Sorbonne université

Rapport

AMAL - Rapport des TMEs 1, 2 et 3

Merrouche Aymen

Novembre 2020

Contents

Chapter 1

INTRODUCTION

Ce rapport consigne les résultats expérimentaux des TMEs 2 et 3. Dans la première partie, on va comparer sur une tache de régression pour les données "Bonston Housing" trois méthodes de descentes de gradients : la descente de gradient stochastique, la descente de gradient mini-batch et la descente de gradient batch. Dans la deuxième partie, on va entraîner un réseau de neurones "auto-encodeur" sur la base de données de "MNIST" dont le but est de projeter les données en entrée sur un espace de dimension moindre. Enfin, dans la dernière partie, on va entraîner un réseau de neurones "HighWay" pour la tache de classification, toujours, sur la base de données "MNIST".

Chapter 2

Descente de gradient :

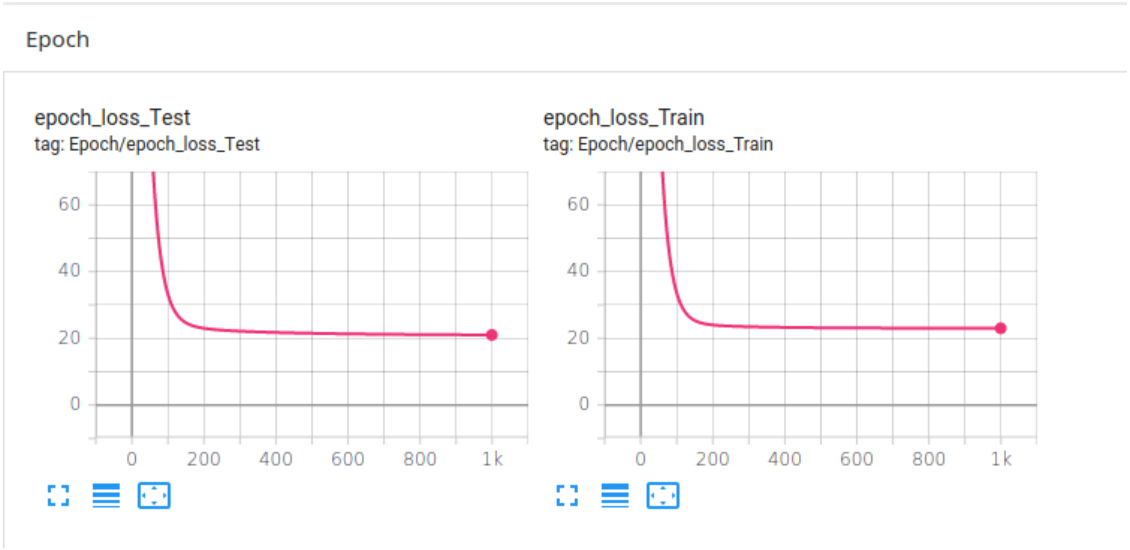
2.1 Comparaison entre la descente de gradient stochastique, mini-batch et batch

Dans ce qui va suivre, on va comparer en traçant les courbes du coût en apprentissage et celles en test pour les trois algorithmes de descente de gradient (SGD, batch et mini-batch) pour la tache de régression sur les données "boston housing".

- Données : Boston Housing, 33% en test et 67% en train. Les features sont centrées et réduites (moyennes et écarts-types calculés sur le train).
- Modèle : linéaire.
- Loss : MSE.
- Hyper-paramètres :
  - Learning rate : 0.001.
  - nombre d'époques : 1000.
- On affiche le coût sur le train set et le test set à la fin de chaque époque.

2.1.1 Descente de gradient batch :

Courbe du coût en apprentissage et celle en test - **Descente de gradient batch**



Remarques : On remarque que l'algorithme converge en ≈ 400 itérations et se stabilise sur un minimum de la fonction d'erreur.

