Deep Learning : Labs 01 & 02

Ben Kabongo
M1 Data Science, 22210136

Avril 2023

1 Objectifs

Le but des deux premiers travaux pratiques du cours de **Deep Learning** ont été la compréhension des réseaux de neurones et leur implémentation from skratch, sans l'utilisation de bibliothèques d'apprentissage profond dédiées telles que **Pytorch**, que nous verrons dans la suite du cours.

Les réseaux de neurones implémentés dans nos travaux pratiques ont été entrainés et évalués sur la base **MNIST**, qui est une base de données d'images de chiffres manuscrits largement utilisée pour entraîner des algorithmes de classification d'images. Elle contient 60 000 images d'entraînement et 10 000 images de test. Les images sont en noir et blanc et ont une taille de 28x28 pixels. Chaque image est étiquetée avec le chiffre qu'elle représente.

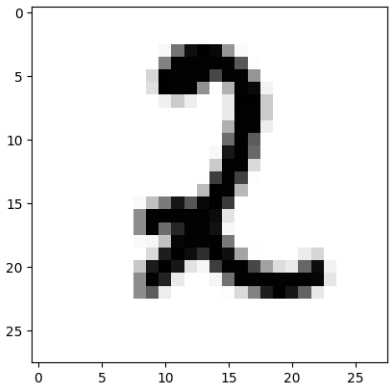


FIGURE 1 – MNIST - 2

Dans ce rapport, nous allons présenter la structure des réseaux de neurones que nous avons utilisée, expliquer les détails d'implémentation des classes et des fonctions pertinentes et enfin présenter les différentes expérimentations que nous avons réalisées.

2 Structure des réseaux de neurones

Nous allons ici présenter la structure des réseaux de neurones implémentés, en utilisant l'approche modulaire des réseaux de neurones. En effet, dans la littérature scientifique, cette approche simplifie énormément la compréhension des réseaux de neurones.

Un module M^k peut être assimilé à une couche (ou une fonction d'activation) du réseau de neurones. Il prend une entrée des données z^{k-1} d'une certaine taille, auquel il applique une fonction f^k pour produire une sortie z^k d'une taille donnée.

Certaines modules ont des paramètres ϕ^k, qui sont les poids des perceptrons de la couche correspondante. D'autres types de modules n'ont pas de paramètres.

On obtient donc :

$$M^k : z^k = f^k(z^{k-1}, \phi^k)$$

Les réseaux de neurones sont donc par conséquent un empilement de module les uns à la suite des autres : tel que la sortie du module précédent est l'entrée du module courant.

Les réseaux de neurones sont donc composés de trois types de modules :

— **Module linéaire** : le module linéaire est composé de perceptrons d'une couche. Ses paramètres sont W l'empilement des poids $w_{j,i}$ et b des biais b_j de chaque perceprton j.
Le module prend en entrée la sortie du module précédent x auquel il applique une transformation linéaire affine $z = Wx + b$, où z l'empilement des sorties des perceptrons.

— **Module d'activation** : les modules d'activation prennent en entrée le résultat d'un module linéaire auquel ils appliquent une fonction d'activation non linéaire telle que **relu** ou **tanh**.

— **Module de coût** : le module de coût prend entrée les sorties du réseau pour le calcul de la **loss** en fonction des classes réelles des exemples. Dans notre cas, la loss utilisée est la **cross-entropy loss**. Il existe d'autres loss telles que le **mean squared error**.

L'étape de **forward** consiste donc à faire passer les données d'entrées module après module en partant du début du réseau. L'étape de **backpropagation** consiste à calculer le gradient de la **loss** par rapport aux paramètres des différents modules en partant de la fin du réseau vers le début.

Dans nos travaux, nous avons implémenté un réseau linéaire simple qui est par conséquent une combinaison d'un module linéaire et du module de coût avec la cross-entropy loss.

Egalement, nous avons implémenté une structure de réseau beaucoup plus complexe avec un nombre arbitraire de couches cachées, différentes fonctions d'activations entre les couches. Les neurones d'une couche donnée sont connectés à tous les neurones de la couche suivantes. Il s'agit donc d'un réseau entièrement connecté.

