Jouve Vincent 3671083
Lenoir Romain 3670199

# TP1:

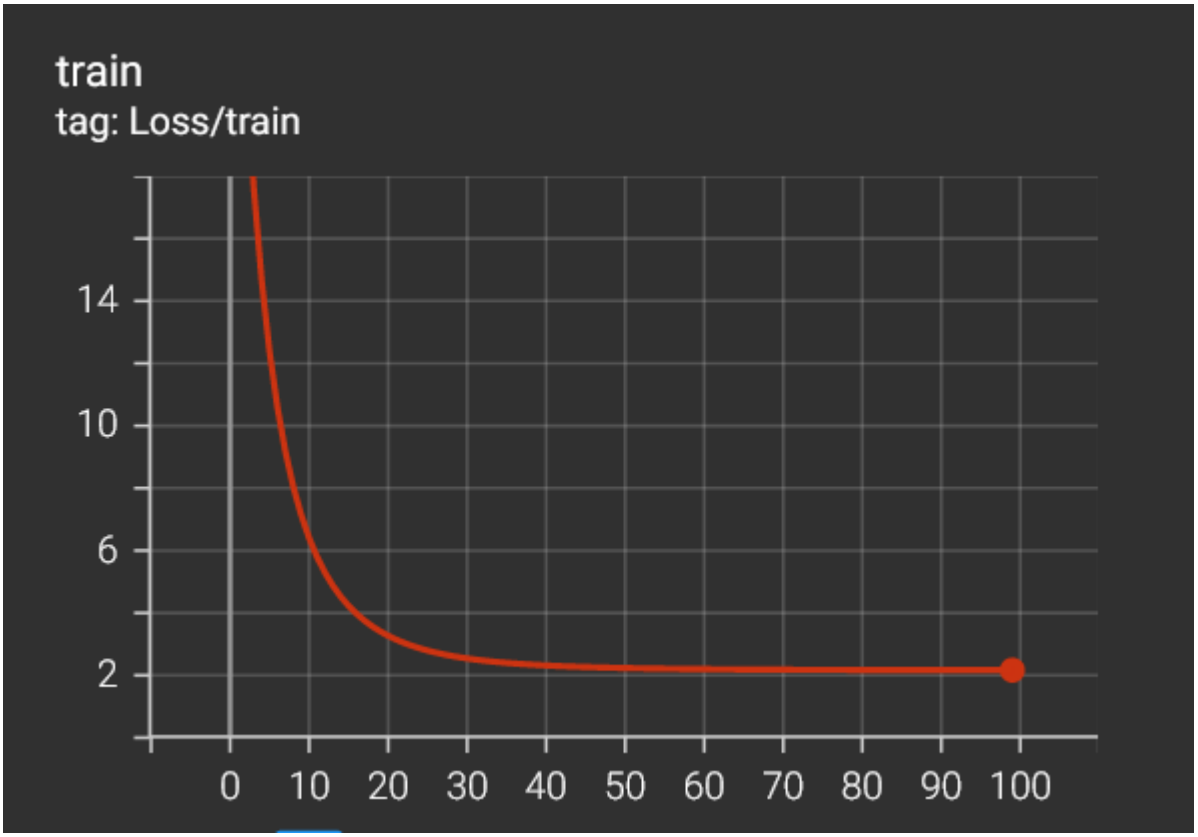
D'abord nous avons codé le module MSE. Pour le forward c'est assez simple : mean(( yhat - y) **2)
Pour le backward il faut retourner la dérivée par rapport à yhat et par rapport à y.
Nous avons donc 2(yhat-y) et -2(yhat-y) qu'il faut multiplier au grad_output histoire de conserver la récursivité de la backpropagation

Ensuite il a fallu faire Linear. Pour le forward linear est une fonction lineaire : yhat = x@w +b avec x les donnees, w le poids des neuronnes de la couche et b le biais
Pour le backward il y a donc 3 dérivées: d(yhat)/d(x) = w, d(yhat)/d(w) = x, d(yhat)/d(b)= 1
Il faut donc retourner les 3 dérivées multiplié au grad_output ce qui donne ainsi:

```
grad_output@w.t(), x.t()@grad_output, torch.sum(grad_output,axis=0)
```

la somme permet de conserver les bonnes dimensions car b est un vecteur colonne.