2.1.2 Descente de gradient stochastique (SGD) :

Courbe du coût en apprentissage et celle en test - **Descente de gradient stochastique**

Remarques : On remarque que l'algorithme "converge" (ou plutôt oscille dans un intervalle stable) après très peu d'itérations (≈ 15) et ce en apprentissage et en test. On remarque aussi dans les courbes des zigzags (des piques) (plus accentués en test). La mise à jour du gradient pour chaque donnée introduit un bruit

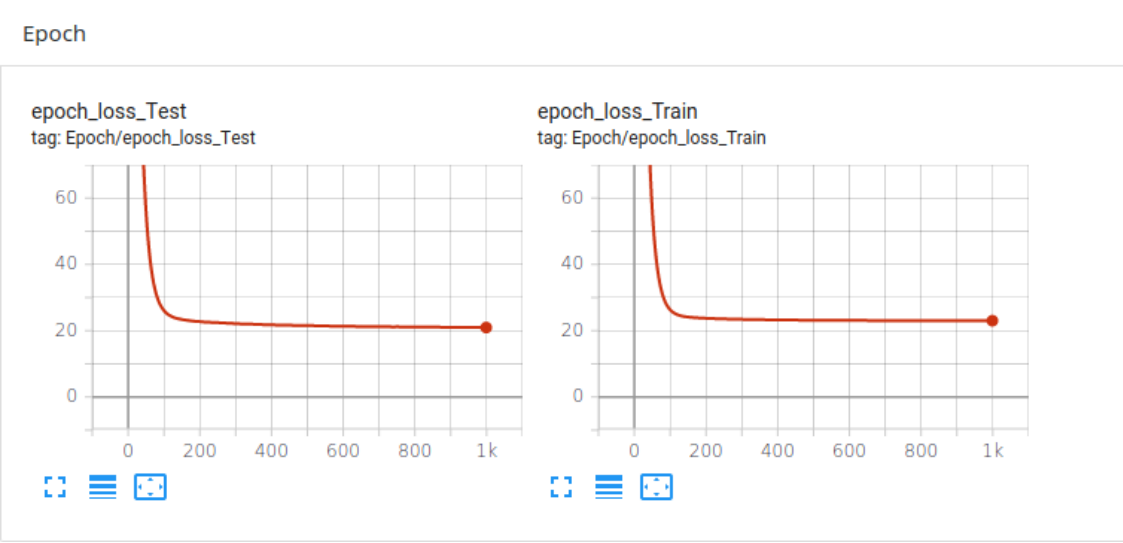
dans le signal du gradient, donc les poids et le biais mis à jour et par conséquent l'erreur oscilleront (grande variance) ce qui rend difficile la stabilisation sur un minimum de la fonction d'erreur.

2.1.3 Descente de gradient mini-batch :

Courbe du coût en apprentissage et celle en test - **Descente de gradient mini-batch taille 5**

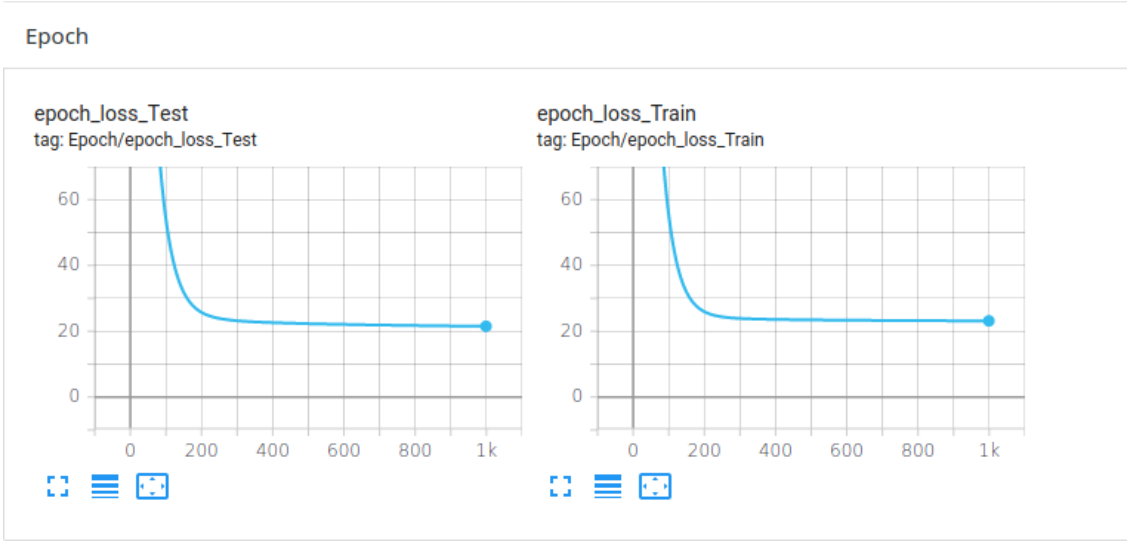
Taille du mini-batch 5 : L'algorithme converge après peu d'itérations (≈ 50). On remarque quand même la présence de bruit dans la courbe d'erreur.