3 Détails d'implémentation

3.1 Tenseurs, paramètres

Afin de simplifier le processus **backpropagation**, nous avons codé une classe d'objet **Tensor** qui correspond à un noeud quelconque dans le graphe de notre réseau. La classe **Parameter** étend cette classe et porte uniquement sur les noeuds du réseau qui sont des paramètres.

Les noeuds du réseau sont soit des paramètres, soit les entrées, soit les sorties des couches. L'attribut **require_grad** est un booléen qui indique si le gradient de la loss par rapport au noeud doit être calculé.

Pour la passe **forward**, il est donc nécessaire de conserver les paramètres de la fonction ayant produit le noeud. Pour ensuite, lors de la passe **backward**, propager le gradient de la sortie par rapport aux paramètres en entrée. D'où l'attribut **backptr** qui sauvegardent pour un noeud donné, ses prédécesseurs dans le graphe des noeuds.

Pour chaque fonction (linéaire, activation, loss), il existe une fonction équivalente qui calcule la dérivée de la sortie par rapport aux paramètres. Ces fonctions sont ainsi sauvegardées dans le noeud auquel ils correspondent ?

3.2 Poids, biais et initialisation

Les poids et les biais du réseau sont les poids et les biais des perceptrons qui composent le réseau.

Une image de la base MNIST faisant 28x28=784 pixels et la base contenant les 10 chiffres de 0 à 9, chaque perceptron de la première couche du réseau a donc 784 poids à fixer et 1 biais. Les couches intermédiaires peuvent avoir des tailles arbitraires.

Les biais sont initialisés à 0. Les autres poids sont initiailisés avec la méthode **glorot** donnée par :

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

Ou la méthode **kaiming** donnée par :

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

Avec n_{in} la taille de l'entrée de la couche et n_{out} la taille de sortie de la couche.

3.3 Fonction linéaire

Chaque perceptron du réseau applique une fonction linéaire sur ses entrées avec ses poids et son biais correspondant. On peut empiler les poids, les biais et les sorties (avant activation) d'une couche. Cela donne la fonction linéaire suivante :

$$z = Wx + b$$

Lors de la passe backward, il faut calculer le gradient par rapport aux paramètres :

$$\frac{\partial z}{\partial W} = x$$

$$\frac{\partial z}{\partial b} = 1$$

3.4 Fonctions d'activation

Les fonctions d'activations appliquent une transformation linéaire à la sortie des perceptrons du réseau. Dans notre cas, nous avons implémenté les fonctions **relu** et **tanh**.

$$relu(x) = max(0, x)$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Les dérivées de ces fonctions par rapport à leur paramètre d'entrée sont :

$$\frac{\partial relu(x)}{\partial x} = \begin{cases} 0 & \text{si } x \leq 0 \\ 1 & \text{si } x > 0 \end{cases}$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

3.5 Fonction de coût

Pour les données MNIST, le réseau produit 10 sorties, correspondant aux 10 classes. Afin de transformer les sorties en mesure de probabilité pour chacune des classes, on utilise la fonction **softmax** donnée par la formule ci-dessous. La prédiction de la classe d'un exemple correspond par la suite à celui ayant la probailité maximale.

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

La fonction coût utilisée est la **cross-entropy loss**, dont la formule est donnée par :

$$nll(x, y) = -log(\sigma(x))_y$$

La dérivée par rapport aux entrées est donnée par :

$$\frac{\partial nll(x, y}{\partial x_i} = \begin{cases} -1 + \sigma(x)_i & \text{si } i = y \\ \sigma(x)_i & \text{si } i \neq y \end{cases}$$

Lors de la backpropagation, pour chaque paramètre, on calcule la valeur accumulée du produit du gradient de la couche suivante et de la dérivée au niveau de la fonction (linéaire, activation ou loss) correspondante.

3.6 Classes des réseaux de neurones

Nous avons implémenté deux classes de réseaux de neurones : une première plus simple **LinearNetwork** et une autre beaucoup plus complexe **DeepNetwork**.

3.6.1 LinearNetwork

La classe LinearNetwork est une classe de réseau de neurones à une seule couche de transformation linéaire affine couplée à la cross-entropy loss. Il n'y a pas de fonction d'activation.

Les paramètres du réseau sont donc les poids de l'unique couche. Dans le cas des données MNIST, il y a donc 7840 (poids W) + 10 (biais) : au total 7850 paramètres.

