AMAL rapport 1 : Introduction à Pytorch

Sorbonne Université
Jean Soler, Nicolas Castanet

October 2020

Abstract

Ce rapport à 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 nécessaire à la **regression linéaire** en pytorch, ici $X \in \mathbb{R}^{q \cdot n}$, $W \in \mathbb{R}^{n \cdot p}$, $b \in \mathbb{R}^p$ et $Y \in \mathbb{R}^{q \cdot p}$. 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 sa 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**:

- $\frac{\partial C}{\partial W} = \frac{\partial C}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial W} = (2/q) \cdot X^T \cdot (\hat{Y} - Y)$
- $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial b} = (2/q) \cdot (\hat{Y} - Y) \cdot sum(0)$ (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 α.

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

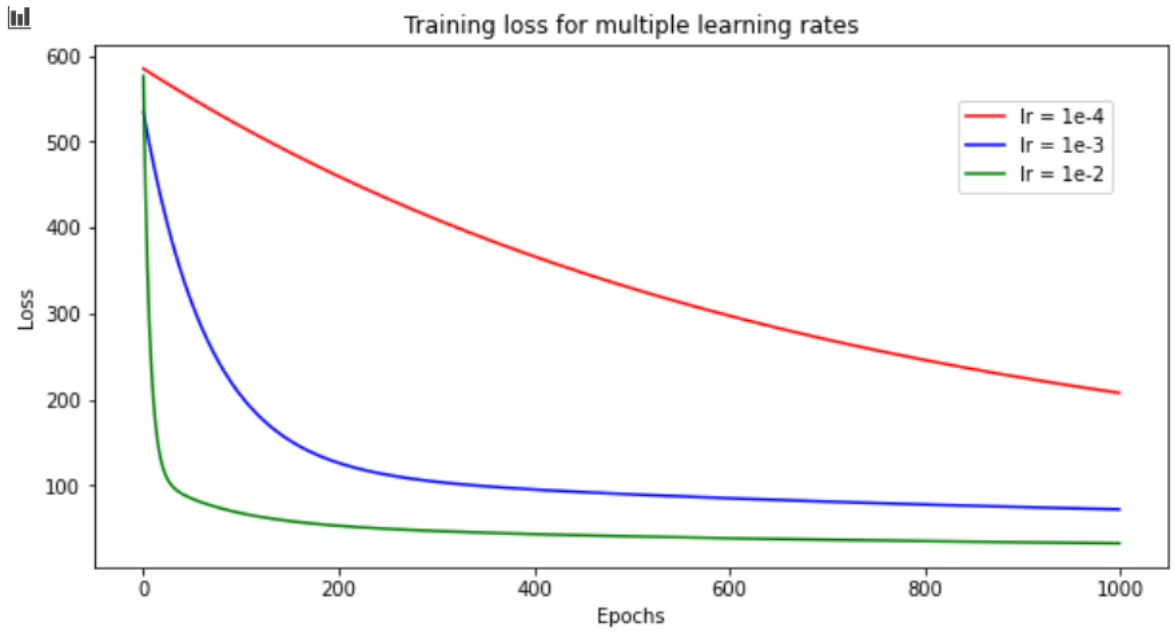

Figure 1: Courbe d'évolution du coût MSE sur 1000 epochs pour différentes learning rate sur les données boston housing.