Courbe du coût en apprentissage et celle en test - **Descente de gradient mini-batch taille 25**



Taille du mini-batch 25 : L'algorithme converge après ≈ 150 itérations.

Courbe du coût en apprentissage et celle en test - **Descente de gradient mini-batch taille 50**



Taille du mini-batch 50 : L'algorithme converge après ≈ 250 itérations.

Remarques : D'une part on remarque que plus la taille du batch est grande plus l'algorithme nécessite plus d'itérations pour converger. D'autre part une taille de batch plus grande et donc une mise à jour mois fréquente du gradient(variance réduite) entraîne une convergence plus stable.

2.1.4 Comparaison et analyse :

Le SGD converge très rapidement, cependant avec une très grande variance de l'erreur. Plus la taille du mini-batch est grande plus on converge lentement, parallèlement cette convergence est plus stable. Lorsque la taille du mini-batch est égale à la taille du train set la descente de gradient par mini-batch revient à faire une décente de gradient batch. Une fréquence de mise à jour du modèle plus grande peut entraîner un apprentissage plus rapide de certains problèmes.

- La descente de gradient batch calcule le gradient sur toutes les données d'apprentissage, cette méthode est adaptée pour des fonctions convexes/assez régulières. Dans ces cas, cette méthode (pour un pas de gradient et une initialisation adéquate) permet généralement de converger vers un extrema global.
- La descente de gradient stochastique calcule le gradient sur un seul exemple des données d'apprentissage ce qui est adapté pour des fonctions qui admettent plusieurs extrema locaux. Car le bruit introduit par un gradient calculé sur un seul point permettra à la suite qu'on construit d'"échapper" à un extrema local (convergence prématurée) pour trouver une valeur plus optimale.
- Dans la descente de gradient mini-batch le calcul du gradient se fait sur une partie des données d'apprentissage. Cette dernière méthode permet de palier au bruit trop important introduit par le calcul du gradient sur un seul exemple dans le SGD. Une bonne valeur de la taille du batch est assez petite pour permettre d'échapper à des extrema locaux de mauvaise qualité et assez grande pour permettre de converger vers des extrema globaux ou des extrema locaux de bonne qualité.

