

**INF8225**

**Hiver 2020**

**TP No. 2**

**1878557 – Bernard Meunier**

**Soumis à :** Christopher Pal

Partie 1 :

A)

Avec :

* Z[] : liste de matrices de valeur préactivation. Une matrice par couche de neurones Dimension de chaque matrice : *nombre d’exemples* *N* \* *neurones par couches*
* a[] : liste de matrices de valeur activées avec un biais. Une matrice par couche de neurones Dimension de chaque matrice : *nombre d’exemples N* \* *neurones par couches + 1*.
* deltas[] : liste contenant les matrices de deltas de chaque couche :

Dimension de chaque matrice : *nombre d’exemples N* \* *neurones par couches*.

* W[] : liste contenant les matrices de poids entre les couches de neurones :

Dimension de chaque matrice : *neurone dans la couche précédente + 1* \* *neurones dans la couche suivante.*

* f() : la fonction d’activation de la couche de sortie
* f ’() : la fonction dérivée de la fonction d’activation de sortie
* h() : la fonction d’activation des couches cachées.
* h’() : la fonction dérivée de la fonction d’activation des couches cachées.
* X : la matrice d’entrée des exemples d’entrainement.

Dimension : *nombre d’exemples N \* nombre de noeud d’entrées*

* Y : La matrice one-hot des classes des exemples d’entrainement.

Dimension: *nombre d’exemples* *N* \* *nombre de classe*.

* lr : constante du taux d’apprentissage
* @ : opération de multiplication de matrice.
* AddBias(M) : Fonction qui ajoute une colonne de 1 biais a la matrice M.
* cut(M) : fonction qui retire la colonne correspondante au biais dans la matrice M.
* L : le nombre de couches cachées
* N : le nombre d’exemples d’entrainement.

**PSEUDOCODE**

Forward propagation :

1. A0 <- AddBias(X)
2. Pour i de 0 à Lexclu faire :
   1. Zi+1 <- ai @ Wi
   2. Ai+1 <- AddBias(h(Zi+1))
3. ZL+1 <- aL @ WL
4. AL+1 <- f(ZL+1)

Backpropagation :

1. deltasL+1<- (Y – aL+1) \* f ’(ZL+1)
2. Pour chaque i de L à 0exclu, faire:
   1. deltasi <- cut(deltasi+1 @ W­i+1tranposé) \* h’(Zi+1)
3. Pour chaque i de 0 a L+1exclu:
   1. gradient <- ( aitransposé@ deltasi+1) / N
   2. Wi <- Wi + lr \* gradient

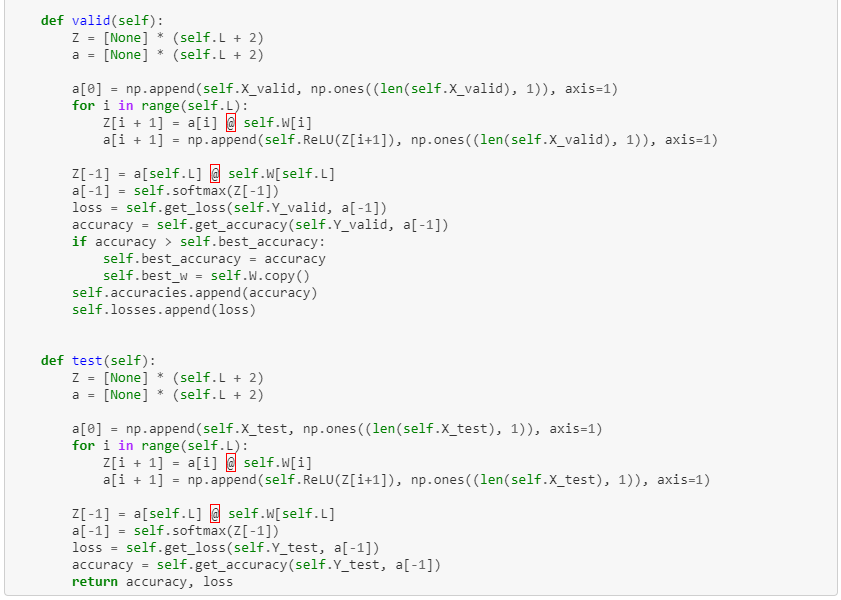
B)

Implémentation du réseau neuronal multicouche.