En dépit de sa simplicité, les performances sur les données de tests dépassent les 90% de bonne prédiction avec les bons hyperparamètres. Nous présenterons les résultats dans la section expérimentation.

3.6.2 DeepNetwork : réseau plus complet

La classe DeepNetwork est une classe réseau de neurones à plusieurs couches. Le nombre de couches est un paramètre de la classe à fixer. Comme fonction d'activation, on choisit d'utiliser soit la fonction **relu** soit la fonction **tanh** pour toutes les couches. La fonction de perte est la **cross-entropy loss**.

Le réseau a une taille d'entrée et une taille de sortie, donnés par la taille d'une image (ici 784) et le nombre de classes (ici 10). Les tailles de l'entrée de la première couche et de la sortie de la dernière couche y sont donc contraints. La taille des couches internes peut être variables. Nous avons optés ici pour des couches internes de même taille.

3.7 Optimiseur, apprentissage, validation et test

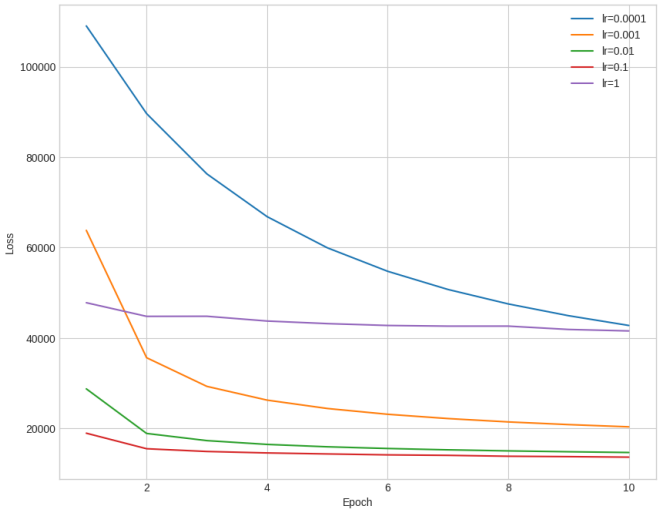
Afin de centraliser le processus de mise à jour des poids lors de la backpropagation, nous avons implémenté la classe **Optimizer** qui permet de mettre à jour les poids d'un réseau.

Pour apprendre les poids du réseau, nous avons divisé les données en trois ensembles : apprentissage, validation et test. Le réseau est entraîné sur les données d'apprentissage pour lequel nous calculons la loss en apprentissage. Afin d'être sûr que nous ne sur-apprenons pas, nous calulons la quantité de bonne classification sur les données de validation à chaque époque d'entraînement. Et, enfin, nous calculons la quantité de bonne classification sur les données de test.

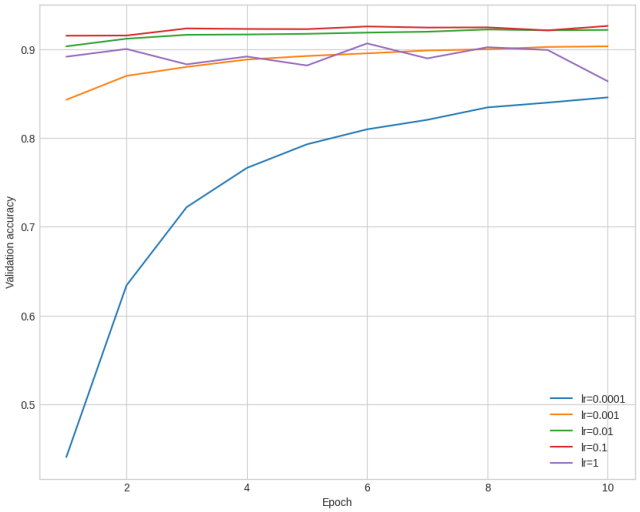
4 Expérimentations

Avec les bons hyperparamètres, l'apprentissage sur les données MNIST donne des performances plutôt excellentes, supérieures à 90%. Certains réseaux atteignent une performance de 97% en test.

