

# AMAL rapport 1 : Introduction à Pytorch

SORBONNE UNIVERSITÉ  
Jean Soler, Nicolas Castanet

October 2020

## Abstract

Ce rapport a pour but de résumer les travaux menés durant les 3 premiers TP de l'UE Advanced Machine Learning and Deep Learning (AMAL) du master DAC. Ces trois TP ont pour but la prise en main des outils et des opérations usuelles du Deep Learning : Pytorch, autograd, graphe de calcul différentiel, descente de gradient. Les algorithmes seront illustrés sur les données immobilières Boston Housing dans un premier temps (TP1 et 2) puis sur les données MNIST (TP3).

## 1 TP1 : Implementation de fonctions en pytorch

### 1.1 Phase Forward/ Backward

Lors du premier TP, nous avons implémenté les fonctions nécessaires à la régression linéaire en pytorch, ici  $X \in \mathbb{R}^{q \times n}$ ,  $W \in \mathbb{R}^{n \times p}$ ,  $b \in \mathbb{R}^p$  et  $Y \in \mathbb{R}^{q \times p}$ . Avec  $q, n, p$  respectivement la taille du batch, la dimension des entrées  $x$  et la dimension des sorties  $y$ .

- La fonction linéaire :  $\hat{Y} = f(X) = X \cdot W + b$
- La fonction coût des moindres carrés :  $C = MSE(\hat{Y}) = (1/q) \cdot \|\hat{Y} - Y\|^2$

Les formules ci-dessus permettent le calcul d'une sortie  $Y$  et d'un coût  $C$  selon une entrée  $X$  et des matrices de paramètres  $W$  et  $b$ , cette phase correspond à la phase **Forward**, la seconde phase correspond, pour chaque fonction, au calcul du gradient de la sortie par rapport à ces entrées, ce qui permet par chaîne d'obtenir le gradient de l'erreur par rapport aux paramètres du modèle, c'est la phase **Backward**:

- $\frac{\partial C}{\partial W} = \frac{\partial C}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial W} = (2/q) \cdot X^T \cdot (\hat{Y} - Y)$
- $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial b} = (2/q) \cdot (\hat{Y} - Y) \cdot \text{sum}(0)$  (somme sur le premier axe)

Ici, nous n'utilisons pas **autograd** de pytorch. Lors de la phase forward, nous retenons dans chaque fonction les tenseurs utiles au calcul des gradients dans un objet contexte, nous appelons ensuite cet objet lors de la phase backward pour la propagation du gradient de l'erreur jusqu'aux paramètres.

## 1.2 Descente de gradient

Une fois les fonctions ci-dessus implémentées et vérifiées grâce à l'outil **gradcheck** de pytorch, nous devons implémenter, encore à la main, l'algorithme de la descente de gradient, voici comment se déroule une itération :

- 1) On initialise 2 objets `Context()`, pour retenir les tenseurs de la fonction linéaire et MSE.
- 2) Phase **Forward** : on calcul la sortie  $\hat{Y}$  de la fonction linéaire fonction de des parametres  $W, b$  et de l'entrée  $X$ . Puis on calcul la sortie de la fonction coût MSE fonction de  $\hat{Y}$ . Les objets `Context()` retiennent alors les entrées des fonctions à chaque étape.
- 3) Phase **backward** : on calcul dans un premier temps le gradient du coût en fonction de  $\hat{Y}$ , le tenseur d'entrée de la fonction MSE. Nous pouvons ensuite calculer la gradient de  $\hat{Y}$  en fonction des paramètres  $W$  et  $b$ . Ainsi en chaînant ces deux informations on peut calculer le gradient de  $C$  en fonction des paramètres.
- 3) Une fois ce gradient calculé, on amène les paramètres dans la direction opposée au gradient afin de minimiser le coût. On pondère ce pas de gradient par la learning rate  $\alpha$ .

Voici la courbe d'apprentissage observée sur les données normalisées boston housing (on considère ici seulement un ensemble d'apprentissage).

