

IFT3395 - Fondements de l'apprentissage machine

Travail Pratique 3

Nom: Paul CHAFFANET
Matricule: 1009543

Nom: Émile Labbé
Matricule: 20019813

I. PARTIE PRATIQUE: implémentation du réseau de neurones.

Question 1

Dans un premier temps, commencez par une implémentation qui calcule le gradient pour un exemple, et vérifiez que le calcul est correct avec la technique de vérification du gradient par différence finie expliquée ci-dessus.

Voir la méthode check_grad(data, k) définie dans Main.py à la ligne 29. La méthode check_grad calcule alors, sur un réseau de neurones mlp, le gradient pour un exemple choisi aléatoirement dans les données des deux lunes. Notre méthode fait donc appel aux méthodes mlp.compute_grad(batch) et mlp.compute_finite_difference(batch) (définie dans la classe Mlp du fichier Mlp.py) qui calcule le gradient avec les paramètres actuels de mlp avec l'unique exemple x contenu dans batch.

Question 2

Vérification du gradient: produire un affichage de vérification du gradient par différence finie pour votre réseau (pour un petit réseau, par ex. $d = 2$ et $d_h = 2$ initialisé aléatoirement) sur 1 exemple.

Ratios W1

```
-----  
[[ 1.      1.      ]  
 [ 1.00000004  0.99999948]]
```

Ratios b1

```
-----  
[[ 1.      ]  
 [ 1.00000046]]
```

Ratios W2

```
-----  
[[ 1.      0.99999858]  
 [ 1.      1.00000142]]
```

Ratios b2

```
[[ 0.99999339]
 [ 1.00000661]]
```

Vérification du gradient réussie pour K = 1

Question 3

Ajoutez à cette version un hyperparamètre de taille de lot K, pour permettre le calcul du gradient par mini-lot de K exemples (présentés sous forme de matrices), en faisant une boucle sur les K exemples (c'est une petit ajout à votre code précédent).

Voir les mêmes méthodes que citées dans la question 1.

Question 4

Vérification du gradient: produire un affichage de vérification du gradient sur les paramètres, par différence finie pour votre réseau (pour un petit réseau, par ex. d = 2 et dh = 2 initialisé aléatoirement) pour un lot de 10 exemples (vous pouvez prendre des exemples des deux classes du jeu de donnée des 2 lunes).