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 Pytoch 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 :

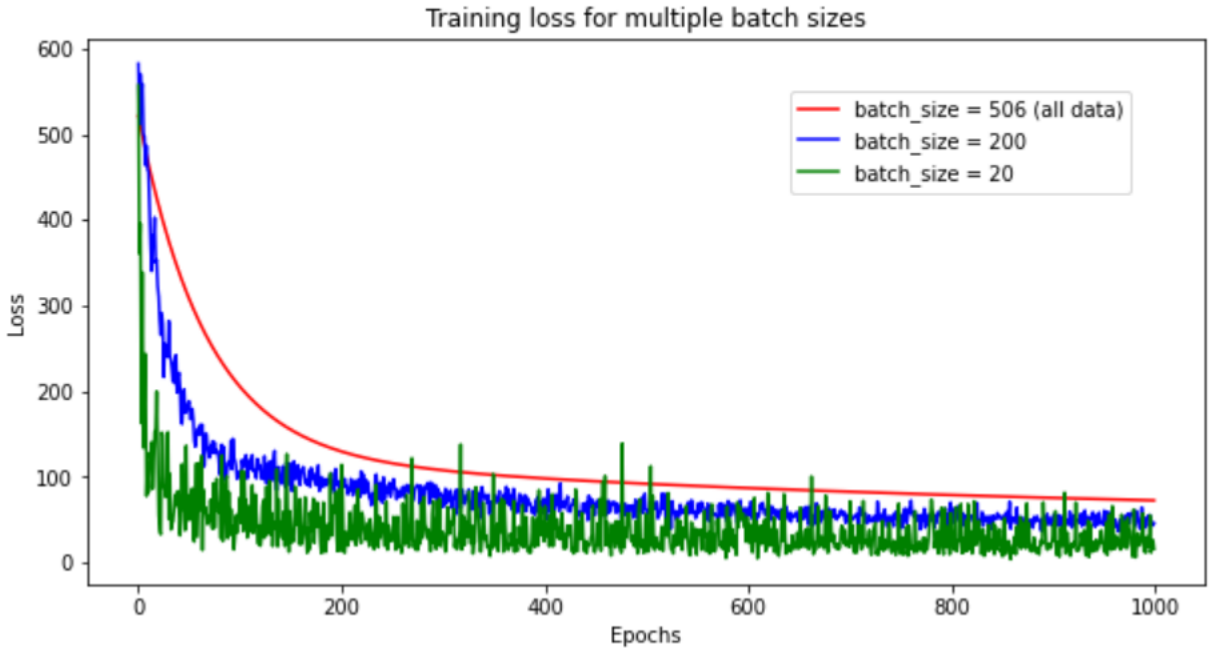

Figure 2: Courbe d'évolution du coût MSE sur 1000 epochs avec lr = 1e-3 pour différentes taille de batch sur les données boston housing.

- **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 utilise 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 donnée 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 précédent. 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-batchs est un hyper-paramètre à régler.

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

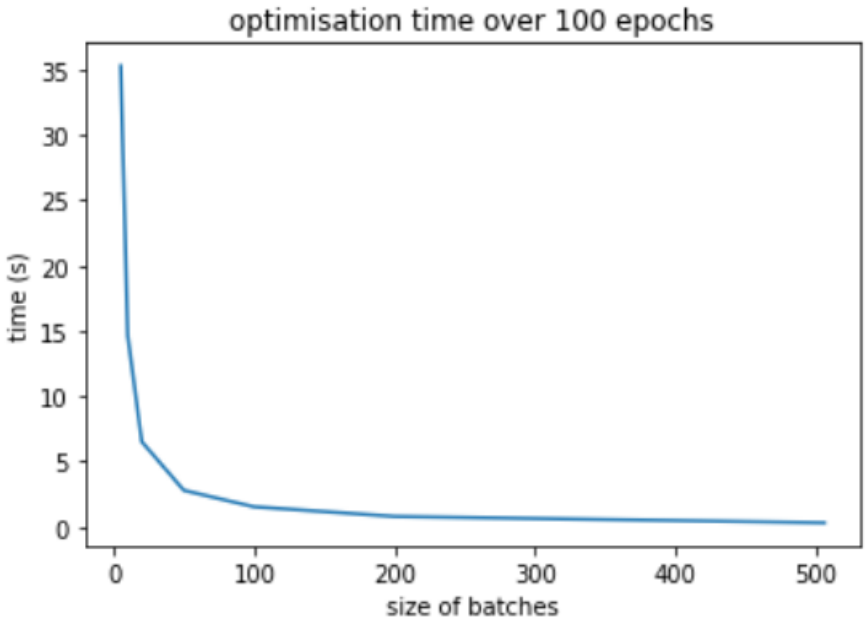

Figure 3: Temps d'apprentissage en fonction de la taille des mini-batchs pour 1000 epochs de régression linéaire sur les données Boston Housing.