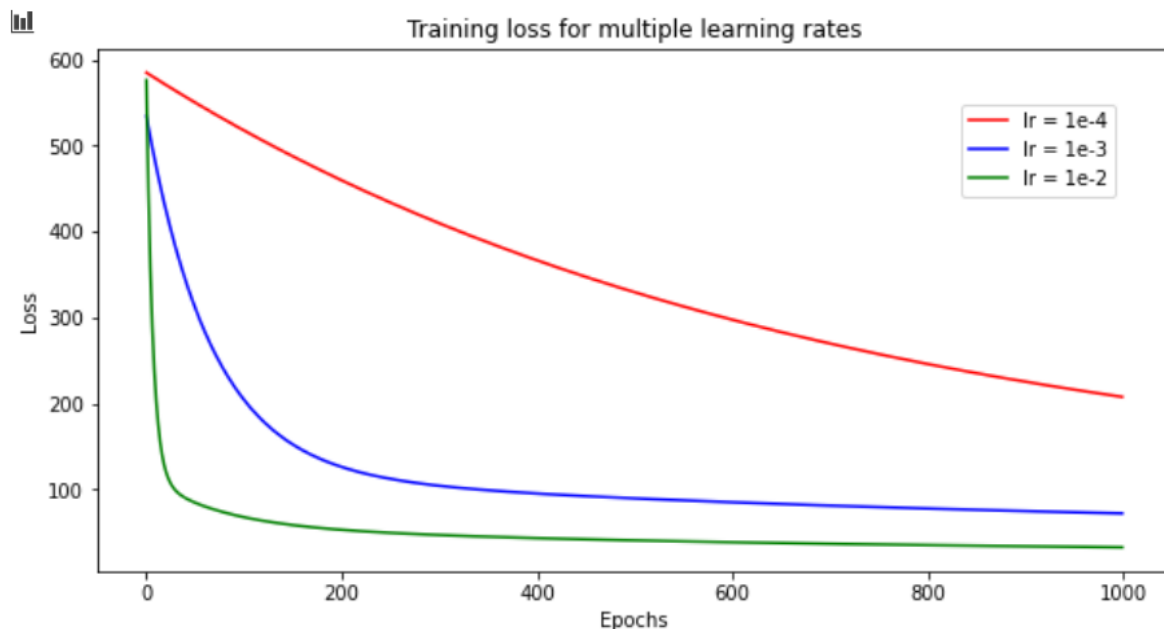


Figure 1: Courbe d'évolution du coût MSE sur 1000 epochs pour différentes learning rate sur les données boston housing.

On voit bien que la courbe converge plus vite avec une learning rate plus élevée, cependant elle diverge vers l'infinie lorsque  $lr > 0.26$ .

## 2 TP2 : Utilisation des fonctions Pytorch

Le but de cette session à été d'obtenir le même résultat mais en utilisant les fonctionnalités déjà implémentées dans Pytorch :

- **Autograd** : nous effectuons à présent la rétropropagation du gradient de manière automatique avec la fonctionnalité autograd de Pytorch.
- **Optimiseur** : Nous utilisons un objet optim Pytorch permettant l'automatisation de la mise à jour des paramètres après la rétropropagation et d'utiliser plusieurs différent type d'algorithme.
- **Module Pytorch** : permet d'encapsuler plusieurs couches (linéaire, activation, loss ...) dans un seul module.

### 2.1 Campagne d'expérience sur la taille des batchs

Nous avons expérimenté la descente de gradient pour différentes taille de batch. A chaque époques on calcul la loss et on propage les gradient pour tous les batchs, les batchs sont choisis aléatoirement. Voici les courbes d'apprentissage pour quelques résultats :

