18/02/2019

Valentin Bouis

1733927

TP2 INF8225

Backpropagation et introduction à Pytorch avec fashion MNIST

**Partie 1**

a) Pour l’algorithme de la back propagation, je me base principalement sur celui fourni dans le livre (page 777).

Input : L’ensemble d’entrainement (vecteur d’entrées et la sortie attendue)

**Initialisation :**

Matrice W <- Valeur aléatoire

Pour chaque ensemble (x,y) dans l’ensemble d’entrainement de 1 à k

**Étape de prédiction :**

Pour chaque neurone i de la couche d’input

a[i] <- x[i]

Pour chaque couche cachée l de 2 à L

Somme <- W[l-1] • a[l-1]

a[l] <- f(Somme)

**Étape de back propagation :**

Δ[L] <- f’(Somme) (y – a[L])

Pour chaque couche cachée l de L -1 à 1

Δ[l] <- f’(Somme) W[l+1] • Δ[l+1]

**Mise à jour des poids avec les gradients :**

W <- W + a x lr x Δ

Quelques explications :

Dans un réseau de neurones en back propagation, la procédure est telle que : On utilise les paramètres actuels pour prédire une sortie; Selon l’erreur obtenue (prédiction par rapport à la valeur attendue) on calcule le gradient sur la couche de la sortie. On propage ce gradient en reculant, c’est-à-dire en partant de la dernière couche jusqu’à la couche d’entrées. On met à jour les poids w (qui comprend le biais dans notre cas) avec le gradient (et le learning rate).

A noter que pendant la back propagation, c’est la dérivée de la fonction d’activation f qui est utilsée. Dans notre cas elle est telle que :

La deuxième remarque est que on multiplie cette dérivée par (y – a) et non par (a – y). Cela est due au fait qu’on parcourt le réseau en reculant, ce qui implique une négation.

*En revanche un facteur 2 a disparu dans l’algorithme du livre : Lorsqu’on dérive la fonction de perte (a – y)2 par rapport à a on obtient 2(a – y) or le 2 n’apparait pas dans la suite de l’algorithme.*

La dernière remarque que je ferai est bien que lorsqu’on parcourt le réseau dans le sens normal (prédiction) on utilise la couche précédente pour calculer la couche suivante (l-1) alors que lorsqu’on le parcourt en arrière on utilise la couche suivante (l+1).

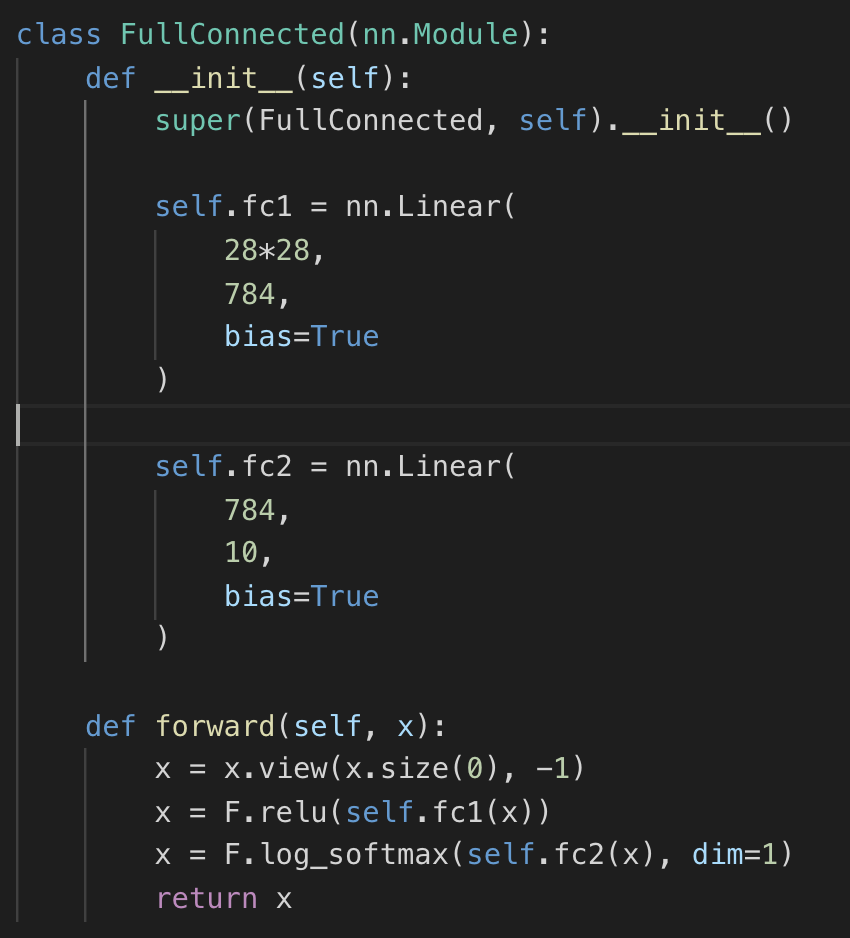
b) Lorsque l’ensemble d’entrainement est très grand, il existe plusieurs techniques afin d’accélérer le phase d’entrainement tout en gardant la back propagation pour mettre à jour les paramètres de notre modèle. La plus connue est la division du set d’entrainement en plusieurs sous-ensembles appelés mini-batchs. La principale différence est qu’au lieu de mettre à jour nos paramètres à chaque exemple d’entrainement on le fait à chaque mini-batch. La taille de ces derniers peut varier, pour un ensemble d’entrainement de taille 500000, on peut imaginer des mini-batch de taille 200 par exemple. Lorsqu’on utilise cette technique, il faut alors accumuler la fonction de perte pour chaque exemple du mini-batch afin de calculer le gradient dessus seulement à la fin du mini-batch (ne pas oublier de normaliser cette perte par la taille du mini-batch).