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, Dataloarder** : 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.

3.1 Campagne d'expérience sur l'autoencoder

Nous explorons les outils cités précédemment dans le cadre d'un autoencodeur sur les données MNIST, le but est d'apprendre une représentation compacte des données.
Dans notre implémentation, les poids de l'encodeur correspondent à la transposée des poids du décodeur. Ce qui réduit le nombre de paramètre à optimiser et donc réduit le temps d'optimisation et le risque de sur apprentissage.

3.1.1 Setup expérimental

Nous considérons dans un premier les hyperparamètres suivants : la dimension de la couche d'encodage et le nombre de couches.
Nous réalisons ces expériences avec un ensemble d'apprentissage de 60000 exemples et un ensemble de test de 10000 exemples. Ces derniers sont encapsulés dans des objets **Dataset** et **Dataloader** qui mélangent les données à chaque epoch.
Nous utilisons une **taille de batch à 100**, une **learning rate à 1e-3** sur **50 epochs** et un **coût MSE**.

3.1.2 Dimension de la couche d'encodage

Dans cette section, nous étudions un autoencodeur avec **une couche d'encodage** afin d'observer les effet de sa dimension. Le but étant d'obtenir la représentation latente des données la plus compacte possible qui permet de reconstruire le plus fidèlement les données. Réduire la dimension de la couche d'encodage force l'autoencoder à capter les patterns discriminants des données.
Voici les résultats des expériences pour différentes taille d'encodage :