2.2 Régression avec un réseau de neurones à deux couches :

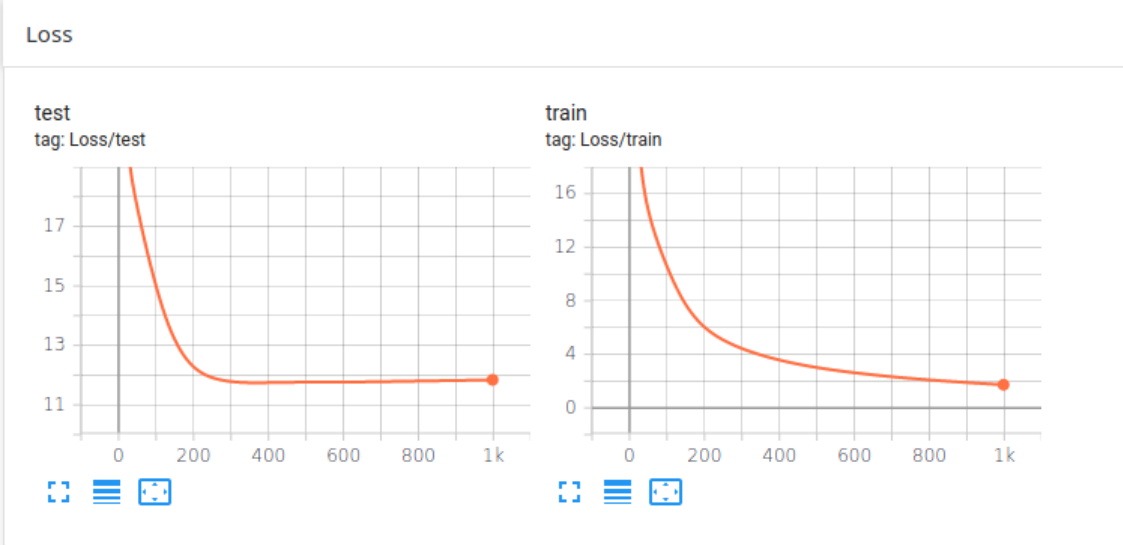
Dans cette partie on va apprendre un réseau de neurones à deux couches pour la tache de régression sur les données boston Housing. L'architecture utilisé est la suivante :

$$Linéaire \longrightarrow tanh \longrightarrow Linéaire \longrightarrow MSE$$

- Données : Boston Housing, 33% en test et 67% en train. Les features sont centrées et réduites (moyennes et écarts-types calculés sur le train).
- Hyper-paramètres :
  - nombre de neurones de la couche cachée : 100.
  - batch size : 15.
  - nombre d'époques : 1000.
  - optimiseur : **SGD** avec learning rate= 0.001.
- On affiche le coût sur le train set et le test set à la fin de chaque époque.

Courbe du coût en apprentissage et celle en test - **Descente de gradient mini-batch taille 15**

–NN 2 couches



Remarques : Cette architecture fait beaucoup mieux que le modèle linéaire précèdent (avec les mêmes hyper-paramètres). Le coût descend jusqu'à $\approx 1,72$ en apprentissage. Au bout de ≈ 330 itérations on atteint en test un coût de $\approx 11,76$ (au lien de ≈ 20 avec le modèle linéaire), au delà on fait face à du sur-apprentissage (le coût continue à descendre en apprentissage et remonte jusqu'à $11,84$ en test)

Chapter 3

Auto-encodeur :

Un auto-encodeur est un réseau de neurones qui a pour but d'apprendre une nouvelle représentation des données et ce dans le but d'en réduire la dimension de l'espace de représentation. L'auto-encodeur se compose de deux partie (similaire à la PCA ou on a deux matrices -compression et décompression-) :

- encodage : $\phi : \mathcal{X} \to \mathcal{F}$
- décodage : $\psi : \mathcal{F} \to \mathcal{X}$

En minimisant le risque empirique : $argmin_{\phi,\psi} \|X - \psi \circ \phi(X)\|$
Dans ce qui va suivre, nous allons apprendre un auto-encodeur sur la base de données MNIST, l'architecture de ce dernier est la suivante :

- Encodage : $Linéaire \longrightarrow ReLU$
- Décodage : $Linéaire \longrightarrow sigmoïde$

De plus les poids de l'auto-encodeur sont liés, on utilise la transposée des poids de l'encodeur comme poids du décodeur.

- Encodage : $z = ReLU(Wx + b) : \mathbb{R}^d \to \mathbb{R}^p$
- Décodage : $x^{'} = sigmoïde(W^t z + b^{'}) : \mathbb{R}^p \to \mathbb{R}^d$

Apprendre un "tied auto-encoder" est motivé par :

- Un apprentissage plus rapide.
- Agit comme une méthode de régularisation.
- Dans notre cas -le cas linéaire- l'auto-encodeur est très similaire à la PCA (où on peut prouver que s'il existe une matrice de compression et une matrice de décompression qui minimisent le risque théorique, alors il existe aussi deux matrice orthogonales telles que l'une est la transposée de l'autre et qui minimisent aussi ce même risque), mais même avec cette similarité architecturale l'auto-encodeur obtenu ne partagent pas toutes les propriétés de la PCA, par exemple l'orthogonalité.

