POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

INF8225 – Intelligence artificielle : techniques probabilistes et d'apprentissage

Hiver 2022

TP 1

Morgan PEJU - *2103232*

Date de rendu
28 février 2022

LIEN COLAB :
https://colab.research.google.com/drive/1BQ_LSz2wb39Yz7tlA6h_9_7w6wVPgGkd?usp=sharing

1. Partie 1 (10 points)

1.1. Avec les tables de probabilité conditionnelles, calculons les requêtes :

Les résultats ont été obtenus via Python (voir Notebook ou Colab, ou Code en annexe)

a) $\Pr(H = 1)$

Avec la règle de la somme et des probabilités jointes dans un réseau bayésien, on a :

$$\Pr(H = 1) = \sum_N \sum_P \sum_A \sum_W \Pr(N, P, A, W, H = 1)$$
$$= \sum_N \sum_P \sum_A \sum_W \Pr(N) \Pr(P|N) \Pr(A|N) \Pr(W|P) \ \Pr(H = 1|P, A)$$
$$= 0.36520$$

b) $\Pr(H = 1|A = 1)$

Avec la règle du produit, la règle de la somme et des probabilités jointes dans un réseau bayésien, on a :

$$\Pr(H = 1|A = 1) = \frac{\sum_N \sum_P \sum_W \Pr(N, P, A = 1, W, H = 1)}{\Pr(A = 1)}$$
$$= \frac{\sum_N \sum_P \sum_W \Pr(N) \Pr(P|N) \Pr(A = 1|N) \Pr(W|P) \ \Pr(H = 1|P, A = 1)}{\Pr(A = 1)} = 0.91231$$

c) $\Pr(H = 1|do(A = 1))$

Ici, il faut prendre en compte la situation interventionnelle, on a :

$$\Pr(H = 1|do(A = 1)) = \sum_N \sum_P \sum_W \Pr(N, P, W, H = 1|do(A = 1))$$
$$= \sum_N \sum_P \sum_W \Pr(N) \Pr(P|N) \Pr(W|P) \ \Pr(H = 1|P, do(A = 1))$$
$$= 0.916$$

d) $\Pr(H = 1|W = 1)$

Avec la règle du produit, la règle de la somme et des probabilités jointes dans un réseau bayésien, on a :

$$\Pr(H = 1|W = 1) = \frac{\sum_N \sum_P \sum_A \Pr(N, P, A, W = 1, H = 1)}{\Pr(W = 1)}$$
$$= \frac{\sum_N \sum_P \sum_A \Pr(N) \Pr(P|N) \Pr(A|N) \Pr(W = 1|P) \ \Pr(H = 1|P, A)}{\sum_P [\Pr(W = 1|P) * \sum_N \Pr(N) \Pr(P|N)]}$$
$$= 0.61293$$

e) $\underline{\Pr(H = 1|do(W = 1))}$

Ici, il faut prendre en compte la situation interventionnelle, on a :

$$\Pr(H = 1|do(W = 1)) = \sum_N \sum_P \sum_A \Pr(N, P, A, H = 1|do(W = 1))$$
$$= \sum_N \sum_P \sum_A \Pr(N) \Pr(P|N) \Pr(A|N)\ \Pr(H = 1|P, A) = 0.36520$$

f) $\underline{\Pr(W = 1|P = 1)}$

Avec la règle du produit, la règle de la somme et des probabilités jointes dans un réseau bayésien, on a :

$$\Pr(W = 1|P = 1) = \frac{\sum_N \sum_A \sum_H \Pr(N, P = 1, A, W = 1, H)}{\Pr(P = 1)}$$
$$= \frac{\sum_N \sum_A \sum_H \Pr(N) \Pr(P = 1|N) \Pr(A|N) \Pr(W = 1|P = 1)\ \Pr(H|P = 1, A)}{\sum_N \Pr(N) \Pr(P = 1|N)}$$
$$= 1$$

g) $\underline{\Pr(W = 1|do(P = 1))}$

Ici, il faut prendre en compte la situation interventionnelle, on a :

$$\Pr(W = 1|do(P = 1)) = \sum_N \sum_A \sum_H \Pr(N, A, W = 1, H|do(P = 1))$$
$$= \sum_N \sum_A \sum_H \Pr(N) \Pr(W = 1|P = 1) \Pr(A|N)\ \Pr(H|do(P = 1), A) = 1$$

h) $\underline{\Pr(H = 1|P = 1)}$

Avec la règle du produit, la règle de la somme et des probabilités jointes dans un réseau bayésien, on a :

$$\Pr(H = 1|P = 1) = \frac{\sum_N \sum_A \sum_W \Pr(N, P = 1, A, W, H = 1)}{\Pr(P = 1)}$$
$$= \frac{\sum_N \sum_A \sum_W \Pr(N) \Pr(P = 1|N) \Pr(A|N) \Pr(W|P = 1)\ \Pr(H = 1|P = 1, A)}{\sum_N \Pr(N) \Pr(P = 1|N)}$$
$$= 1$$

i) $\underline{\Pr(H = 1|do(P = 1))}$

Ici, il faut prendre en compte la situation interventionnelle, on a :

$$\Pr(H = 1|do(P = 1)) = \sum_N \sum_A \sum_W \Pr(N, A, W, H = 1|do(P = 1))$$
$$= \sum_N \sum_A \sum_W \Pr(N) \Pr(W|P = 1) \Pr(A|N)\ \Pr(H = 1|do(P = 1), A) = 1$$

j) $\underline{\Pr(P = 1|W = 1, H = 1, N = 1)}$

On a :

$$\Pr(P=1|W=1,H=1,N=1)=\frac{\sum_A \Pr(N=1,P=1,A,W=1,H=1)}{\sum_P\sum_A \Pr(N=1,P,A,W=1,H=1)}$$
$$=\frac{\sum_A \Pr(N=1)\Pr(P=1|N=1)\Pr(A|N=1)\Pr(W=1|P=1)\ \Pr(H=1|P=1,A)}{\sum_P\sum_A \Pr(N=1)\Pr(P|N=1)\Pr(A|N=1)\Pr(W=1|P)\ \Pr(H=1|P,A)}$$
$$=0.97371$$

1.2. Questions

a) Vrai ou Faux :

On notera (X) pour signifier « sachant X ».

i. $H \perp\!\!\!\perp N|P$: **FAUX**

Il faut considérer tous les chemins possibles de H à N. Si tous les chemins de H à N sont bloqués, alors on dira que H est d-séparé de N par P.
Ici, on a le chemin N-> A ->H qui est connecté. Donc tous les chemins de H à N ne sont pas bloqués, alors $H \perp\!\!\!\perp N|P$ est **FAUX.**

ii. $H \perp\!\!\!\perp N|A$: **FAUX**

Ici, on a le chemin N -> P -> H qui est connecté. Donc tous les chemins de H à N ne sont pas bloqués, alors $H \perp\!\!\!\perp N|A$ est **FAUX.**

iii. $W \perp\!\!\!\perp H|P$: **VRAI**

Ici, on a les chemins suivants :
- {W,P,H} qui est orienté de la manière suivant : W <- (P) -> H. C'est le cas d'influence divergente, le chemin est donc bloqué.
- {W,P,N,A,H}. On décompose par triplet : {W,P,N}, {P,N,A} et {N,A,H}. Or le chemin {W,P,N} est orienté comme ceci : N-> (P) ->W. Ce triplet est donc séparé (influence pipelinée). Donc l'entièreté du chemin est bloqué.