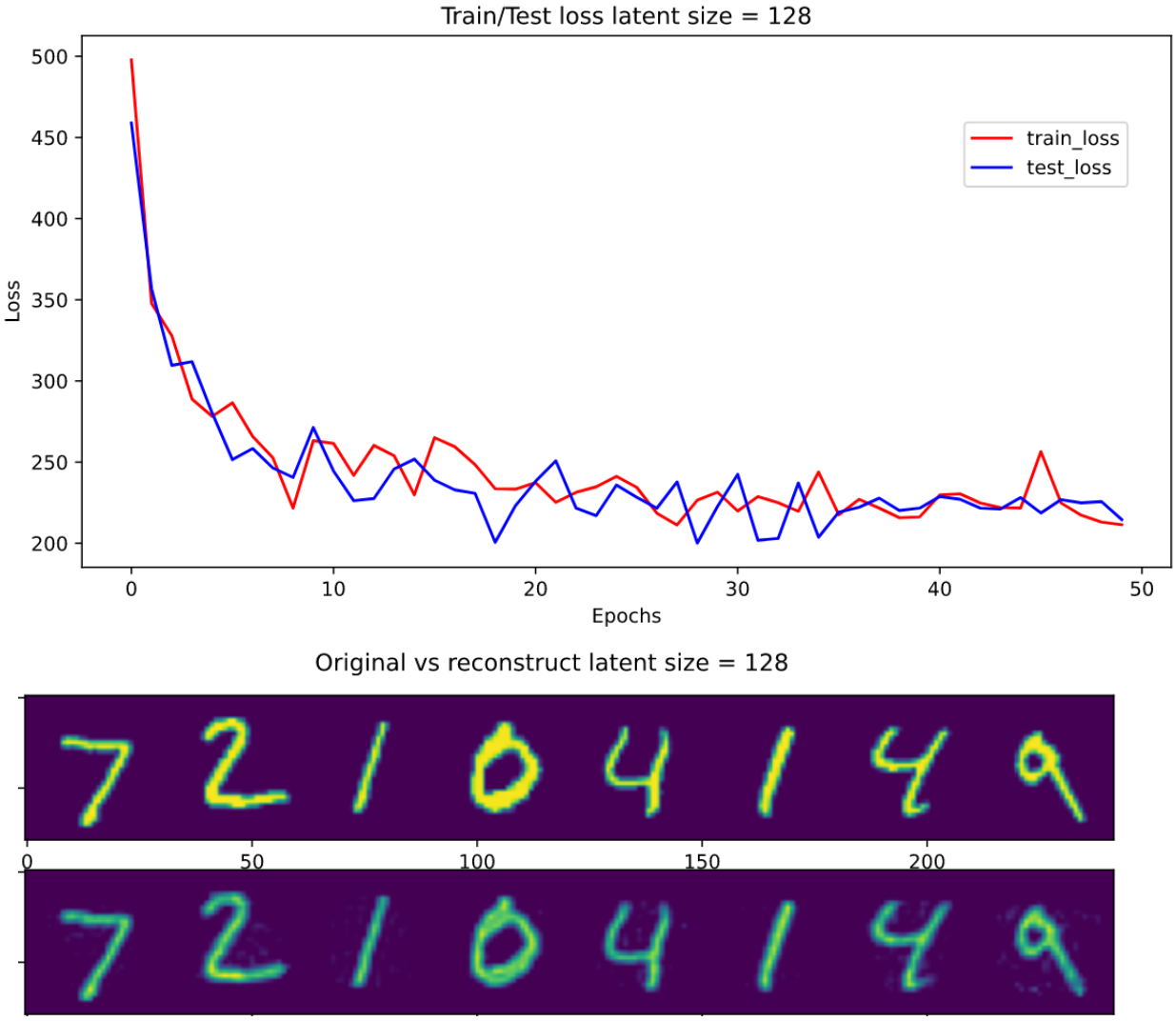

Figure 4: Courbe train/test du coût MSE sur 50 epochs et reconstruction de chiffres de l'ensemble test. Une couche d'encodage de taille 25.

En passant d'une représentation de dimension 784 (image applatie) à 128, l'autoencodeur est capable d'apprendre les patterns nécessaire à la reconstruction des chiffres comme le montre les résultats ci-dessus. Cependant, il serait souhaitable de diminuer encore la taille d'encodage afin de capter les patterns les plus importants.