3.1 Expérimentations :

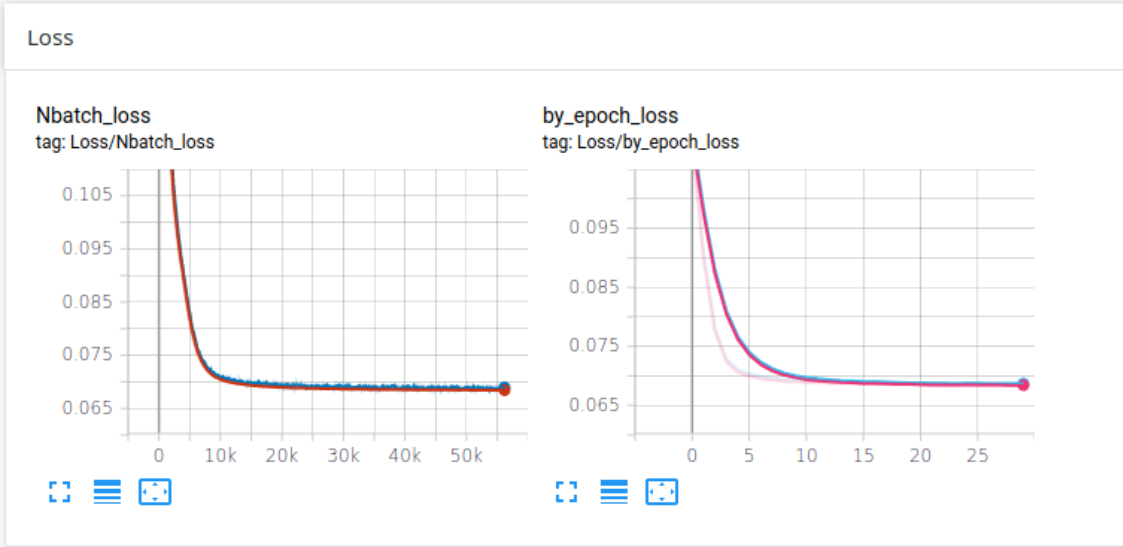
- Données : base MNIST (images 28×28 en niveaux de gris).
- coût : cross-entropy.

- Hyper-paramètres :
  - Dimension de l'espace de départ : $28 \times 28 = 784$.
  - Dimension de l'espace de projection : 100.
  - batch size : 32.
  - nombre d'époques : 30.
  - optimiseur : **ADAM** avec learning rate= 0.001.
- On affiche le coût sur le train set et le test set à la fin de chaque époque, on affiche aussi le coût au bout de chaque 100 itérations (passage sur 100 batchs).
- On affiche la reconstruction des images (résultat de la partie "decoding") après chaque epoch.

Courbes du coût :

Courbes du coût en apprentissage et celle en test - **autoencodeur - dimesion de l'espace de projection = 100**

– par époque et par 100 itérations

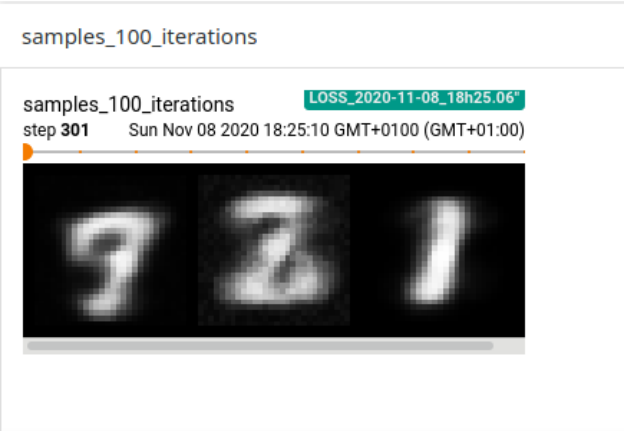


Remarques : Le modèle converge au bout de ≈ 9 itérations.