Ainsi, tous les chemins de W à H sont bloqués. Donc W est d-séparé de H par P. → **VRAI**

iv. $P \perp\!\!\!\perp A|N$: **VRAI**

Ici, on a les chemins suivants :
- {P,H,A} qui est orienté de la manière suivant : P -> H <- A. Le chemin est donc bloqué.
- {P,N,A} qui est orienté de la manière suivant : P <- (N) -> A (influence divergente). Le chemin est donc bloqué.

Ainsi, tous les chemins de P à A sont bloqués. Donc P est d-séparé de A par N. → **VRAI**

v. $P \perp\!\!\!\perp A|N,H$: **FAUX**

Ici, on a le chemin P -> (H) <- A qui est connecté (influence convergente). Donc tous les chemins de P à A ne sont pas bloqués, alors $H \perp\!\!\!\perp N|P$ est **FAUX.**

vi. $H \perp\!\!\!\perp N|A$ → **même réponse que ii)**

b) Explications

i. Pourquoi est-ce que $\Pr(W|P)=\Pr\big(W|do(P)\big)$?

On sait de manière générale que do(X=x) a le même effet sur Y qu'une observation passive X=x lorsqu'on a la condition (suffisante) que Y est d-séparé des parents de X sachant X.

Or dans le cas ici, N est le seul parent de P. On a comme chemin de W à N le chemin :
N -> (P) -> W (influence pipelinée) donc le chemin est bloqué.
On a aussi le chemin {N,A,H,P,W} qui décomposé en triplet donne : N -> A -> H (connecté) et W <- (P) -> H (chemin bloqué car influence divergente).
Ainsi, tous les chemins de W à N (parents de P) sont bloqués, donc on a $W \perp\!\!\!\perp Parents(P) | P$.
La condition est remplie donc on a bien $\Pr(W|P) = \Pr\big(W|do(P)\big)$.

ii. Pourquoi est-ce que $\Pr(H|A) \neq \Pr(H|do(A))$?

Dans le cas ici, N est le seul parent de A. Or on sait d'après la question ii) précédente que $H \perp\!\!\!\perp N | A$ est **FAUX.** La condition n'est pas remplie donc on a bien $\Pr(H|A) \neq \Pr(H|do(A))$.

2. Partie 2 (20 points)

2.1. Code à compléter

Le code de la partie 2 est à retrouver en annexe, via le notebook dans le .zip ou sur Colab : https://colab.research.google.com/drive/1BQ_LSz2wb39Yz7tlA6h_9_7w6wVPgGkd?usp=sharing

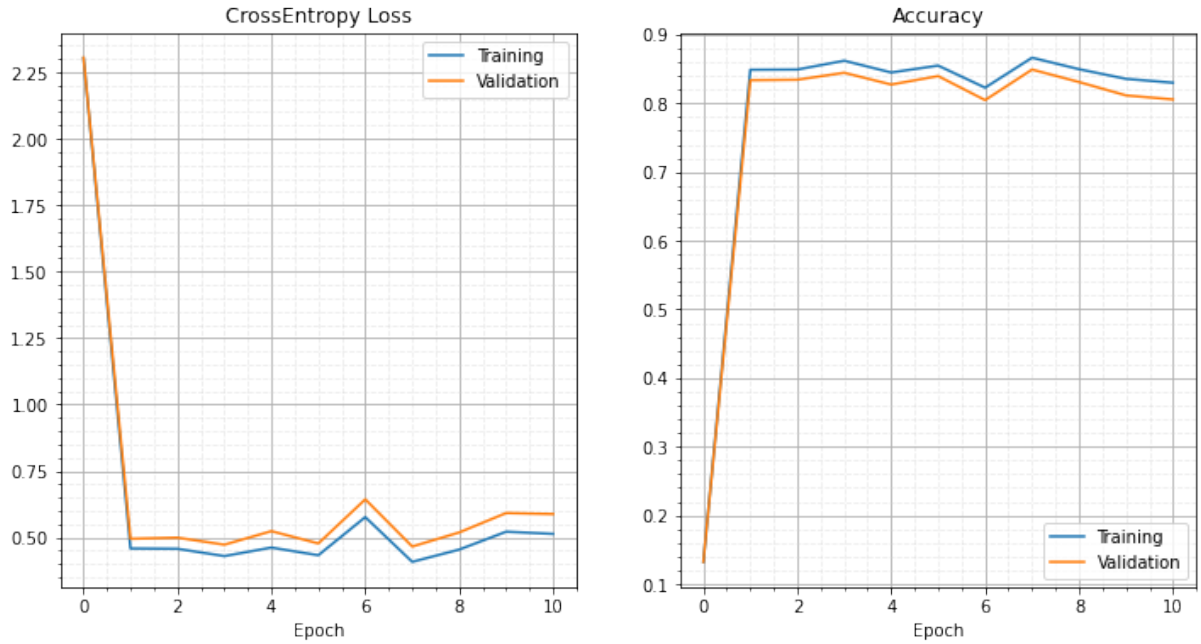
2.2. Analyse des résultats

Afin de pouvoir reproduire les mêmes résultats à chaque fois, j'ai fixé un seed(2103232).
SGD :
Après l'implémentation du code, j'ai pu obtenir les résultats de précisions suivants en fonction de la taille du batch et du taux d'apprentissage :

learning rate\batch_size	5	20	200	1000
0.1	83.217%	82.100%	81.900%	80.217%
0.01	**84.450%**	82.950%	83.250%	78.333%
0.001	83.917%	83.967%	82.883%	77.350%

On peut voir qu'on obtient une précision autour des 80% dans tous les cas. Seules les paires (batch=1000, lr=0.01) et (batch=1000, lr=0.001) donnent des résultats légèrement inférieurs à 80% (respectivement 78.3% et 77.4%). On notera donc que la taille du batch a également son influence sur la précision. Toutefois, on remarque logiquement que plus le batch est grand, plus la vitesse d'exécution est rapide. C'est pourquoi j'ai écarté le cas « batch-size=1 » pour le remplacer par 5 pour avoir un temps de calcul plus raisonnable.
On obtient alors le meilleur résultat pour batch-size=5 et taux d'apprentissage égale à 0.01.

En utilisant ce résultat pour tester le meilleur modèle, on obtient les courbes suivantes :



Graphiques – (à gauche) La perte d'entropie-croisée en fonction de l'epoch, (à droite) la précision en fonction de l'epoch

On remarque que notre modèle apprend bien. En effet, à l'epoch 0, le modèle effectue la première propagation avec les paramètres aléatoires initiaux ce qui explique la perte élevée et la précision de 10%. Puis la modèle apprend en actualisant les paramètres par la minimisation de l'entropie croisée. On remarque bien que l'entropie croisée diminue autour de 0.5. La précision est en effet bien meilleure, avec une précision maximale de validation atteinte à l'epoch 7 avec 84.917%.
Voici le détail par epoch :

```
Epoch  0,                  Train: loss=2.303, accuracy=13.2%,                  Valid: loss=2.305, accuracy=13.2%
Epoch  1,                  Train: loss=0.458, accuracy=84.9%,                  Valid: loss=0.495, accuracy=83.4%
Epoch  2,                  Train: loss=0.457, accuracy=84.9%,                  Valid: loss=0.499, accuracy=83.4%
Epoch  3,                  Train: loss=0.430, accuracy=86.2%,                  Valid: loss=0.473, accuracy=84.5%
Epoch  4,                  Train: loss=0.462, accuracy=84.5%,                  Valid: loss=0.524, accuracy=82.7%
Epoch  5,                  Train: loss=0.433, accuracy=85.5%,                  Valid: loss=0.477, accuracy=84.0%
Epoch  6,                  Train: loss=0.576, accuracy=82.3%,                  Valid: loss=0.643, accuracy=80.5%
Epoch  7,                  Train: loss=0.408, accuracy=86.6%,                  Valid: loss=0.466, accuracy=84.9%
Epoch  8,                  Train: loss=0.454, accuracy=85.0%,                  Valid: loss=0.519, accuracy=83.1%
Epoch  9,                  Train: loss=0.522, accuracy=83.6%,                  Valid: loss=0.592, accuracy=81.2%
Epoch 10,                  Train: loss=0.513, accuracy=83.0%,                  Valid: loss=0.588, accuracy=80.6%
Best validation accuracy = 84.917
Evaluation of the best training model over test set
------
Loss : 0.618
Accuracy : 80.190
```