Classe neural network :





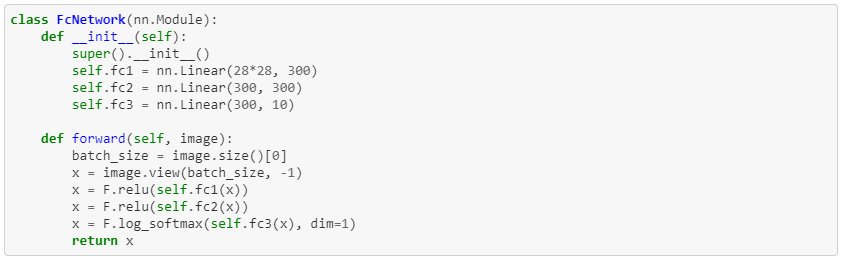


Implémentation pytorch :

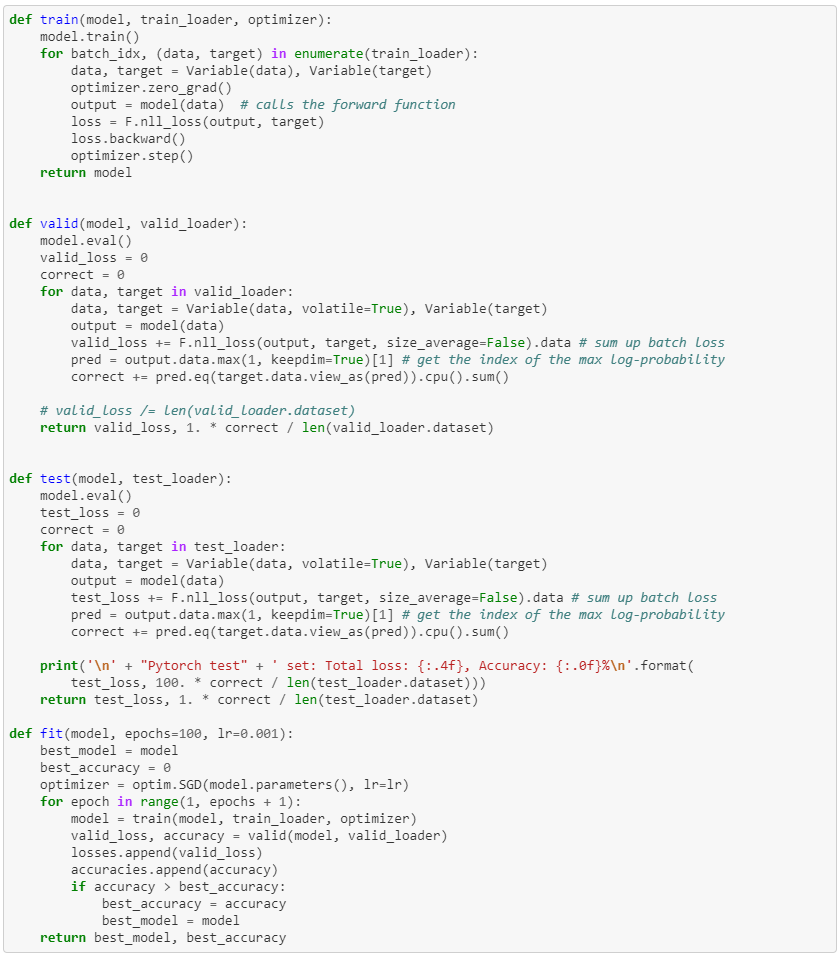
Importation des données



Classe FcNetwork



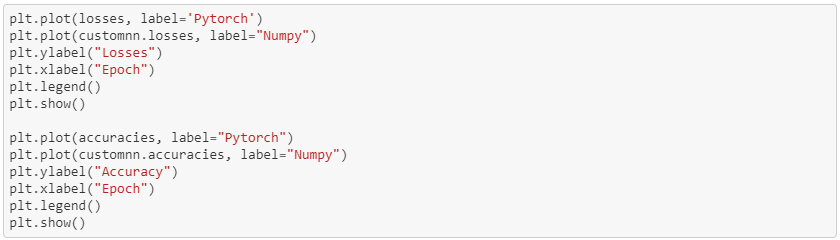
Méthode d’entrainement:

