinal = cardinal + n_samples
            loss     = loss + xentrp
            n_accurate_preds  = n_accurate_preds + n_acc

    loss = loss / float(cardinal)
    acc  = n_accurate_preds / float(cardinal)

    return acc, loss

def train_autograd(model, lr=0.1, nb_epochs=10, data_loader_train=None, data_loader_val=None):
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    best_model = None
    best_val_accuracy = 0
    logger = Logger()

    for epoch in range(nb_epochs+1):
        if epoch > 0:
            model.train()
            for x, y in data_loader_train:
                x, y = reshape_input(x, y)
                y_pred = model(x)
                loss = loss_fn(y_pred, y)

                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

        model.eval()
        accuracy_train, loss_train = test_autograd(data_loader_train, model, loss_fn)
        accuracy_val, loss_val = test_autograd(data_loader_val, model, loss_fn)

        if accuracy_val > best_val_accuracy:
            best_model = model
            best_val_accuracy = accuracy_val

        logger.log(accuracy_train, loss_train, accuracy_val, loss_val)
        print(f"Epoch {epoch:2d}, \
                Train:loss={loss_train.item():.3f}, accuracy={accuracy_train.item()*100:.1f}%, \
                Valid: loss={loss_val.item():.3f}, accuracy={accuracy_val.item()*100:.1f}%", flush=True)

    return best_model, best_val_accuracy, logger
```

```
# Autograd
# Montrez les résultats pour différents nombre de couche, e.g. 1, 3, 5, et différent nombres de neurone, e.g. 25, 100, 300, 500, 1000.
depth_list = [1, 3, 5]   # Define ranges in a list
width_list = [25, 100, 300, 500, 1000]   # Define ranges in a list
lr = 0.001           # Some value
batch_size = 16   # Some value

for depth in depth_list:
    for width in width_list:
        print("------------------------------------------------------------------")
        print("Training model with a depth of {0} layers and a width of {1} units".format(depth, width))
        data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

        model = MLPAutograd(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
        _, val_accuracy, _ = train_autograd(model, lr=lr, nb_epochs=5, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
        print(f"validation accuracy = {val_accuracy*100:.3f}")
```

```
------------------------------------------------------------------
Training model with a depth of 1 layers and a width of 25 units
Epoch  0,                          Train:loss=0.144, accuracy=10.2%,                               Valid: loss=0.144, accuracy=9.9%
Epoch  1,                          Train:loss=0.143, accuracy=21.5%,                               Valid: loss=0.143, accuracy=21.7%
Epoch  2,                          Train:loss=0.142, accuracy=37.2%,                               Valid: loss=0.141, accuracy=38.5%
Epoch  3,                          Train:loss=0.138, accuracy=41.7%,                               Valid: loss=0.138, accuracy=42.3%
Epoch  4,                          Train:loss=0.133, accuracy=37.9%,                               Valid: loss=0.132, accuracy=38.6%
Epoch  5,                          Train:loss=0.129, accuracy=44.7%,                               Valid: loss=0.128, accuracy=45.5%
validation accuracy = 45.467
------------------------------------------------------------------
Training model with a depth of 1 layers and a width of 100 units
Epoch  0,                          Train:loss=0.144, accuracy=4.0%,                                Valid: loss=0.144, accuracy=4.2%
Epoch  1,                          Train:loss=0.143, accuracy=33.0%,                               Valid: loss=0.143, accuracy=33.0%
Epoch  2,                          Train:loss=0.140, accuracy=35.6%,                               Valid: loss=0.140, accuracy=35.6%
Epoch  3,                          Train:loss=0.134, accuracy=53.6%,                               Valid: loss=0.134, accuracy=52.8%
Epoch  4,                          Train:loss=0.127, accuracy=54.6%,                               Valid: loss=0.127, accuracy=53.6%
Epoch  5,                          Train:loss=0.123, accuracy=55.1%,                               Valid: loss=0.124, accuracy=54.2%
validation accuracy = 54.217
------------------------------------------------------------------
Training model with a depth of 1 layers and a width of 300 units
Epoch  0,                          Train:loss=0.144, accuracy=7.3%,                                Valid: loss=0.144, accuracy=7.7%
Epoch  1,                          Train:loss=0.143, accuracy=11.7%,                               Valid: loss=0.143, accuracy=12.2%
Epoch  2,                          Train:loss=0.141, accuracy=27.2%,                               Valid: loss=0.141, accuracy=27.8%
Epoch  3,                          Train:loss=0.135, accuracy=43.3%,                               Valid: loss=0.135, accuracy=44.2%
Epoch  4,                          Train:loss=0.129, accuracy=51.7%,                               Valid: loss=0.128, accuracy=52.7%
Epoch  5,                          Train:loss=0.124, accuracy=54.9%,                               Valid: loss=0.124, accuracy=55.9%
validation accuracy = 55.883
------------------------------------------------------------------
Training model with a depth of 1 layers and a width of 500 units
Epoch  0,                          Train:loss=0.144, accuracy=2.3%,                                Valid: loss=0.144, accuracy=2.1%
Epoch  1,                          Train:loss=0.142, accuracy=43.3%,                               Valid: loss=0.142, accuracy=43.7%
Epoch  2,                          Train:loss=0.138, accuracy=37.8%,                               Valid: loss=0.138, accuracy=39.0%
```