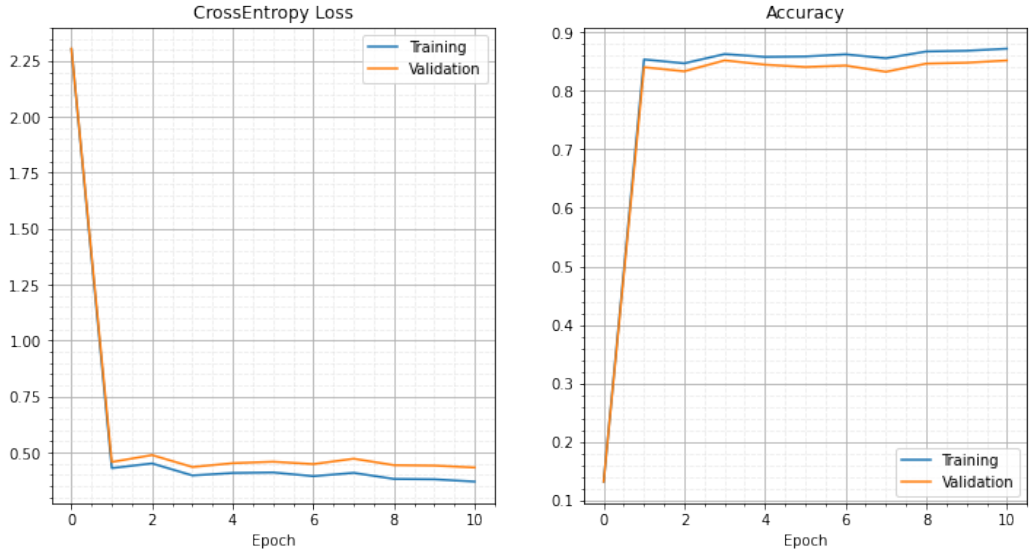
ADAM :

Cette fois-ci en utilisant ADAM pour l'actualisation, j'ai pu obtenir les résultats de précisions suivants en fonction de la taille du batch et du taux d'apprentissage :

learning rate\batch_size	5	20	200	1000
0.1	82.050%	82.117%	82.783%	82.067%
0.01	82.283%	83.083%	84.350%	84.183%
0.001	**85.183%**	84.933%	83.850%	81.283%

On peut voir qu'on obtient une précision entre 81 et 85% dans tous les cas. Les résultats sont donc légèrement supérieurs que précédemment. **On obtient alors le meilleur résultat pour batch-size=5 et taux d'apprentissage égale à 0.001, avec un taux de précision de 85%.**

En utilisant ce résultat pour tester le meilleur modèle, on obtient les courbes suivantes :



Graphiques – (à gauche) La perte d'entropie-croisée en fonction de l'epoch, (à droite) la précision en fonction de l'epoch

On remarque que notre modèle apprend bien. En effet, à l'epoch 0, le modèle effectue la première propagation avec les paramètres aléatoires initiaux ce qui explique la perte élevée et la précision de 13%. Puis la modèle apprend en actualisant les paramètres par la méthode ADAM. On remarque bien que l'entropie croisée diminue autour de 0.370. La précision est en effet bien meilleure, avec une précision maximale de validation atteinte à l'epoch 10 avec 85.183%.

Voici le détail par epoch :

```
Epoch  0,                    Train: loss=2.303, accuracy=13.2%,                    Valid: loss=2.305, accuracy=13.2%
Epoch  1,                    Train: loss=0.430, accuracy=85.3%,                    Valid: loss=0.457, accuracy=84.0%
Epoch  2,                    Train: loss=0.451, accuracy=84.7%,                    Valid: loss=0.488, accuracy=83.3%
Epoch  3,                    Train: loss=0.397, accuracy=86.3%,                    Valid: loss=0.435, accuracy=85.2%
Epoch  4,                    Train: loss=0.408, accuracy=85.8%,                    Valid: loss=0.452, accuracy=84.5%
Epoch  5,                    Train: loss=0.410, accuracy=85.9%,                    Valid: loss=0.458, accuracy=84.0%
Epoch  6,                    Train: loss=0.394, accuracy=86.2%,                    Valid: loss=0.448, accuracy=84.3%
Epoch  7,                    Train: loss=0.409, accuracy=85.5%,                    Valid: loss=0.472, accuracy=83.2%
Epoch  8,                    Train: loss=0.381, accuracy=86.7%,                    Valid: loss=0.443, accuracy=84.6%
Epoch  9,                    Train: loss=0.380, accuracy=86.8%,                    Valid: loss=0.441, accuracy=84.8%
Epoch 10,                    Train: loss=0.370, accuracy=87.2%,                    Valid: loss=0.433, accuracy=85.2%
Best validation accuracy = 85.183
```

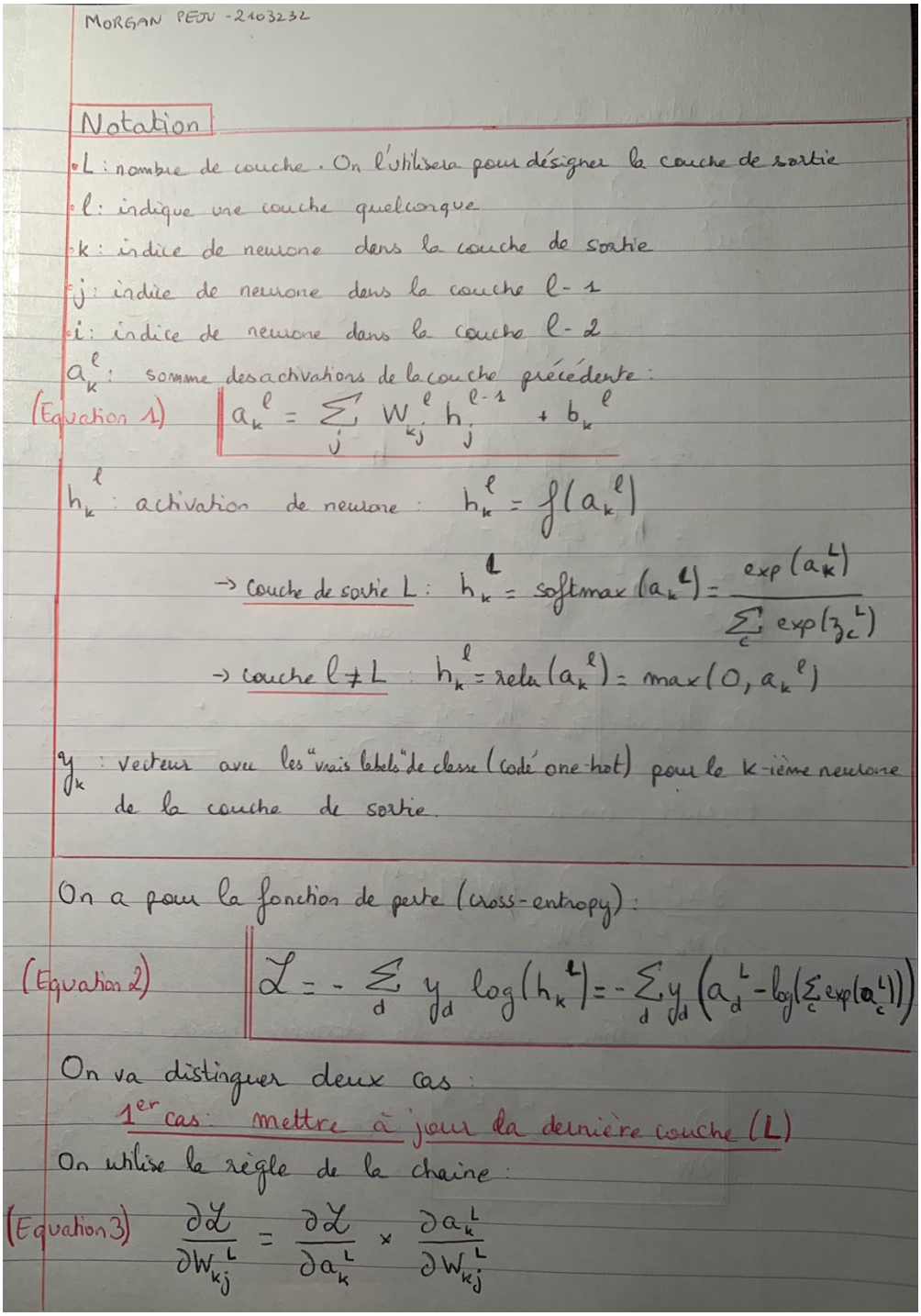
Ainsi, avec la méthode Adam, on obtient des résultats (en terme de précision) meilleurs qu'avec la méthode classique SGD. On peut remarquer également, que peu importe la méthode utilisée