**Partie 2 (Faite avec : Wassim Guellati 17…….)**

Dans cette partie il s’agit d’implémenter un réseau de neurones capable d’identifier des vêtements provenant de la fashion MNIST. L’intérêt est d’appréhender la librairie Pytorch et d’expérimenter différents modèles afin de voir lequel fonctionne le mieux pour ce genre de problème. De manière à mieux cerner l’influence des couches de notre modèle, nous garderons les paramètres suivants identiques : nombre d’époch, batch size, learning rate, fonction de perte. Le code principal main.py se base sur le code github de AlexPiche.

**Modèle 1**

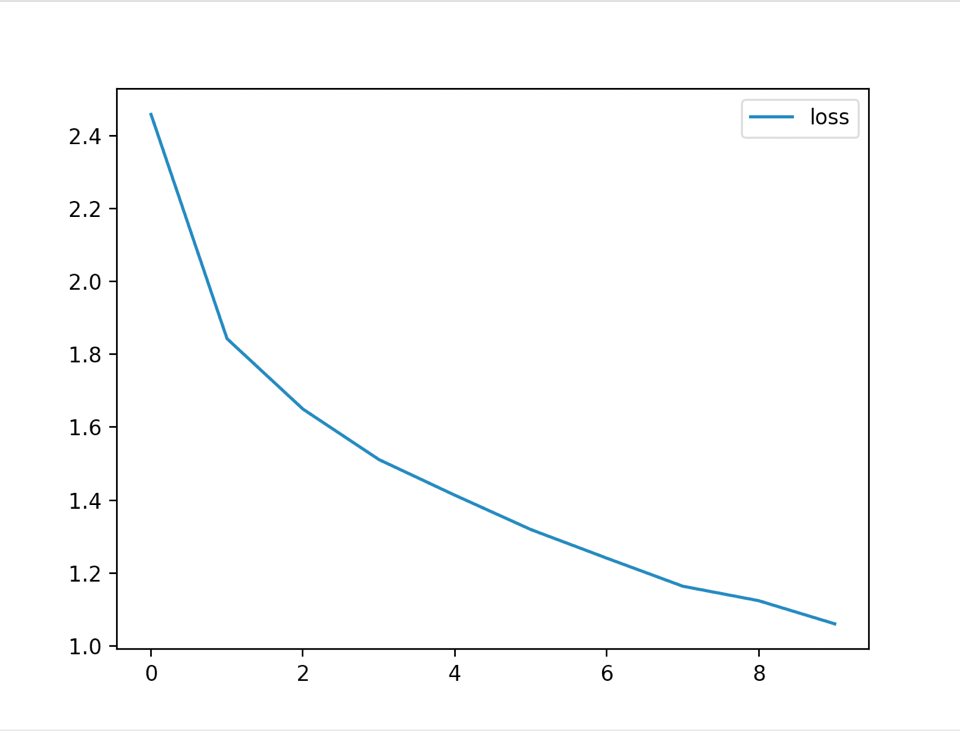
Le premier modèle est un réseau full connected avec 2 hidden layers, tel que dans le tutorial pytorch donné dans le github de AlexPiche. Il est constitué de :