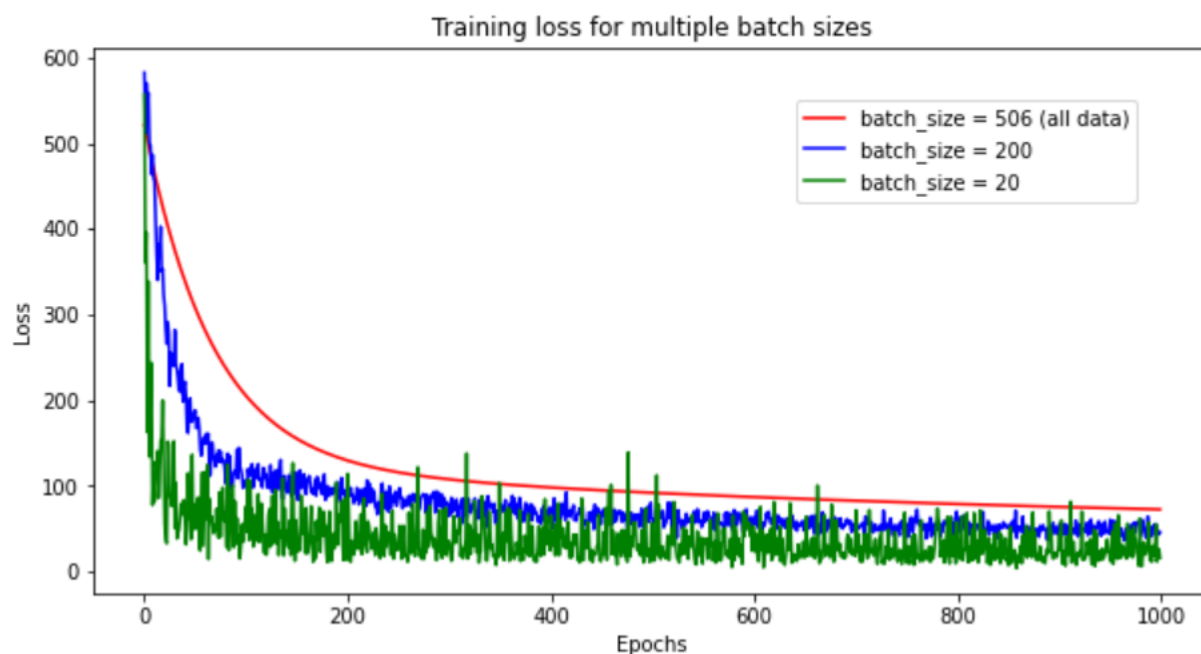


Figure 2: Courbe d'évolution du coût MSE sur 1000 epochs avec  $lr = 1e-3$  pour différentes taille de batch sur les données boston housing.

- **Algorithme Batch** : 1 epoch = 1 pas de gradient.  
Pour chaque pas de gradient, on utilise la loss moyenne sur toutes les données. Il en résulte une courbe lisse au long des epochs ainsi qu'un temps d'apprentissage rapide du à la possibilité de calcul matriciel sur toute les données. Cependant, la convergence est lente car on met à jour les paramètres une fois par epoch.

- **Algorithme SGD** : 1 epoch = 1 pas de gradient par exemple.

La convergence est donc beaucoup plus rapide ce qui peut être utilisé pour de gros dataset. En effet, on n'aurait pas à calculer le gradient sur toutes les données pour chaque mise à jour des paramètres. L'apprentissage n'est plus moyenné sur l'ensemble des données ce qui induit du bruit. L'inconvénient majeur est la durée d'apprentissage, on prend les exemples 1 par 1 on ne peut donc pas utiliser le calcul matriciel.

- **Algorithme Mini-Batch** : 1 epoch = 1 pas de gradient par mini batch.

Cet algorithme est un compromis des deux précédents. On met à jour les paramètres plus fréquemment que pour le SGD et il est possible d'utiliser le calcul matriciel sur chaque mini-batch. Il en résulte un calcul moins long que pour le SGD avec une convergence plus rapide que pour le batch. La taille des mini-batches est un hyper-paramètre à régler.

Voici le temps d'apprentissage pour ces différents algorithmes, ce qui illustre les remarques précédentes :

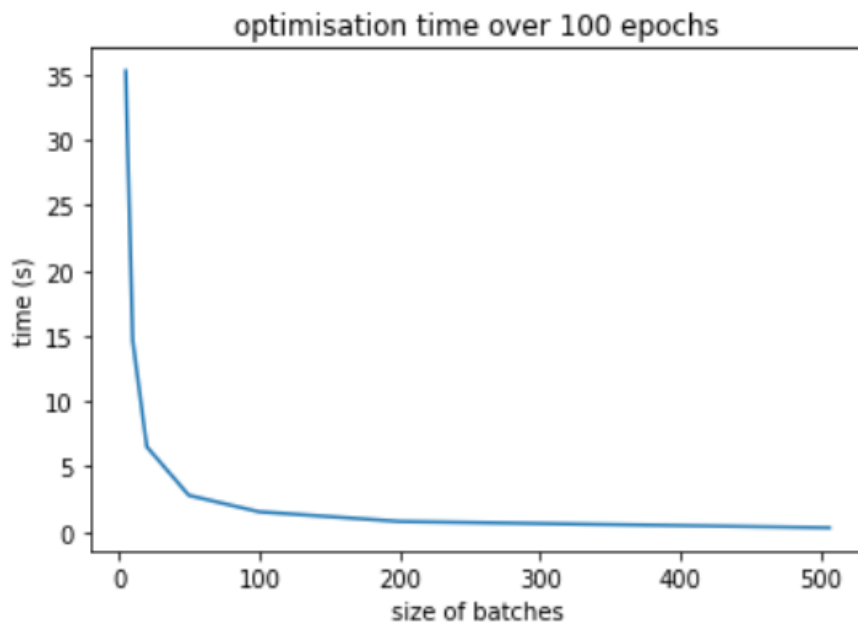


Figure 3: Temps d'apprentissage en fonction de la taille des mini-batches pour 1000 epochs de régression linéaire sur les données Boston Housing.

### 3 TP3 : Autoencoder, Dataset, Dataloader, GPU, Checkpointing

Ce TP clôture l'exploration non exhaustive de Pytorch. Voici les nouvelles fonctionnalités que nous utilisons dans cette session pour nos expériences :

- **Dataset, Dataloader** : permet la gestion efficace des données du chargement au prétraitement. Notamment ils permettent de charger les données une par une en mémoire ce qui est primordiale pour les gros dataset. Ils permettent aussi de choisir la taille du atchbatch.
- **Passage des tenseurs sur GPU** : permet de gagner en temps de calcul.
- **Checkpointing** : permet de sauvegarder le modèle et ses paramètres à un moment donné de l'apprentissage.

### References