ici, dès que l'algorithme commence à apprendre (epoch 1 avec la première actualisation des paramètres), la précision converge tout de suite vers la précision maximale.

3. Partie 3 (20 points) (Réalisée seul)

3.1. Calcul du gradient et pseudo-code

Dans cette partie, nous allons d'abord aborder le calcul théorique du gradient, avant de présenter le pseudo-code précisant les calculs matriciels.



MORGAN PEJU - 2103232

Notation

- L : nombre de couche. On l'utilisera pour désigner la couche de sortie
- l : indique une couche quelconque
- k : indice de neurone dans la couche de sortie
- j : indice de neurone dans la couche $l-1$
- i : indice de neurone dans la couche $l-2$

a_k^l : somme des activations de la couche précédente :

(Equation 1) $$a_k^l = \sum_j W_{kj}^l h_j^{l-1} + b_k^l$$

h_k^l : activation de neurone : $h_k^l = f(a_k^l)$

→ Couche de sortie L : $h_k^L = \text{softmax}(a_k^L) = \frac{\exp(a_k^L)}{\sum_c \exp(z_c^L)}$

→ couche $l \neq L$: $h_k^l = \text{relu}(a_k^l) = \max(0, a_k^l)$

y_k : vecteur avec les "vrais labels" de classe (codé one-hot) pour le k-ième neurone de la couche de sortie.

On a pour la fonction de perte (cross-entropy) :

(Equation 2) $$\mathcal{L} = -\sum_d y_d \log(h_k^L) = -\sum_d y_d \left(a_d^L - \log\left(\sum_c \exp(a_c^L)\right)\right)$$

On va distinguer deux cas :

1er cas : mettre à jour la dernière couche (L)

On utilise la règle de la chaine :

(Equation 3) $$\frac{\partial \mathcal{L}}{\partial W_{kj}^L} = \frac{\partial \mathcal{L}}{\partial a_k^L} \times \frac{\partial a_k^L}{\partial W_{kj}^L}$$

MORGAN PEJU - 2103232

En utilisant l'équation (2), on a :

$$\frac{\partial \mathcal{L}}{\partial a_k^L} = \frac{\partial}{\partial a_k^L}\left[-\sum_d y_d \left(a_d^L - \log\left(\sum_c \exp(a_c^L)\right)\right)\right]$$

car $\frac{\partial}{\partial a_k^L}\log\left(\sum_c \exp(a_c^L)\right) = \frac{\frac{\partial}{\partial a_k^L}\left(\sum_c \exp(a_c^L)\right)}{\sum_c \exp(a_c^L)}$

$$= -\sum_d y_d \left(\mathbb{1}_{d=k} - \frac{\exp(a_k^L)}{\sum_c \exp(a_c^L)}\right)$$

$$= -\sum_d y_d \left(\mathbb{1}_{d=k} - h_k^L\right) = \sum_d y_d h_k^L - \sum_d y_d \mathbb{1}_{d=k}$$

$$= h_k^L \sum_d y_d - y_k$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_k^L} = h_k^L - y_k}$$

avec $\mathbb{1}_{d=k}$ la fonction identité : $\mathbb{1}_{d=k} = \begin{cases} 1 \text{ si } d=k \\ 0 \text{ sinon} \end{cases}$

On définit $\boxed{\delta_k^L = h_k^L - y_k = \frac{\partial \mathcal{L}}{\partial a_k^L}}$ sachant que : $h_k^L = \text{softmax}(a_k^L) = \hat{y}$

Ensuite, on a :

$$\frac{\partial a_k^L}{\partial W_{kj}^L} = \frac{\partial}{\partial W_{kj}^L}\left(W_{kj}^L h_j^{L-1}\right) = h_j^{L-1}$$

On a donc pour la dernière couche L :

$$\boxed{\frac{\partial \mathcal{L}}{\partial W_{kj}^L} = \delta_k^L h_j^{L-1}}$$

et pour le biais : $\frac{\partial \mathcal{L}}{\partial b_k^L} = \frac{\partial \mathcal{L}}{\partial a_k^L} \times \frac{\partial a_k^L}{\partial b_k^L} = \delta_k^L$

MORGAN PEJU - 2103232

2ème cas: mettre à jour la couche précédente $\ell-1$:

On utilise la règle de la chaine :

$$\frac{\partial \mathcal{L}}{\partial W_{ji}^{\ell-1}} = \frac{\partial \mathcal{L}}{\partial h_j^{\ell-1}} \times \frac{\partial h_j^{\ell-1}}{\partial a_j^{\ell-1}} \times \frac{\partial a_j^{\ell-1}}{\partial W_{ji}^{\ell-1}}$$

On a :

$$\frac{\partial \mathcal{L}}{\partial h_j^{\ell-1}} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^{\ell}} \times \frac{\partial a_k^{\ell}}{\partial h_j^{\ell-1}} = \sum_k \delta_k^{\ell} W_{kj}^{\ell}$$

$$\frac{\partial h_j^{\ell-1}}{\partial a_j^{\ell-1}} = f'(a_j^{\ell-1}) \quad \text{et} \quad \frac{\partial a_j^{\ell-1}}{\partial W_{ji}^{\ell-1}} = h_i^{\ell-2}$$

Finalement on a pour le couche $\ell-1$:

$$\boxed{\frac{\partial \mathcal{L}}{\partial W_{ji}^{\ell-1}} = h_i^{\ell-2} f'(a_j^{\ell-1}) \sum_k \delta_k^{\ell} W_{kj}^{\ell}}$$

On définit : $\boxed{\delta_j^{\ell-1} = \frac{\partial \mathcal{L}}{\partial h_j^{\ell-1}} \times \frac{\partial h_j^{\ell-1}}{\partial a_j^{\ell-1}} = f'(a_j^{\ell-1}) \sum_k \delta_k^{\ell} W_{kj}^{\ell}}$

On a alors : $\boxed{\frac{\partial \mathcal{L}}{\partial W_{ji}^{\ell-1}} = \delta_j^{\ell-1} h_i^{\ell-2}}$

- Pour le biais, on a : $\frac{\partial \mathcal{L}}{\partial b_j^{\ell-1}} = \frac{\partial \mathcal{L}}{\partial h_j^{\ell-1}} \times \frac{\partial h_j^{\ell-1}}{\partial a_j^{\ell-1}} \times \frac{\partial a_j^{\ell-1}}{\partial b_j^{\ell-1}} = \delta_j^{\ell-1}$

Grâce à ces calculs, on peut ensuite décrire le pseudo-code avec les calculs matriciels associés :

Algorithme de rétropopagation dans un réseau de neurones:

Notes :

- Le calcul du gradient a été détaillé ci-dessus, nous utiliserons la forme matricielle dans le pseudo-code qui suit.
- $élément^{(i)}$ désigne un élément de la couche i.
- élement.T désigne la transposée de l'élément

Inputs : lr (taux d'apprentissage) et (X,Y)
Initialisation : Initialiser les paramètres W^i avec i = 0 à L (L : nombre de couches). Il y a donc L+1 tenseurs de paramètres.