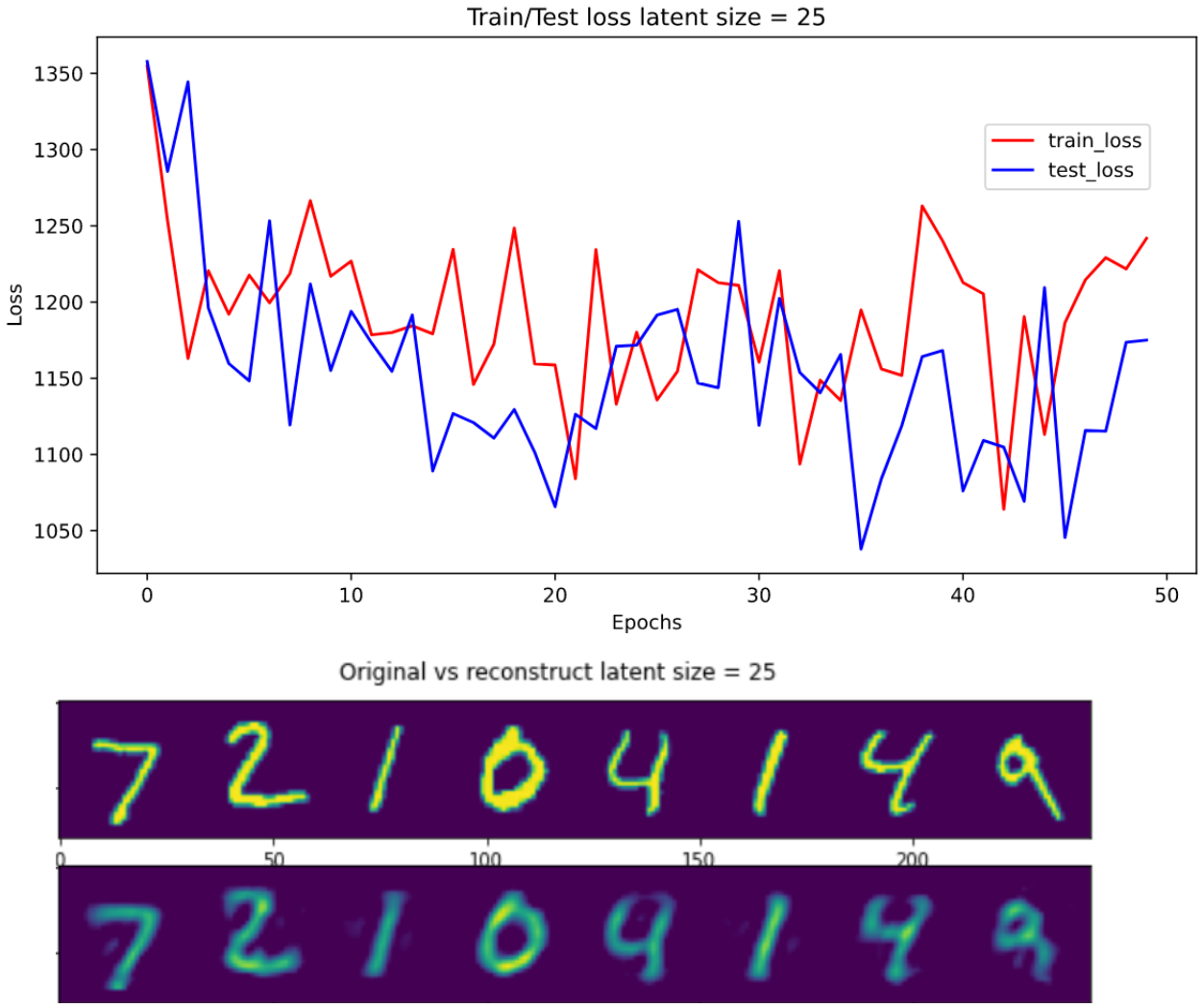

Figure 5: Courbe train/test du coût MSE sur 50 epochs et reconstruction de chiffres de l'ensemble test. Une couche d'encodage de taille 25.

On observe ici qu'en passant d'une représentation de dimension 784 à 25, l'autoencodeur ne parvient pas à apprendre les patterns discriminants, la reconstruction des chiffres est de mauvaise qualité. Soit le nombre de paramètre est insuffisant pour apprendre une représentation, soit l'apprentissage de patterns n'est pas assez graduel, il faut alors rajouter une couche intermédiaire.

3.1.3 Nombre de couches d'encodage

Le nombre de couches impacte directement le nombre de paramètres à entraîner du modèle et donc augmente le temps d'apprentissage, mais aussi le risque de sur-apprentissage. Cependant, si on désire obtenir une dimension d'encodage très faible, on est obliger de réduire petit à petit la dimension des données à travers plusieurs couches, afin que ces dernières captent de manière graduelle les patterns présents dans les données.