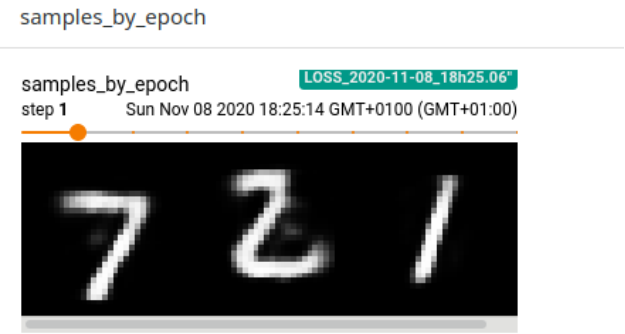
Reconstruction :

Reconstruction des images - **epace de projection = 100 — Sans apprentissage**

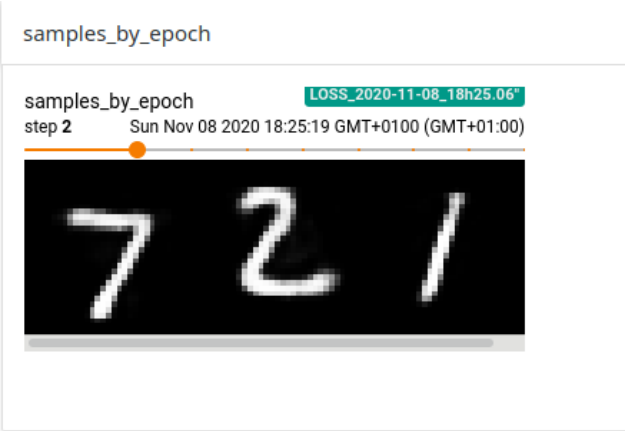
Reconstruction des images - **espace de projection = 100 — Après 300 itérations de la première époque**



Reconstruction des images - **espace de projection = 100 — Après 1 époque**



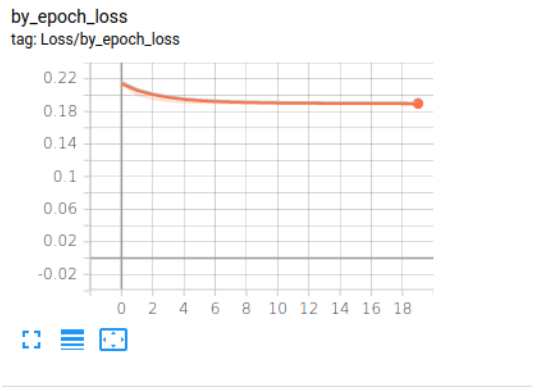
Reconstruction des images - **espace de projection = 100 — Après 2 époques**



Remarques : Sans apprentissage le décodage retourne un bruit. Après 300 itérations de la première époque, on commence à distinguer la forme de chiffres. Après seulement 1 époque on obtient une reconstruction très satisfaisante.

3.1.1 Comparaison en fonction de la dimension de l'espace de projection :

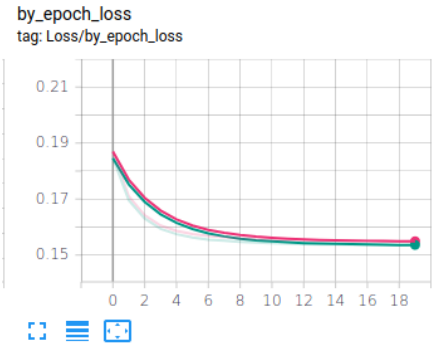
Dans cette partie on va faire varier la dimension de l'espace de projection $p = 5, 10, 40, 80$ et comparer la qualité des reconstructions obtenues :