Pour chaque batch d'exemples **faire**:
Propagation

- $a^{(0)} = XW^{(0)}$
- Stocker X comme "activation initiale" $h0$ dans une liste h
- **Pour** le nombre de couches cachées **faire**:
  - Appliquer l'activation $h^{(l)} = \text{relu}(a^{(l-1)})$
  - Augmenter $h^{(l)}$ avec un 1 pour prendre en compte le biais de la couche dans le calcul du gradient (voir note à la fin du pseudocode)
  - Ajouter l'activation $h^{(l)}$ dans la liste h
  - Calculer $a^{(l)} = h^{(l)}W^{(l)}$

  fin Pour
- Appliquer softmax : y_pred = $\text{softmax}(a^{(L)})$

Rétropropagation

- **Pour** le nombre de couches **faire** :
  - **SI** couche = couche de sortie L :
    - Calculer $\delta^{(L)}$ = Y_pred - Y
  - **SINON** :
    - Calculer $\delta^{(l)} = (\delta^{(l+1)}W^{(l+1)}.\text{T})*\text{relu}'(a^{(l)})$ (voir note à la fin du pseudocode)
  - Stocker le $\delta^{(i)}$ calculé dans une liste a
  - Calculer le gradient pour la couche : $grad^{(l)} = h^{(l-1)}.T\ \delta^{(l)}$

  fin Pour
- *Mettre à jour les paramètres :*
- **Pour** le nombre de couches **faire** :
  - Mettre à jour : $W^{(l)} = W^{(l)}$ - lr*$grad^{(l)}$

  fin Pour

fin Pour

Note:
Pour le calcul : $\delta^{(l)} = (\delta^{(l+1)}W^{(l+1)}.T)*\text{relu}'(a^{(l)})$, il faut noter que, comme démontré dans la solution analytique présentée précédemment, il faut faire attention à ne pas prendre la dernière ligne de W. En effet, comme ici nous utilisons une forme matricielle où le biais est intégré à la

dernière ligne du tenseur de paramètres et que le gradient du biais ne dépend pas de cette ligne (voir démonstration précédemment), il faut la retirer pour calculer ce qu'on nomme ici delta δ. Dans le cas contraire, on ne respecterait ni les dimensions ni la formule du gradient. Le gradient du biais est en fait pris en compte lors de la multiplication qui suit, à savoir $grad^{(l)} = h^{(l-1)}.T\ \delta^{(l)}$, où h a été augmenté avec un 1 pour prendre en compte le biais de la couche. Cela est bien pris en compte dans le code qui suit.

3.2. Code à compléter et analyse des résultats

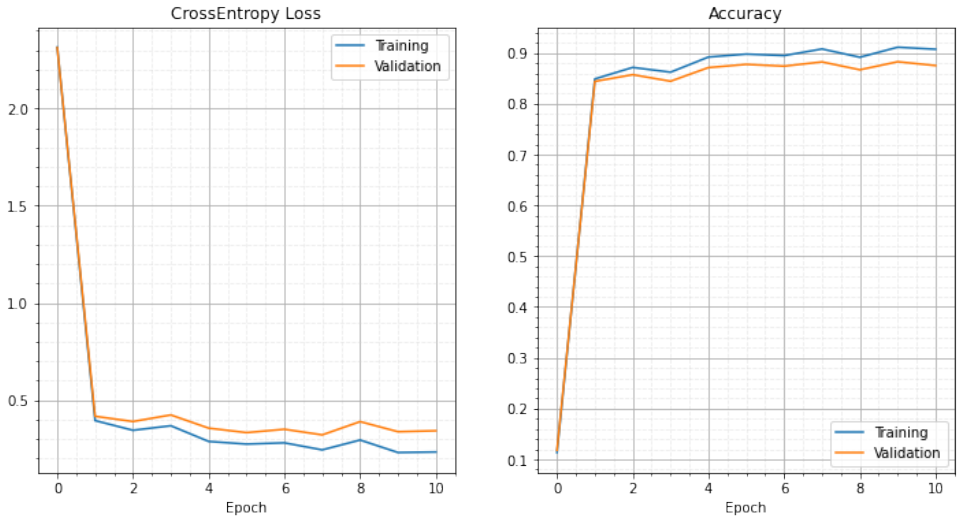
Le code de la partie 3 est à retrouver en annexe, via le notebook dans le .zip ou sur Colab : https://colab.research.google.com/drive/1BQ_LSz2wb39Yz7tlA6h_9_7w6wVPgGkd?usp=sharing

Afin de pouvoir reproduire les mêmes résultats à chaque fois, j'ai fixé un seed(2103232). De plus, étant donné le temps de calcul pour un batch petit, on prendra ici batch-size=20 et lr=0.01.

SGD :
Après l'implémentation du code, j'ai pu obtenir les résultats de précisions suivants en fonction de la profondeur et largeur du réseau de neurones :

depth\width	25	100	300	500	1000
1	85.750%	87.367%	87.367%	87.667%	87.467%
3	84.933%	86.500%	87.083%	**87.750%**	87.683%
5	83.267%	85.650%	87.083%	87.250%	87.683%

On peut voir qu'on obtient une précision entre 83 et 87% dans tous les cas. On remarque que plus le réseau est large (nombre de neurones par couche), meilleure est la précision. Toutefois, on remarque logiquement le temps de calcul augmente également.
On peut voir qu'avec 3 couches cachées et 500 neurones par couches, on obtient le meilleur résultat avec une précision de 87.750%.
En utilisant ce résultat pour tester le meilleur modèle, on obtient les courbes suivantes :



Graphiques – (à gauche) La perte d'entropie-croisée en fonction de l'epoch, (à droite) la précision en fonction de l'epoch

On remarque que notre modèle apprend bien. En effet, à l'epoch 0, le modèle effectue la première propagation avec les paramètres aléatoires initiaux ce qui explique la perte élevée et la précision de 10%. Puis la modèle apprend en actualisant les paramètres par la minimisation de l'entropie croisée. On remarque bien que l'entropie croisée diminue sous 0.5. La précision est en effet bien meilleure, avec une précision maximale de validation atteinte à l'epoch 9 avec 88.267%.
Voici le détail par epoch :

```
Epoch  0,                    Train:loss=2.316, accuracy=11.4%,                    Valid: loss=2.313, accuracy=11.8%
Epoch  1,                    Train:loss=0.394, accuracy=84.9%,                    Valid: loss=0.416, accuracy=84.4%
Epoch  2,                    Train:loss=0.344, accuracy=87.1%,                    Valid: loss=0.390, accuracy=85.7%
Epoch  3,                    Train:loss=0.368, accuracy=86.2%,                    Valid: loss=0.423, accuracy=84.4%
Epoch  4,                    Train:loss=0.287, accuracy=89.2%,                    Valid: loss=0.355, accuracy=87.1%
Epoch  5,                    Train:loss=0.273, accuracy=89.8%,                    Valid: loss=0.332, accuracy=87.7%
Epoch  6,                    Train:loss=0.280, accuracy=89.5%,                    Valid: loss=0.350, accuracy=87.4%
Epoch  7,                    Train:loss=0.244, accuracy=90.8%,                    Valid: loss=0.321, accuracy=88.2%
Epoch  8,                    Train:loss=0.294, accuracy=89.2%,                    Valid: loss=0.389, accuracy=86.7%
Epoch  9,                    Train:loss=0.230, accuracy=91.1%,                    Valid: loss=0.337, accuracy=88.3%
Epoch 10,                    Train:loss=0.233, accuracy=90.7%,                    Valid: loss=0.342, accuracy=87.5%
Best validation accuracy = 88.267
```