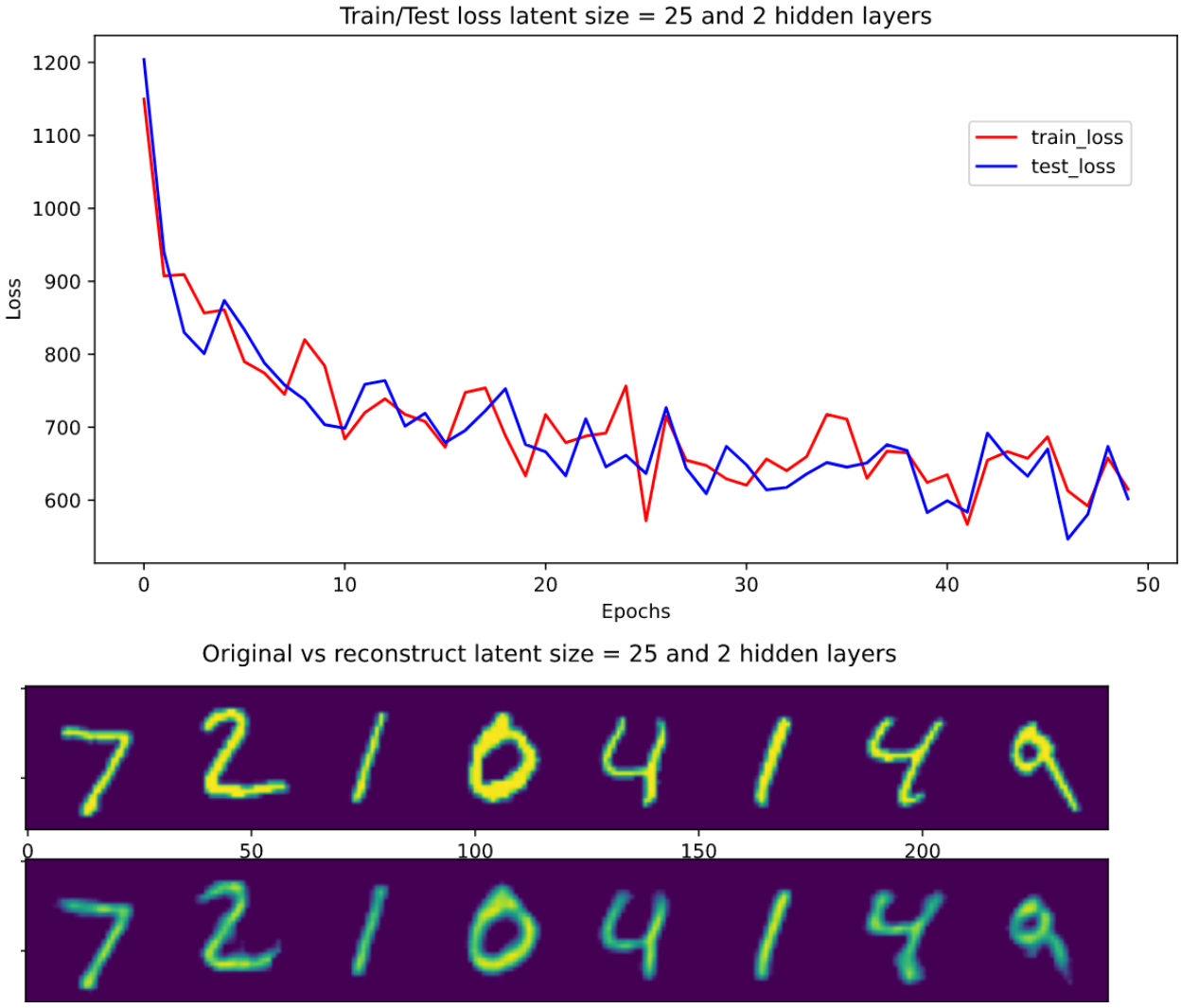

Figure 6: Courbe train/test du coût MSE sur 50 epochs et reconstruction de chiffres de l'ensemble test. Taille d'encodage = 25 avec une couche d'encodage intermédiaire (et donc de décodage) supplémentaire de taille 128.