(a) Courbes du coût en apprentissage et celle en test

(b) Reconstruction des images à la fin de l'apprentissage

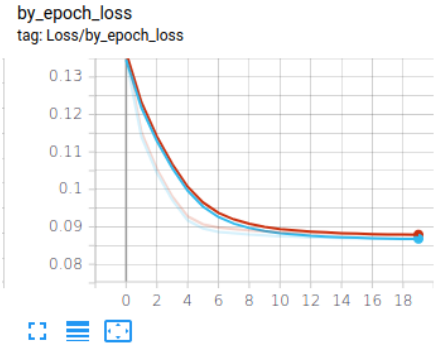
Dimension de l'espace de projection = 5



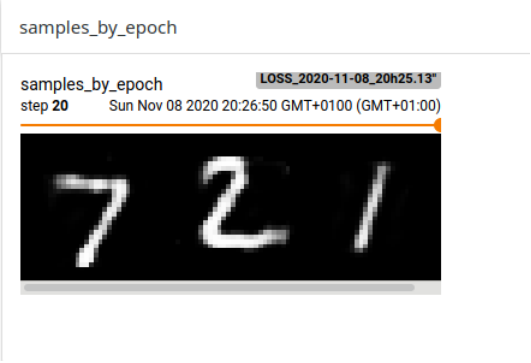
(c) Courbes du coût en apprentissage et celle en test

(d) Reconstruction des images à la fin de l'apprentissage

Dimension de l'espace de projection = 10

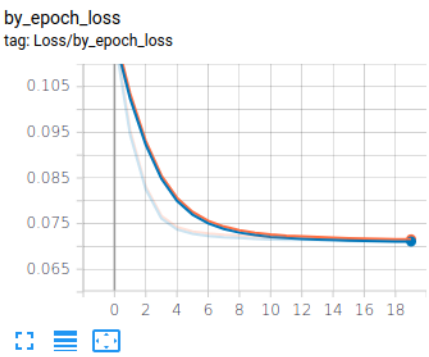


(e) Courbes du coût en apprentissage et celle en test

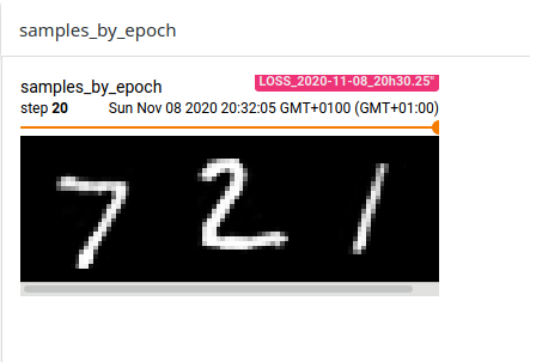


(f) Reconstruction des images à la fin de l'apprentissage

Dimension de l'espace de projection = 40



(g) Courbes du coût en apprentissage et celle en test



(h) Reconstruction des images à la fin de l'apprentissage

Dimension de l'espace de projection = 80

Remarques :

- Pour $d = 5$, on converge vers ≈ 0.19 après ≈ 8 époques. La dimension de l'espace de compression est beaucoup trop petite, la perte d'information est beaucoup trop importante et ne permet de reconstruire les images.
- Pour $d = 10$, on converge vers ≈ 0.153 après ≈ 12 époques. La dimension de l'espace de compression est trop petite, la perte d'information est trop importante et ne permet de reconstruire des images claires, on distingue cependant la forme des chiffres.
- Pour $d = 40$, on converge vers ≈ 0.0875 après ≈ 14 époques. La dimension de l'espace de compression est suffisamment grande pour permettre une reconstruction acceptable, on distingue les chiffres clairement.
- Pour $d = 80$, on converge vers ≈ 0.071 après ≈ 17 époques. La dimension de l'espace de compression est suffisamment grande pour permettre une reconstruction de bonne qualité, on distingue les chiffres clairement.

Comparaison : Plus la taille de l'espace de projection est grande mieux est la reconstruction (et parallèlement plus le risque empirique est petit). Projeter sur des espaces de représentation de dimensions trop petites engendre une perte d'information trop importante, ce qui est attendu. D'autre part, plus la dimension de l'espace de projection est grande plus l'apprentissage nécessite du temps, car on a plus de paramètres à apprendre (la matrice W est de taille plus grande).