ADAM :

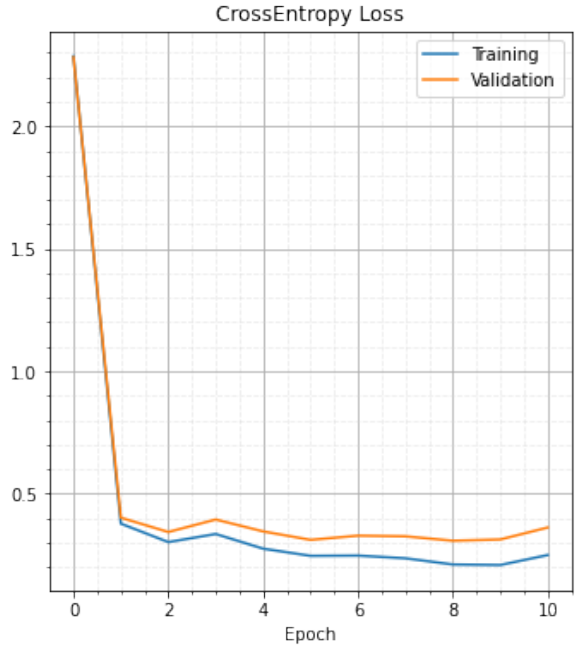
Cette fois-ci en utilisant ADAM pour l'actualisation, j'ai pu obtenir les résultats de précisions suivants en fonction de la profondeur et largeur du réseau de neurones :

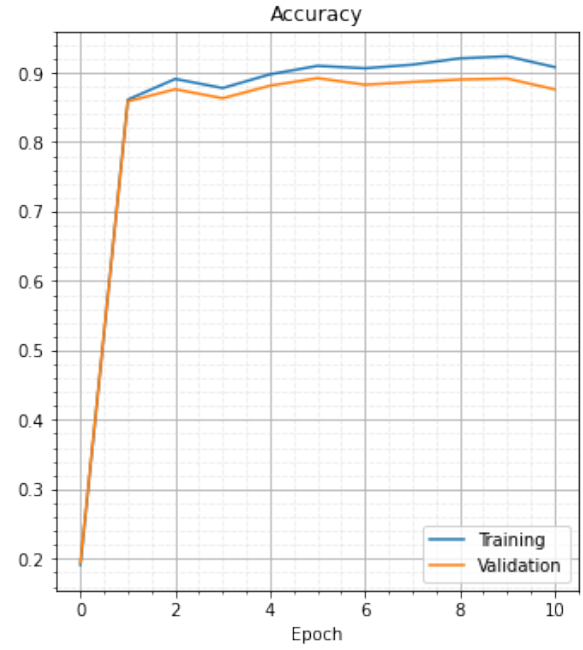
depth\width	25	100	300	500	1000
1	86.633%	87.567%	88.550%	88.750%	**89.200%**
3	86.950%	87.633%	88.217%	88.150%	88.583%
5	86.233%	87.267%	87.900%	88.267%	87.867%

On peut voir qu'on obtient une précision entre 86 et 89% dans tous les cas. On remarque une nouvelle fois que plus le réseau est large (nombre de neurones par couche), meilleure est la précision.

On peut voir qu'avec 1 couche cachée et 1000 neurones par couches, on obtient le meilleur résultat avec une précision de 89.200%.

En utilisant ce résultat pour tester le meilleur modèle, on obtient les courbes suivantes :





Graphiques – (à gauche) La perte d'entropie-croisée en fonction de l'epoch, (à droite) la précision en fonction de l'epoch

On remarque que notre modèle apprend bien. En effet, à l'epoch 0, le modèle effectue la première propagation avec les paramètres aléatoires initiaux ce qui explique la perte élevée et la précision de 19%. Puis la modèle apprend en actualisant les paramètres par la méthode ADAM. On remarque bien que l'entropie croisée diminue sous 0.5. La précision est en effet bien meilleure, avec une précision maximale de validation atteinte à l'epoch 5 avec 89.2%.
Voici le détail par epoch :

```
Epoch  0,                  Train:loss=2.286, accuracy=19.1%,                Valid: loss=2.282, accuracy=19.4%
Epoch  1,                  Train:loss=0.378, accuracy=86.1%,                Valid: loss=0.402, accuracy=85.9%
Epoch  2,                  Train:loss=0.302, accuracy=89.1%,                Valid: loss=0.344, accuracy=87.6%
Epoch  3,                  Train:loss=0.335, accuracy=87.8%,                Valid: loss=0.395, accuracy=86.3%
Epoch  4,                  Train:loss=0.276, accuracy=89.8%,                Valid: loss=0.346, accuracy=88.1%
Epoch  5,                  Train:loss=0.246, accuracy=91.0%,                Valid: loss=0.312, accuracy=89.2%
Epoch  6,                  Train:loss=0.247, accuracy=90.6%,                Valid: loss=0.329, accuracy=88.2%
Epoch  7,                  Train:loss=0.236, accuracy=91.1%,                Valid: loss=0.326, accuracy=88.7%
Epoch  8,                  Train:loss=0.211, accuracy=92.0%,                Valid: loss=0.308, accuracy=89.0%
Epoch  9,                  Train:loss=0.209, accuracy=92.3%,                Valid: loss=0.313, accuracy=89.1%
Epoch 10,                  Train:loss=0.250, accuracy=90.8%,                Valid: loss=0.362, accuracy=87.6%
Best validation accuracy = 89.200
```

Ainsi, avec la méthode Adam, on obtient des résultats (en terme de précision) meilleurs qu'avec la méthode classique SGD. En effet, avec ADAM on gagne 2 à 3% de précision. On peut remarquer également, que peu importe la méthode utilisée ici, dès que l'algorithme commence à apprendre (epoch 1 avec la première actualisation des paramètres), la précision converge tout de suite vers la précision maximale.

Enfin, si on compare à la partie 2, on notera qu'en ajoutant des couches cachées avec des neurones, la précision augmente passant des 80% de précision à ~87% . Finalement, on aura mis en évidence le performance de la régression logistique et des réseaux de neurones pour un problème de classification.