Ratios W1

```
[[ 1.00000035  0.99999963]
 [ 1.00000016  0.99999832]]
```

Ratios b1

```
[[ 1.00000039]
 [ 1.00000022]]
```

Ratios W2

```
[[ 1.00000043  1.00000116]
 [ 0.99999957  0.99999884]]
```

Ratios b2

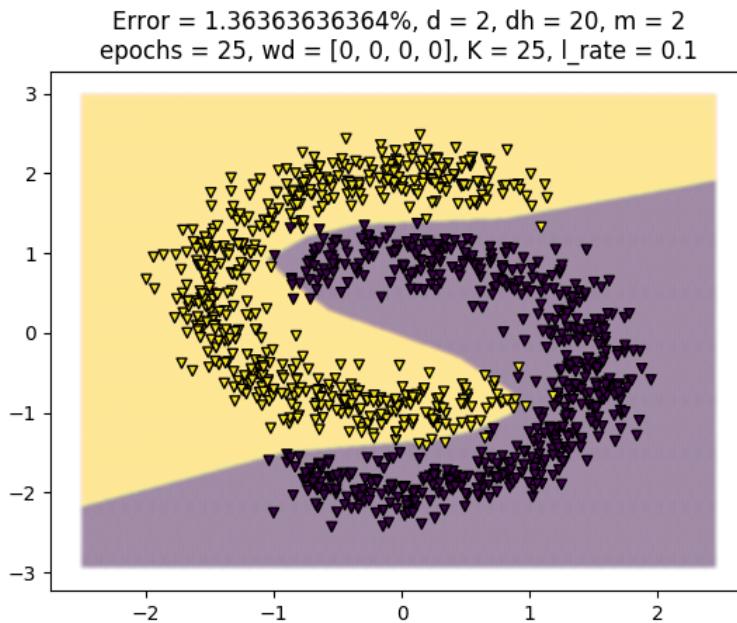
```
[[ 1.00000144]
 [ 0.99999856]]
```

Vérification du gradient réussie pour $K = 10$

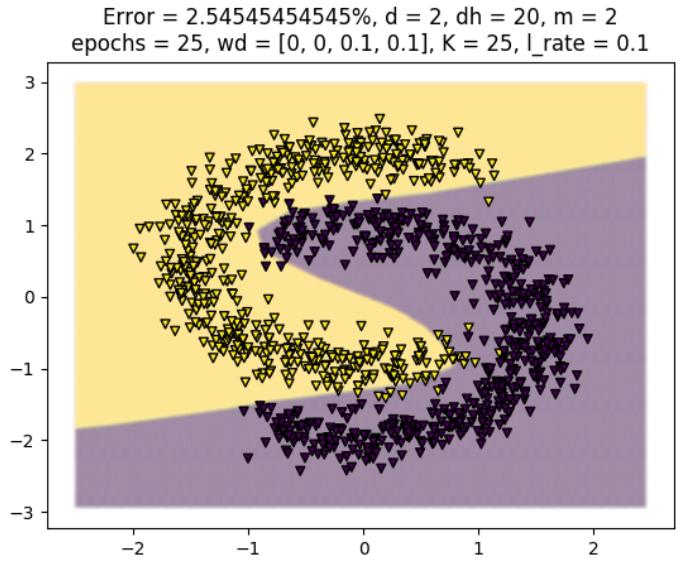
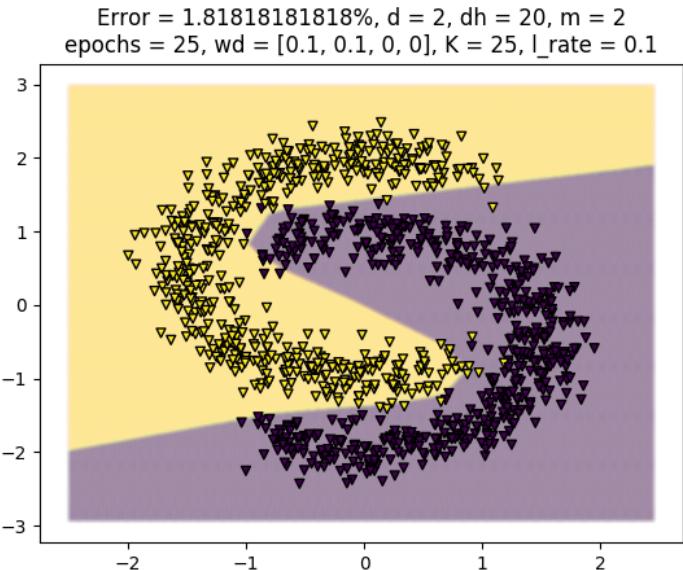
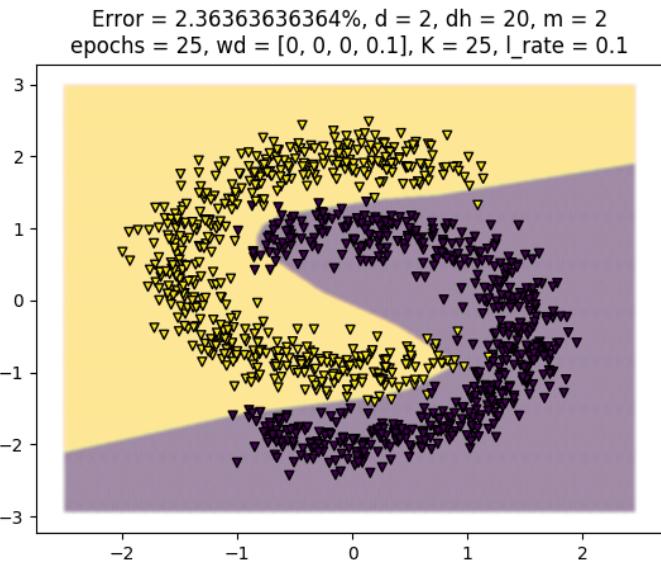