Il ne reste plus qu'à chaque époque de forwarder le linéaire puis la loss, de faire le backward et enfin de mettre à jour les paramètres.

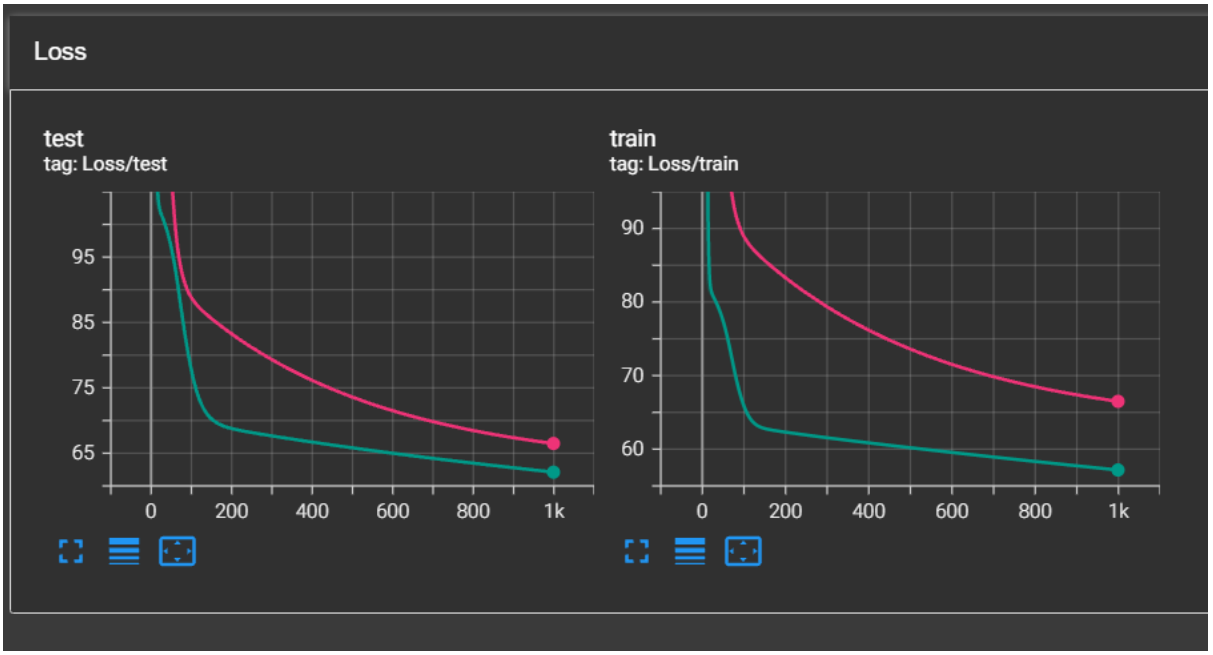



Ici, la courbe de la loss en train, au fil des itérations, avec une couche de 3 linéaires sur des données générées aléatoirement. La loss converge comme prévu.

# TP2:

Pour commencer nous avons fait un réseau avec une seule couche linéaire d'un seul neurone et une mse loss. Nous l'avons fait à l'aide de tensor torch. Il fallait donc mettre à True le paramètre requires_grad pour que le calcul de gradient se fasse à l'aide de la méthode .backward().

Ensuite nous avons codé un réseau de neurones avec 2 couches linéaires et une couche tangente hyperbolique. La première couche comprend 20 neurones tandis que la 2e seulement 1 seul.