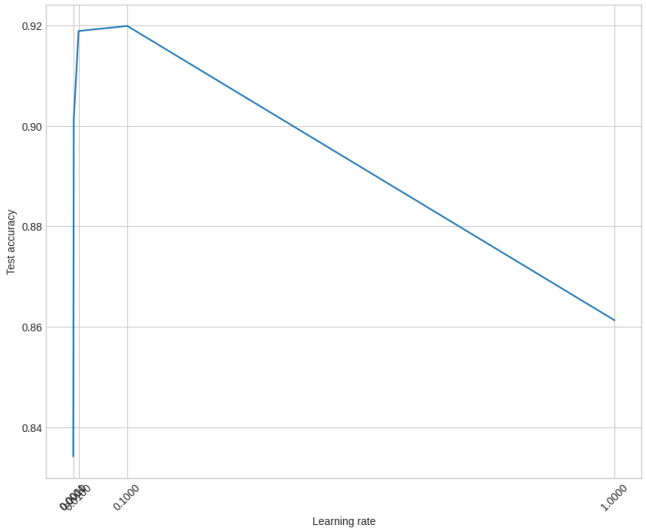
Nous avons étudié l'impact de la variation des différents hyperparamètres de nos réseaux de neurones sur l'accuracy. Nous allons ici présenter quelques résultats.



(a) Linéaire - Learning rate - Train Loss

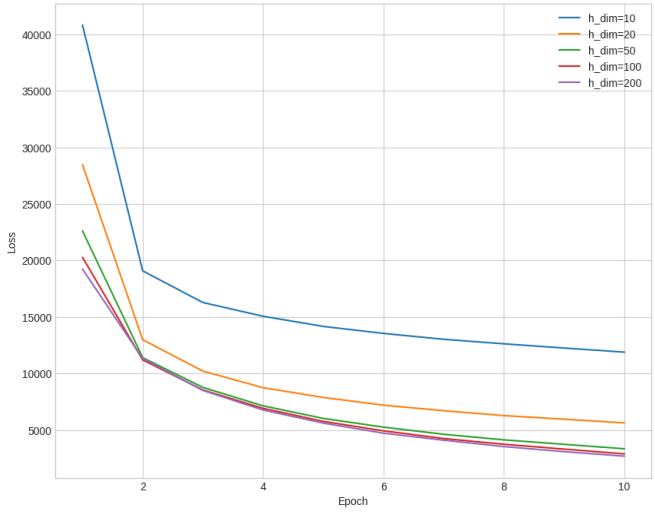


(b) Linéaire - Learning rate - Validation accuracy

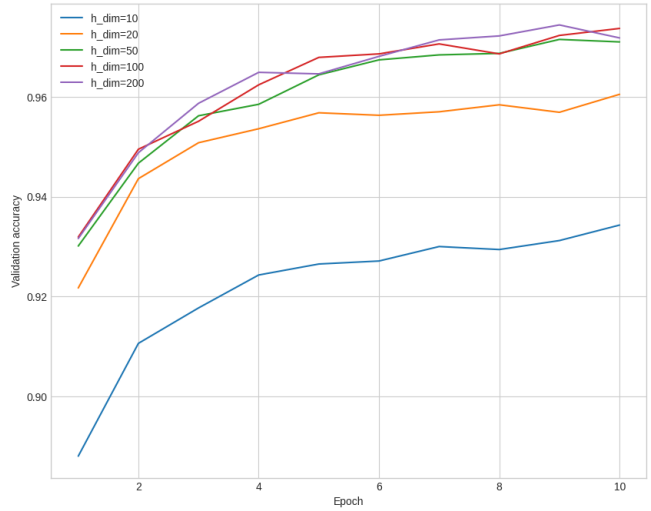


(c) Linéaire - Learning rate - Test accuracy

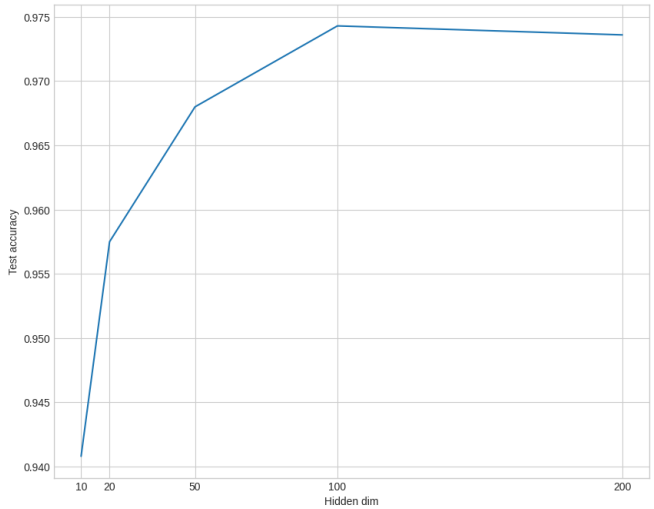
FIGURE 2 – Linéaire : learning rate, n_epochs=10