```
Epoch  3,                         Train:loss=0.131, accuracy=45.5%,                          Valid: loss=0.131, accuracy=46.8%
Epoch  4,                         Train:loss=0.126, accuracy=53.9%,                          Valid: loss=0.126, accuracy=55.1%
Epoch  5,                         Train:loss=0.123, accuracy=55.0%,                          Valid: loss=0.123, accuracy=56.0%
validation accuracy = 56.017
-----------------------------------------------------------------
Training model with a depth of 1 layers and a width of 1000 units
Epoch  0,                         Train:loss=0.144, accuracy=12.9%,                          Valid: loss=0.144, accuracy=12.1%
Epoch  1,                         Train:loss=0.141, accuracy=30.6%,                          Valid: loss=0.141, accuracy=29.9%
Epoch  2,                         Train:loss=0.134, accuracy=39.8%,                          Valid: loss=0.134, accuracy=38.7%
Epoch  3,                         Train:loss=0.128, accuracy=52.3%,                          Valid: loss=0.128, accuracy=51.7%
Epoch  4,                         Train:loss=0.123, accuracy=58.2%,                          Valid: loss=0.123, accuracy=58.1%
Epoch  5,                         Train:loss=0.120, accuracy=62.8%,                          Valid: loss=0.120, accuracy=62.9%
validation accuracy = 62.933
-----------------------------------------------------------------
Training model with a depth of 3 layers and a width of 25 units
Epoch  0,                         Train:loss=0.144, accuracy=10.0%,                          Valid: loss=0.144, accuracy=10.4%
Epoch  1,                         Train:loss=0.144, accuracy=10.0%,                          Valid: loss=0.144, accuracy=10.4%
Epoch  2,                         Train:loss=0.144, accuracy=10.0%,                          Valid: loss=0.144, accuracy=10.4%
Epoch  3,                         Train:loss=0.144, accuracy=10.0%,                          Valid: loss=0.144, accuracy=10.4%
Epoch  4,                         Train:loss=0.144, accuracy=12.4%,                          Valid: loss=0.144, accuracy=12.8%
Epoch  5,                         Train:loss=0.144, accuracy=17.0%,                          Valid: loss=0.144, accuracy=17.5%
validation accuracy = 17.533
-----------------------------------------------------------------
Training model with a depth of 3 layers and a width of 100 units
Epoch  0,                         Train:loss=0.144, accuracy=3.6%,                           Valid: loss=0.144, accuracy=3.2%
Epoch  1,                         Train:loss=0.144, accuracy=15.2%,                          Valid: loss=0.144, accuracy=15.0%
```

Tableau pour la précision sur l'ensemble de validation

N.B. que les lignes correspondent aux nombre de couche et les colonnes correspondent au nombre de neurone dans chaque couche. Les valeurs ci-dessous sont donné comme exemples; remplacez-les par les valeurs que vous avez utilisées pour votre recherche d'hyperparamètres.

depth\width	25	100	300	500	1000
1	45.467	54.217	55.883	56.017	62.933
3	17.533	23.833	24.833	12.683	21.200
5	10.217	10.250	9.917	11.383	10.000

En ce qui concerne nos hyperparamètres pour Autograd, il semble y avoir une seuil à partir d'un **depth** de **1** qui semble ralentir énormément notre apprentissage. Il est très possible que cela soit dû au **learning_rate** assez bas. Du côté du **width**, il n'y a pas vraiment de lien avec les données. On peut voir une augmentation lorsque notre apprentissage fonctionne bien, mais cela n'est pas vraiment suffisant pour l'affirmer. J'ai toujours utilisé les mêmes paramètres de **learning_rate** ainsi que de **batch_size** utilisé avec SGD afin de faire une meilleure comparaison.

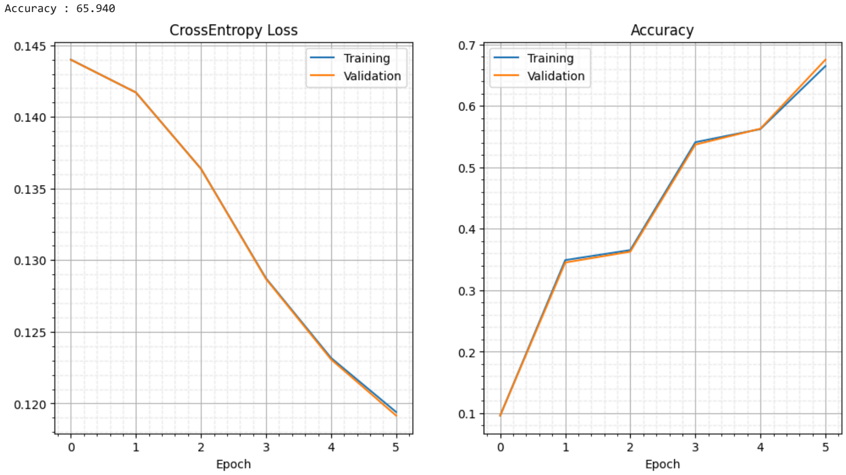
On peut aussi voir que l'apprentissage semble beaucoup plus lent avec ce modèle. Même lorsque notre précision finale semble un peu bonne, peut être que continuer sur plus d'époques aurait pu aider notre modèle à bien performé. On peut aussi remarqué qu'il semble y avoir un seuil sur le loss. Je ne sais pas exactement pourquoi ce comportement arrive, mais il semble justement ralentir notre apprentissage.

La meilleure valeur de précision que j'ai obtenu se retrouve pour un **width** de **1000** et une **depth** de **1** où l'on obtient une précision d'environ **62.933**. Ce sont ces valeurs que nous prendrons pour l'analyse du modèle Autograd.

Autograd: Analyse du meilleur modèle