4. ANNEXES

4.1. Code Partie 1

CODE PYTHON

```python
import numpy as np

# Les tableaux sont bâtis avec les dimensions (N, P, A, W, H)
# et chaque dimension est (False, True)

Pr_N = np.array([0.8, 0.2]).reshape(2, 1, 1, 1, 1)
Pr_P_given_N = np.array([[0.9,0.1],[0.6,0.4]]).reshape(2, 2, 1, 1, 1) # TODO
Pr_A_given_N = np.array([[0.7,0.3],[0.9,0.1]]).reshape(2, 1, 2, 1, 1) # TODO
Pr_W_given_P = np.array([[0.8, 0.2],[0,1]]).reshape(1, 2, 1, 2, 1) # TODO
Pr_H_given_PA = np.array([[1,0],[0.1,0.9],[0,1],[0,1]]).reshape(1, 2, 2, 1, 2) # TODO

print (f"Pr(N)=\n{np.squeeze(Pr_N)}\n")
print (f"Pr(P|N)=\n{np.squeeze(Pr_P_given_N)}\n")
print (f"Pr(A|N)=\n{np.squeeze(Pr_A_given_N)}\n")
print (f"Pr(W|P)=\n{np.squeeze(Pr_W_given_P)}\n")
print (f"Pr(H|P,A)=\n{np.squeeze(Pr_H_given_PA)}\n")


# Question a
answer = np.sum(Pr_H_given_PA*Pr_W_given_P*Pr_P_given_N*Pr_A_given_N*Pr_N, axis=(0,1,2,3), keepdims=True)[0,0,0,0,1]
print(f"Pr(H=1)={answer:.5f}")

# Question b
Pr_A = (Pr_N*Pr_A_given_N).sum(axis=0)
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,1,3), keepdims=True)/Pr_A)[0,0,1,0,1]
print(f"Pr(H=1|A=1)={answer:.5f}")

# Question c
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,1,3), keepdims=True))[0,0,1,0,1]
print(f"Pr(H=1|do(A=1))={answer:.5f}")

# Question d
Pr_P = (Pr_N*Pr_P_given_N).sum(axis=0)
Pr_W=np.sum(Pr_W_given_P*Pr_P, axis=(1), keepdims=True)
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,1,2), keepdims=True)/Pr_W)[0,0,0,1,1]
print(f"Pr(H=1|W=1)={answer:.5f}")

# Question e
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_H_given_PA, axis=(0,1,2), keepdims=True))[0,0,0,0,1]
print(f"Pr(H=1|do(W=1))={answer:.5f}")

# Question f
Pr_P = (Pr_N*Pr_P_given_N).sum(axis=0)
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,2,4), keepdims=True)/Pr_P)[0,1,0,1,0]
print(f"Pr(W=1|P=1)={answer:.5f}")

# Question g
answer = np.sum(Pr_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,2,4), keepdims=True)[0,1,0,1,0]
print(f"Pr(W=1|do(P=1))={answer:.5f}")

# Question h
Pr_P = (Pr_N*Pr_P_given_N).sum(axis=0)
answer = (np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,2,3), keepdims=True)/Pr_P)[0,1,0,0,1]
print(f"Pr(H=1|P=1)={answer:.5f}")

# Question i
answer = (np.sum(Pr_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(0,2,3), keepdims=True))[0,1,0,0,1]
print(f"Pr(H=1|do(P=1))={answer:.5f}")

# Question j
numerateur = np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(2), keepdims=True)[1,1,0,1,1]
denominateur = np.sum(Pr_N*Pr_P_given_N*Pr_A_given_N*Pr_W_given_P*Pr_H_given_PA, axis=(1,2), keepdims=True)[1,0,0,1,1]

answer = numerateur/denominateur

print(f"Pr(P=1|W=1,H=1,N=1)={answer:.5f}")
```

4.2. Code Partie 2

CODE PYTHON

```python
# fonctions pour charger les ensembles de donnees
from torchvision.datasets import FashionMNIST
from torchvision import transforms
import torch
```

```python
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):
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  dataset = FashionMNIST("./dataset", train=True,  download=True, transform=transforms.Compose([transforms.ToTensor()]))
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  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
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  dataset_train, dataset_val = random_split(dataset, [len_train, len_val])
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  data_loader_train = DataLoader(dataset_train, batch_size=batch_size,shuffle=True,num_workers=4)
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  data_loader_val   = DataLoader(dataset_val, batch_size=batch_size,shuffle=True,num_workers=4)
  torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
  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()




from gc import DEBUG_UNCOLLECTABLE
from logging import logProcesses

def accuracy(y, y_pred) :
    # todo : nombre d'éléments à classifier.
    card_D = y.shape[0]
    # todo : calcul du nombre d'éléments bien classifiés.
    ind_pred = torch.argmax(y_pred, dim=1)
    ind_target = torch.argmax(y, dim=1)
    card_C = (ind_pred == ind_target).sum()
    # todo : calcul de la précision de classification.
    acc = np.abs(card_C)/np.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):
    # todo : calcul de la valeur d'entropie croisée.
    epsilon = 1e-8  # Pour éviter le cas log(0)
    loss = -(y*torch.log(y_pred + epsilon)).sum()
    return loss

def softmax(x, axis=-1):
    # assurez vous que la fonction est numeriquement stable
    # e.g. softmax(np.array([1000, 10000, 100000], ndim=2))

    # todo : calcul des valeurs de softmax(x)
    x = x - torch.max(x, axis=1, keepdims=True)[0]   # Pour que softmax soit numériquement stable et sans modifier le
résultat
```

```
    values = torch.exp(x)/torch.exp(x).sum(axis=1, keepdims=True)

    return  values

def inputs_tilde(x, axis=-1):
    # augments the inputs `x` with ones along `axis`
    # todo : implémenter code ici.
    X1 = torch.ones([len(x),1])
    x_tilde = torch.cat([x,X1], axis=axis)
    return x_tilde




class LinearModel:
    def __init__(self, num_features, num_classes):
      torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
      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):
      # todo : implémenter calcul des outputs en fonction des inputs `x`.
      inputs = inputs_tilde(x)
      outputs = softmax(torch.matmul(inputs,self.params))
      return outputs

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

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

      pass

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

      self.t += 1

      self.m_t = beta_1*self.m_t + (1 - beta_1)*grads
      m_t_corrected = self.m_t/(1 - beta_1**self.t)

      self.v_t = beta_2*self.v_t + (1 - beta_2)*(grads**2)
      v_t_corrected = self.v_t/(1 - beta_2**self.t)

      self.params = self.params - lr*(m_t_corrected/(torch.sqrt(v_t_corrected)+epsilon))
      pass

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

    best_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_accuracy:
```

```
            #
            best_model = model
            best_val_accuracy, best_accuracy = accuracy_val, accuracy_val
            #
            pass   # TODO : record the best model parameters and best validation accuracy

        logger.log(accuracy_train, loss_train, accuracy_val, loss_val)
        #if epoch % 5 == 1: # prints every 5 epochs, you can change it to % 1 for example to print each epoch
        if epoch >=0: # prints every 5 epochs, you can change it to % 1 for example to print each epoch
          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




########################################
## SGD - RECHERCHE DES HYPERPARAMETRES  ##
########################################
# 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 = [5,20,200,1000]   # Define ranges in a list
lr_list = [0.1,0.01,0.001]          # Define ranges in a list

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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}")

########################################
##         SGD - MEILLEUR MODELE        ##
########################################
# SGD
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
batch_size = 5 # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.01         # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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=10, 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}")


########################################
## ADAM - RECHERCHE DES HYPERPARAMETRES  ##
########################################
# 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 = [5,20,200,1000]   # Define ranges in a list
lr_list = [0.1,0.01,0.001]           # Define ranges in a list

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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}")


########################################
##        ADAM - MEILLEUR MODELE      ##
########################################
# ADAM
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
batch_size = 5 # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001         # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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=10, 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}")
```