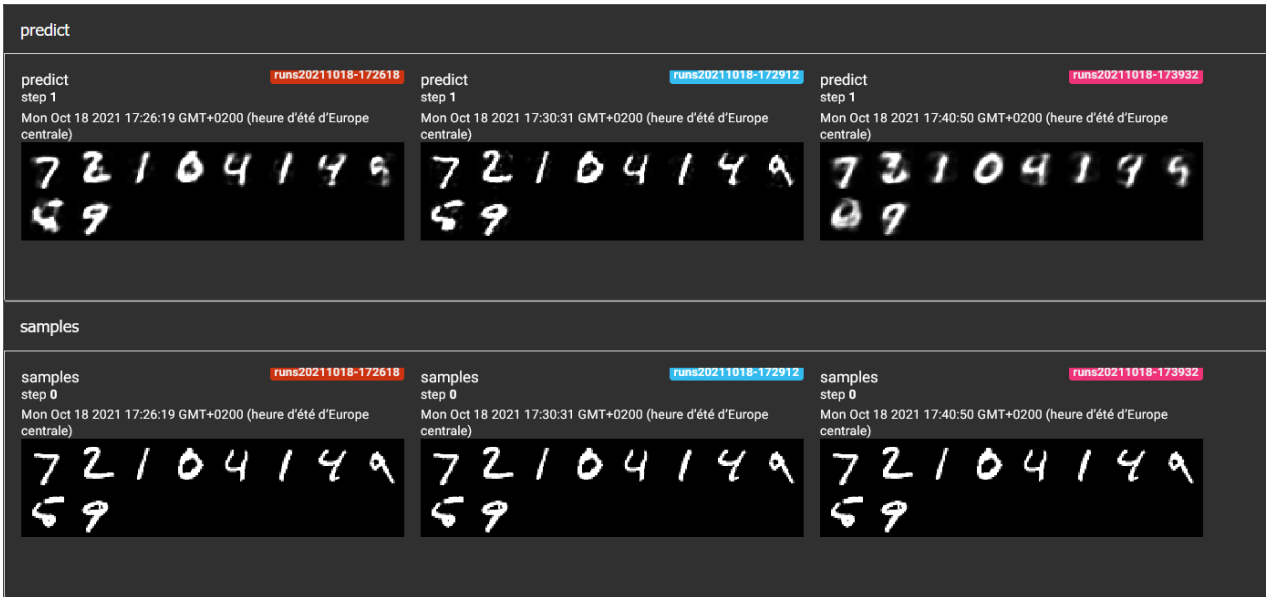



En rose on peut voir la courbe du premier réseau de neurones. En vert il y a la courbe du second. On peut voir qu'en test la loss ne remonte pas. Il n'y a donc aucun signe de surapprentissage.

# TP3 :

Après avoir normalisé les pixels des images, et mis sur une dimension pour passer l'image au réseau de neurones. On a codé l'encodeur de la façon suivante, Linear -> sigmoid(pour remettre les données entre 0 et 1)->Linear(transposé du premier avec un nouveau biais)->sigmoid pour remettre comme au format de l'entrée.

On a testé 3 compressions, une de taille 30(à gauche) et une autre de taille 100(au milieu) et 10(à droite).




En haut(predict) c'est l'image encodée puis décodée et en bas(sample) c'est l'image de base.
On voit que ça fonctionne plutôt bien, on reconnaît bien le chiffre de base, pour la compression de taille 30 et 100. On voit aussi qu'avec la compression de taille 30 le décodage est plus compliqué, on a perdu un peu d'informations. Par contre la compression de taille 10 on ne reconnait plus certains chiffres, on a trop compressé et on a perdu trop d'informations.

Ici nous avons mélangé l'embedding d'un 2 et d'un 7 pour lambda de 0.1 à 0.9(de gauche à droite) entre lambda 0.4 et 0.6 on n'a réellement une sorte de mélange des deux chiffres.


Puis nous avons testé un réseau à 3 couches, 1er couche avec une dimension de sortie de 500, puis 100, puis 30.