```python
# Autograd
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
depth = 1    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
width = 1000    # Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001           # Some value
batch_size = 16   # Some value

data_loader_train, data_loader_val, data_loader_test = get_fashion_mnist_dataloaders(val_percentage=0.1, batch_size=batch_size)

model = MLPAutograd(n_features=784, n_hidden_features=width, n_hidden_layers=depth, n_classes=10)
best_model, best_val_accuracy, logger = train_autograd(model,lr=lr, nb_epochs=5, data_loader_train=data_loader_train, data_loader_val=data_loader_val)
logger.plot_loss_and_accuracy()
print(f"Best validation accuracy = {best_val_accuracy*100:.3f}")

accuracy_test, loss_test = test_autograd(data_loader_test, best_model, nn.CrossEntropyLoss())
print("Evaluation of the best training model over test set")
print("------")
print(f"Loss : {loss_test:.3f}")
print(f"Accuracy : {accuracy_test*100.:.3f}")
```

```
Epoch  0,                         Train:loss=0.144, accuracy=9.6%,                           Valid: loss=0.144, accuracy=9.6%
Epoch  1,                         Train:loss=0.142, accuracy=34.9%,                          Valid: loss=0.142, accuracy=34.5%
Epoch  2,                         Train:loss=0.136, accuracy=36.5%,                          Valid: loss=0.136, accuracy=36.3%
Epoch  3,                         Train:loss=0.129, accuracy=54.1%,                          Valid: loss=0.129, accuracy=53.7%
Epoch  4,                         Train:loss=0.123, accuracy=56.2%,                          Valid: loss=0.123, accuracy=56.3%
Epoch  5,                         Train:loss=0.119, accuracy=66.5%,                          Valid: loss=0.119, accuracy=67.5%
Best validation accuracy = 67.517
Evaluation of the best training model over test set
------
Loss : 0.120
Accuracy : 65.940
```



On peut voir dans le graphique que notre entrainement est beaucoup plus proportionné à l'ensemble des époque. Il y a donc un plateau à partir de l'époque 1 et il arrête d'optimiser, soit car il arrête d'apprendre ou qu'il a trouver un minimum valide. On voit aussi un certain ralentissement plus l'apprentissage a progressé dans les époques.

Il n'y a ainsi aucun sur-apprentissage (overfitting) ni sous-apprentissage (underfitting) comme l'apprentissage ne semble pas complètement terminé.

Pour ce qui est des résultats, on obtient une **valeur de perte finale** de **0.120** et une **valeur de précision finale** de **65.940**

Au final, il y a une grande différence dans chacun des modèles testés. Du côté du SGD, on a un apprentissage un peu moins précis que Adam, mais plus rapide que ce dernier. Du côté de Autograd, il y a un apprentissage beaucoup plus lent en terme de précision, mais qui est beaucoup plus stable et ne semble pas affecté par l'explosion des gradients qu'on a eu dans les autres modèles.