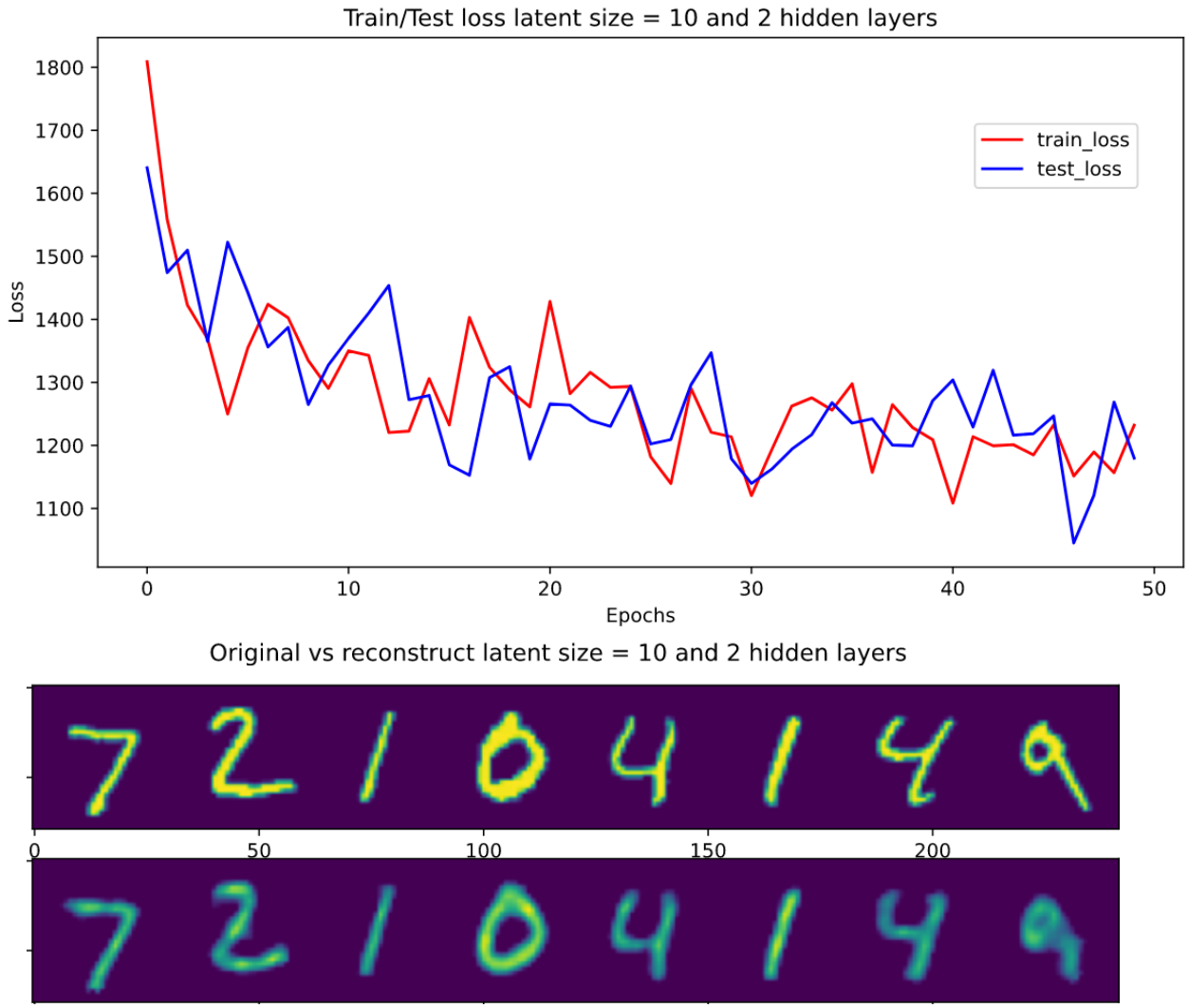

Figure 7: Courbe train/test du coût MSE sur 50 epochs et reconstruction de chiffres de l'ensemble test. Taille d'encodage = 10 avec une couche d'encodage intermédiaire (et donc de décodage) supplémentaire de taille 128.

References