(a) Deep - Hidden dim - Train Loss

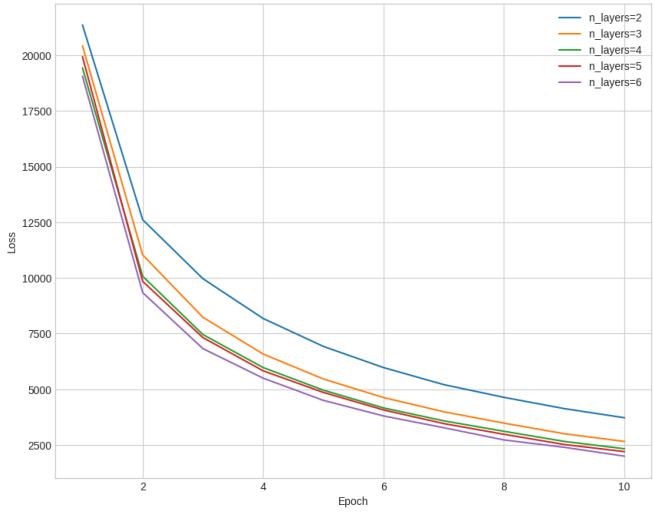


(b) Deep - Hidden dim - Validation accuracy

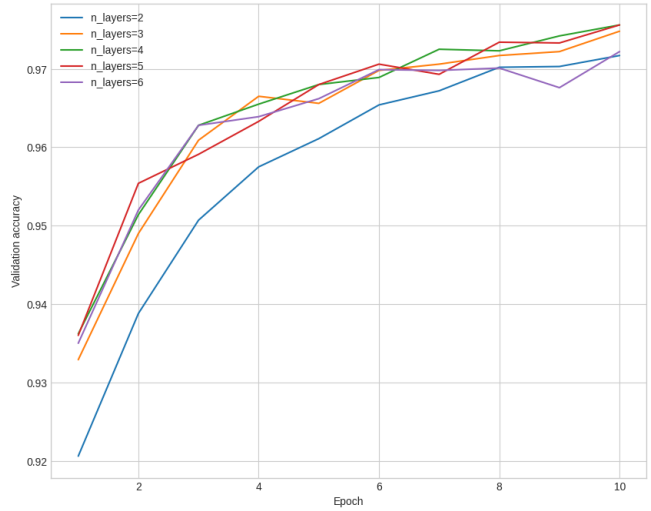


(c) Deep - Hidden dim - Test accuracy

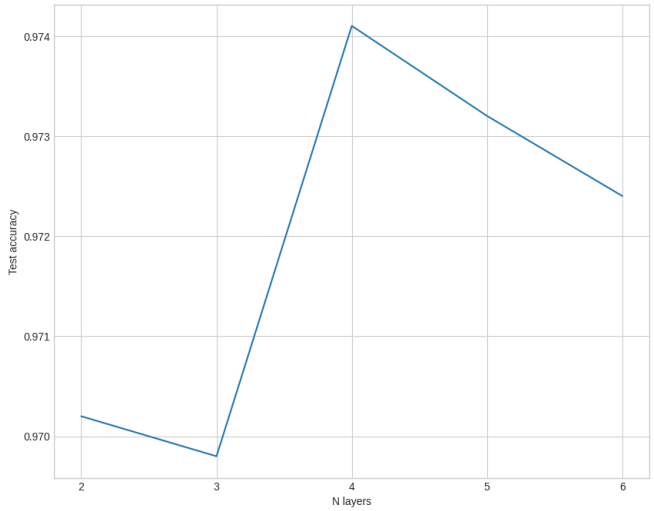
FIGURE 3 – DeepNetwork : lr=1e-2, n_epochs=10, n_layers=3, use_tanh=True



(a) Deep - N layers - Train Loss



(b) Deep - N layers dim - Validation accuracy



(c) Deep - N layers - Test accuracy

FIGURE 4 – DeepNetwork : lr=1e-2, n_epochs=10, hidden_dim=100, use_tanh=True