À noter la fonction relu() utilisée lors de la première couche afin d’éliminer des potentiels nombres négatifs et la fonction softmax() pour avoir une répartition probabilistes sur la sortie.

Sur le set de validation on trouve une précision de 8828/10000 (**88%**) et une perte moyenne de **0.3657**.

On obtient la fonction de perte suivante :



**Modèle 2**

Le deuxième modèle est tiré d’un tutorial du site mc.ai. Il consiste en 5 couches :

* Couche d’entrées
* Couche de convolution 2D (1 -> 16)
* Couche de convolution 2D (16 -> 32)
* Couche fully connected (32 -> 10)
* Couche de sortie

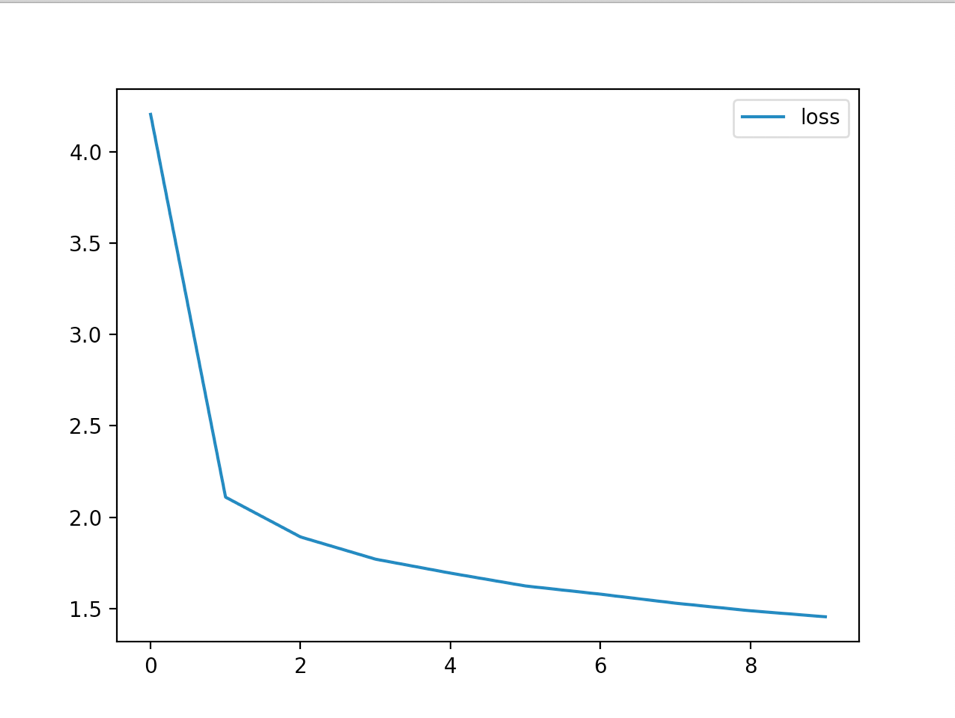
Le but des couches de convolution est d’extraire des maps de features afin que les couches linéaires fully connected se basent sur des features plutôt que sur chaque pixel d’entrée. Ici nous extrayons d’abord 16 features grace à 16 filtres différents puis à partir de ces features on extrait de nouveau 32 features. On utilise une taille de kernel de 5 et un stride de 2 afin d’accélerer le processus (Un décalage de 1 pixel est bien plus long).

Sur le set de validation on trouve une précision de 8765/10000 (87%)et une perte moyenne de **0.3326**.

La structure exacte est telle que :

****

On obtient la fonction de perte :



**Modèle 3**