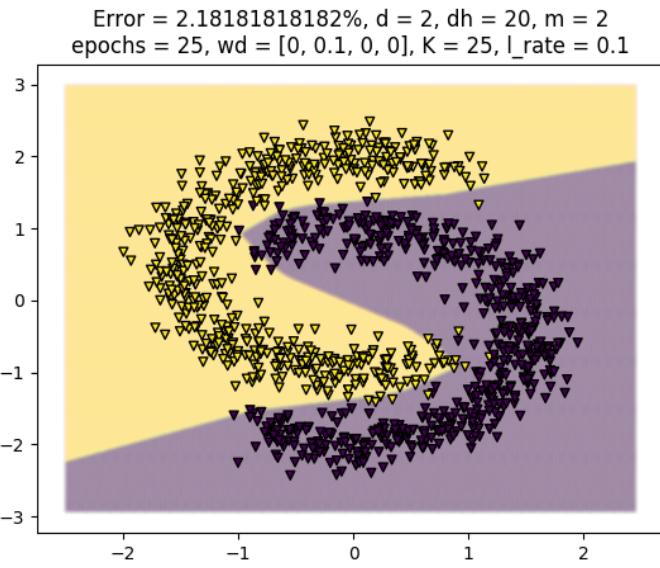
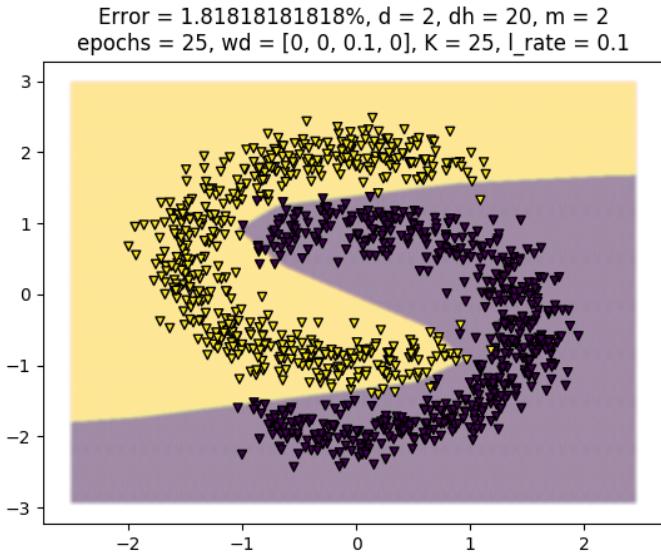
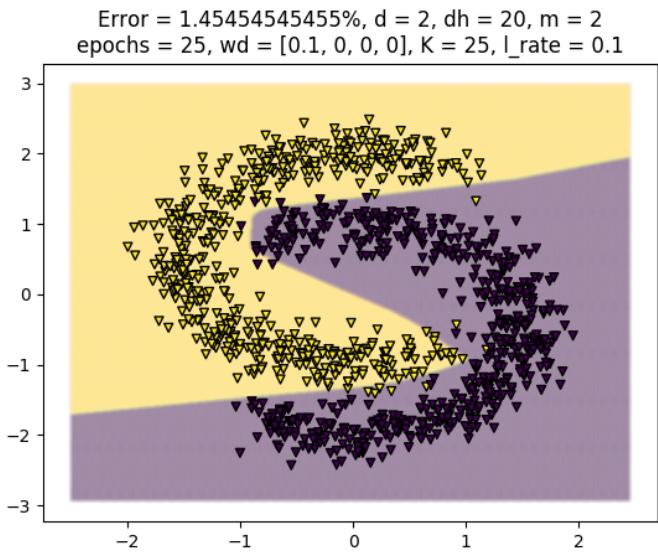
Question 5

Entraîner votre réseau de neurones par descente de gradient sur les données du problème des deux-lunes. Afficher les régions de décision pour différentes valeurs d'hyper-paramètres (weight decay, nombre d'unites cachées, arrêt prématué) de façon à illustrer leur effet sur le contrôle de capacité.