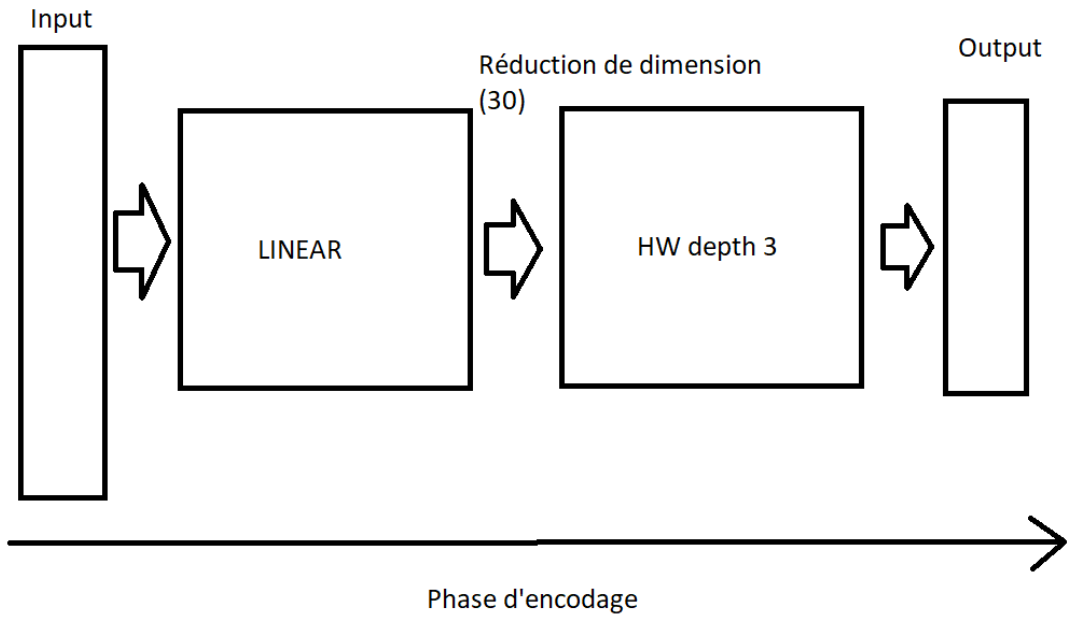

Ici on compare,sur quelques exemples, le réseau avec une seule couche en dimension 30(à droite) avec le réseau multicouche présenté ci-dessus(à gauche). On voit une petite amélioration pour le réseau multicouche, alors qu'au final nous avons compressé l'information en dimension 30 dans les 2 cas.

| | train loss | test loss |
|---|---|---|
| NN (1 couche) | 0,0236 | 0,0234 |
| NN (3 couches) | 0,0127 | 0,0131 |

L'amélioration est confirmée en regardant les loss en train et en test.

## Highway

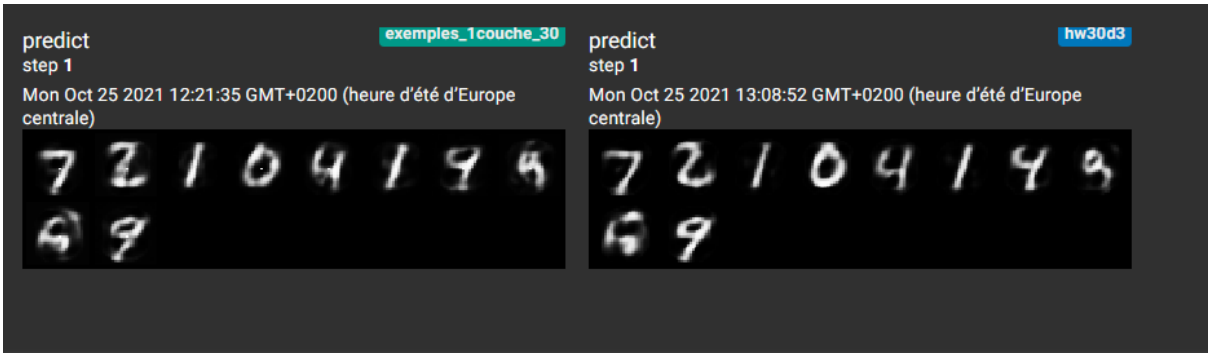
On a construit un réseau de cette forme.



Pour comparer avec le réseau à 1 couche caché en dimension 30.

| | train loss | test loss |
|---|---|---|
| NN (1 couche) | 0,0236 | 0,0234 |
| HW | 0,0175 | 0,0173 |

Et voilà ce que ça donne sur quelques exemples.




à gauche le réseau simple à une couche et à droite le HW. On voit une petite amélioration visuelle comme prévue par la loss.