4.3. Code Partie 3

CODE PYTHON

```
''' Les fonctions dans cette cellule peuvent avoir les mêmes déclarations que celles de la partie 2'''
def accuracy(y, y_pred) :
    # todo : nombre d'éléments à classifier.
    card_D = y.shape[0]
    # todo : calcul du nombre d'éléments bien classifiés.
    ind_pred = torch.argmax(y_pred, dim=1)
    ind_target = torch.argmax(y, dim=1)
    card_C = (ind_pred == ind_target).sum()
    # todo : calcul de la précision de classification.
    acc = np.abs(card_C)/np.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`
    # todo : implémenter code ici.
    X1 = torch.ones([len(x),1])
    x_tilde = torch.cat([x,X1], axis=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))
```

```
    # todo : calcul des valeurs de softmax(x)
    x = x - torch.max(x, axis=1, keepdims=True)[0]   # Pour que softmax soit numériquement stable et sans modifier le
résultat
    values = torch.exp(x)/torch.exp(x).sum(axis=1, keepdims=True)
    return  values

def cross_entropy(y, y_pred):
    # todo : calcul de la valeur d'entropie croisée.
    epsilon = 1e-8  # Pour éviter le cas log(0)
    loss = -(y*torch.log(y_pred + epsilon)).sum()
    return loss

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

def relu_forward(x):
    # todo : calcul des valeurs de relu(x)
    values = torch.max(x,torch.tensor([0.]))
    return values

def relu_backward(x):
    # todo : calcul des valeurs du gradient de la fonction `relu`
    values = (x > 0) * 1 # On met à 1 lorsque x > 0, 0 sinon.
    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

        # todo : initialiser la liste des paramètres Teta de l'estimateur.
        # On initialise les poids avec la méthode 'Xavier'

        # Paramètres d'inputs
        limit_input = np.sqrt(6 / float((n_features + 1) + n_hidden_features))
        np.random.seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
        self.params_input = [torch.from_numpy(np.random.uniform(low=-limit_input, high=limit_input, size=(n_features + 1,
n_hidden_features))).float()]
        # Paramètres d'output
        limit_output = np.sqrt(6 / float(n_hidden_features + n_classes))
        np.random.seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
        self.params_output = [torch.from_numpy(np.random.uniform(low=-limit_output, high=limit_output,
size=(n_hidden_features + 1, n_classes))).float()]
        #Paramètres des couches cachées
        if n_hidden_layers > 1:
          # Ici, il faut prendre également en compte, comme le montre le schéma de l'énoncé, qu'on a un biais à chaque
couche
          limit_hidden_layer = np.sqrt(6 / float(n_hidden_features * 2))
          np.random.seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
          self.params_hidden_layer = [torch.from_numpy(np.random.uniform(low=-limit_hidden_layer,
high=limit_hidden_layer, size=(n_hidden_features+1, n_hidden_features))).float() for x in range(1,n_hidden_layers)]
          self.params = self.params_input + self.params_hidden_layer + self.params_output
        else :
          self.params = self.params_input + self.params_output
        print(f"Teta params={[p.shape for p in self.params]}")

        self.a = [] # liste contenant le resultat des multiplications matricielles
        self.h = [] # liste contenant le resultat des fonctions d'activations

        self.t = 0
        self.m_t = [0 for i in range(0,len(self.params))]
        self.v_t = [0 for i in range(0,len(self.params))]

    def forward(self, x):
        # todo : implémenter calcul des outputs en fonction des inputs `x`.
        self.h = [] # Réinitialisation
        inputs = inputs_tilde(x)
        z_x = inputs @ self.params[0]
        self.h.append(inputs)

        # On applique ensuite les activations relu de chaque couche
        for layer in range(1, self.n_hidden_layers+1):
          h_x = relu_forward(z_x)
          h_x = inputs_tilde(h_x) # On augmente h avec un 1 pour prendre en compte le biais de la couche
          self.h.append(h_x)    # On conserve les activations pour les réutiliser dans la descente de gradient
          z_x = h_x @ self.params[layer]

        # On applique la dernière activation (softmax) pour obtenir les probabilités de prédiction
```

```
        outputs = softmax(z_x)
        return outputs

    def backward(self, y, y_pred):
        # todo : implémenter calcul des gradients.
        layers = np.arange(0,self.n_hidden_layers+1,1).tolist()
        grads = []

        # Ici on va traiter 2 cas: le cas où il s'agit de la dernière couche L, le cas où il s'agit d'un couche cachée
        for layer in reversed(layers):
          h = self.h.pop()
          if layer == self.n_hidden_layers : # Couche de sortie (softmax)
            delta = softmax_cross_entropy_backward(y, y_pred)
          else: # Couches cachées
            a = self.a.pop()
            # Pour le calcul ci-dessous, comme démontré dans la solution analytique présentée précédemment, il faut faire attention
            # à ne pas prendre la dernière ligne de self.params. En effet, comme ici nous utilisons une forme matricielle où
            # le biais est intégré à la dernière ligne du tenseur de paramètres et que le gradient du biais ne dépend pas de cette ligne
            # (voir démonstration précédemment), il faut la retirer pour calculer ce qu'on nomme ici delta. Dans le cas contraire,
            # on ne respecterait ni les dimensions ni la formule du gradient.
            delta = (a @ self.params[layer+1][:-1,:].T)*relu_backward(h @ self.params[layer])

          grad = h.T @ delta # Calcul du gradient

          # On stocke delta et le gradient de la couche
          self.a.append(delta)
          grads.insert(0, grad)

        return grads

    def sgd_update(self, lr, grads):
      # TODO : implémenter mise à jour des paramètres ici.
      for layer in range (0, len(self.params)):
        self.params[layer] = self.params[layer] - lr * grads[layer]
      pass

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

      self.t += 1

      for layer in range (0, len(self.params)):
        self.m_t[layer] = beta_1*self.m_t[layer] + (1 - beta_1)*grads[layer]
        m_t_corrected = self.m_t[layer]/(1 - beta_1**self.t)

        self.v_t[layer] = beta_2*self.v_t[layer] + (1 - beta_2)*(grads[layer]**2)
        v_t_corrected = self.v_t[layer]/(1 - beta_2**self.t)

        self.params[layer] = self.params[layer] - lr*(m_t_corrected/(torch.sqrt(v_t_corrected)+epsilon))
      pass

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:
          best_val_accuracy = accuracy_val
          best_model = model
          pass   # TODO : record the best model parameters and best validation accuracy
```

```
        logger.log(accuracy_train, loss_train, accuracy_val, loss_val)
        #if epoch % 5 == 0: # prints every 5 epochs, you can change it to % 1 for example to print each epoch
        #if epoch % 5 == 1: # prints every 5 epochs, you can change it to % 1 for example to print each epoch
        if epoch >=0: # prints every 5 epochs, you can change it to % 1 for example to print each epoch
          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



########################################
## SGD - RECHERCHE DES HYPERPARAMETRES  ##
########################################
# 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 = 0.01           # Some value
# Etant donné le temps de calcul pour un batch petit, on prendra ici batch-size = 20
batch_size = 20   # Some value

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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}")


########################################
##         SGD - MEILLEUR MODELE        ##
########################################

# SGD
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
depth = 3   # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
width = 500    # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.01           # Some value
# Etant donné le temps de calcul pour un batch petit, on prendra ici batch-size = 20
batch_size = 20   # Some value

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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=10, 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}")

########################################
## ADAM - RECHERCHE DES HYPERPARAMETRES ##
########################################
# 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
# Etant donné le temps de calcul pour un batch petit, on prendra ici batch-size = 20
batch_size = 20   # Some value

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)
```

```
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}")


########################################
##        ADAM - MEILLEUR MODELE      ##
########################################


# ADAM
# Montrez les résultats pour la meilleure configuration trouvez ci-dessus.
depth = 1    # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
width = 1000    # TODO: Vous devez modifier cette valeur avec la meilleur que vous avez eu.
lr = 0.001          # Some value
batch_size = 20   # Some value

torch.manual_seed(2103232) # Seed pour la reproductibilité des résultats seed(matricule étudiant)

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=10, 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}")
```