Chapter 4

HighWay network :

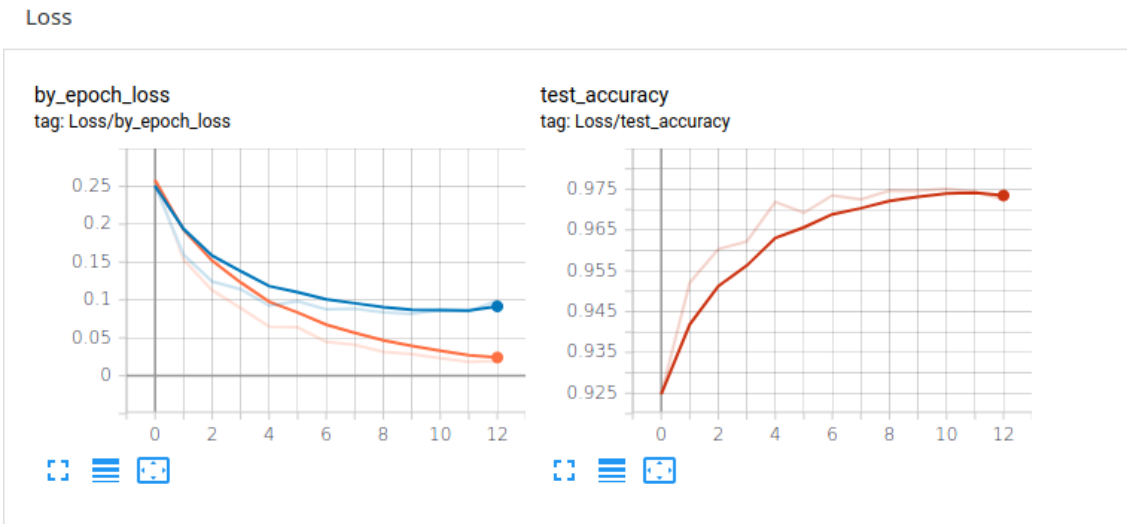
Dans cette partie on va implémenter un HighWay network tel que décrit dans l'article suivant **HighWay network**. Nous allons entraîner notre HighWay network sur la base de données MNIST pour la tache de classification. L'architecture est résumée ci-dessous :

- Une première couche linéaire fully connected.
- Un nombre "L" de couches HighWay definies comme suit :
  - $y = H(x, W_H).T(x, W_T) + x.(1 - T(x, W_T))$
  - Fonction d'activation pour T : sigmoïde.
  - Fonction d'activation pour H : ReLU ou tanh.
- Une dernière couche linéaire qui projette vers un espace de dimension d égale au nombre de classe que nous avons (10 dans notre cas).

4.1 Expérimentations :

- Données : base MNIST (images 28×28 en niveaux de gris).
- coût : cross-entropy.
- Hyper-paramètres :
  - Dimension de l'espace de départ : $28 \times 28 = 784$.
  - Initialisation du bais pour T à -3.
  - Nombre de couche HighWay $L = 9, 19$.
  - Number of units in each layer : 50.
  - batch size : 32.
  - nombre d'époques : 20.
  - optimiseur : **SGD** avec learning rate= 0.001 et momentum = 0.9.
- On affiche le coût sur le train set et le test set à la fin de chaque époque, on affiche aussi l'accuracy sur le test à chaque époque.

Courbe du coût en apprentissage et celle en test - Pourcentage de bonne classification sur l'ensemble de test - **L = 9**



Courbe du coût en apprentissage et celle en test - Pourcentage de bonne classification sur l'ensemble de test - **L = 19**

Remarques :

- L = 9 : On remarque qu'au bout de 11 époques, le coût en test (en bleu) atteint son minimum (≈ 0.08), au delà on fait face à du sur-apprentissage (le coût en train (en orange) continue à baisser et le coût en test (en bleu) remonte). Parallèlement pour l'accuracy, on atteint $97,5\%$ de bonnes classifications au bout de 11 itérations, au delà l'accuracy décroît. Par ailleurs, une seule époque est suffisante pour avoir une accuracy de $92,5\%$.
- L = 19 : le comportement des deux réseaux est similaire. On obtient de meilleurs résultat avec $L = 19$ (accuracy plus grande et coût plus petit).