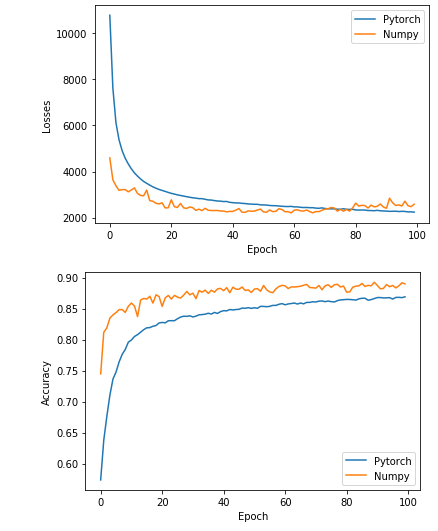


Code d’exécution :

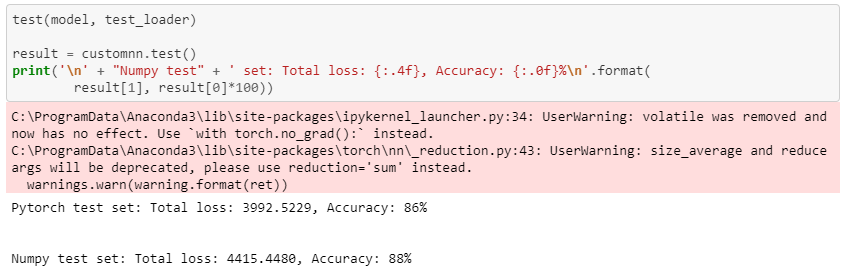


Graphique





Test



Conclusion :

Les deux implémentations de réseau neuronal permettent d’obtenir un résultat satisfaisant et fonctionnel.

On y retrouve quand même quelque différence entre les deux implémentations. Pour le scénario testé, nous avons utilisé des minibatchs de 1000 exemples et un taux d’apprentissage de 0.001. Les deux implémentations utilisent un optimisateur à descente de gradient.

En regardant le graphique de losses par epoch, on remarque que l’implémentation pytorch semble être beaucoup plus stable que celle avec numpy. Cette dernière donne des résultats beaucoup moins prévisibles.

Cependant, même si l’implémentation avec numpy semble beaucoup plus chaotique que celle de pytorch, elle semble procurer de meilleure performance en termes de précision autant avec les données de validation qu’avec les données de test. La précision obtenue avec l’implémentation numpy avec les données de test est de 88% alors que celle avec pytorch est de 86%.

On peut aussi voir que l’implémentation avec numpy semble atteindre sa meilleure performance avec seulement 50 epoch alors que l’implémentation pytorch ne semble pas avoir plafonné après 100 epoch, mais sont taux d’amélioration est très faible. Cependant, cela met à risque d’overfitting beaucoup plus facilement l’algorithme de numpy.

De plus, lors de l’exécution des deux algorithmes, on remarque que celle utilisant numpy est significativement plus rapide que celle de pytorch, ce qui est plus pratique d’utilisation.

Finalement, pour l’implémentation elle-même des algorithmes, il est évidant qu’il est beaucoup plus facile et rapide à implémenté le réseau de neurones en utilisant pytorch, une fois qu’on est en connaissance de la librairie. Du côté de numpy, l’implémentation est beaucoup plus à risque de bogue et de mauvaise optimisation, mais permet un contrôle et observation totale du fonctionnement du réseau.

Voici un tableau qui résume les pour et les contres de chacune des implémentations :

|  |  |  |
| --- | --- | --- |
|  | Numpy | Pytorch |
| Précision | Meilleure précision | Moins bonne précision |
| Apprentissage | Chaotique | Prédictible |
| Temps d’entrainement | Plus rapide | Moins rapide |
| Overfitting | Plus à risque | Peu à risque |
| Rapidité d’exécution | Plus rapide | Moins rapide |
| Facilité d’implémentation | Difficile | Facile |