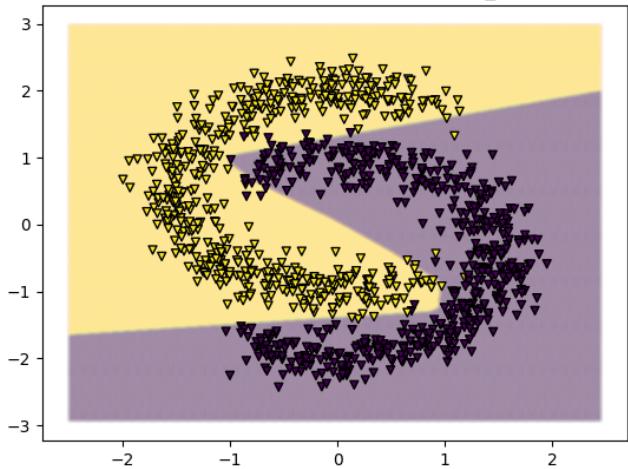
Pour bien comprendre l'influence de chaque paramètres sur le contrôle de capacité, et de manière plus générale, sur l'apprentissage, nous avons commencé par entraîner notre réseau de neurones avec des hyper-paramètres fixés $d = 2$, $d_h = 20$, $m = 2$, $K = 25$ et $epoch = 25$ et nous avons testé l'effet de la modification du weight-decay sur différents paramètres.



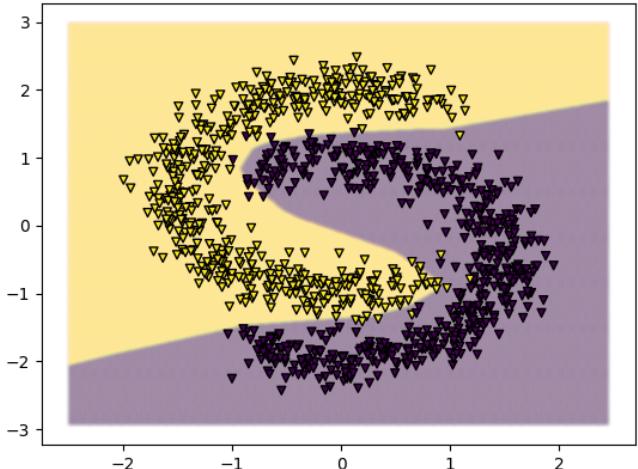
En comparant le graphique de la page 4 avec ceux des pages 5 et pages 6, on peut facilement observer l'influence du weight-decay. Le weight-decay limite l'influence des poids forts et peut donc apporter une meilleur généralisation de notre jeu de donnée. L'erreur d'entraînement est ainsi plus importante, mais peut tendre à minimiser l'erreur de test ce qui est une bonne chose. Le weight-decay permet ainsi de limiter la capacité de notre modèle.



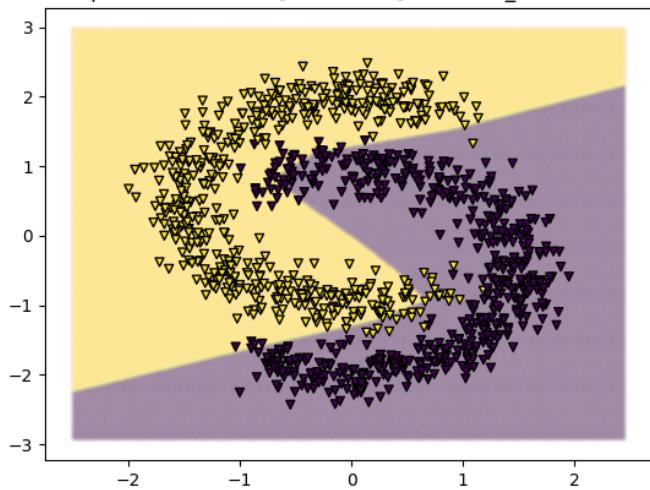
Error = 2.18181818182%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0.5, 0, 0, 0], K = 25, l_rate = 0.1



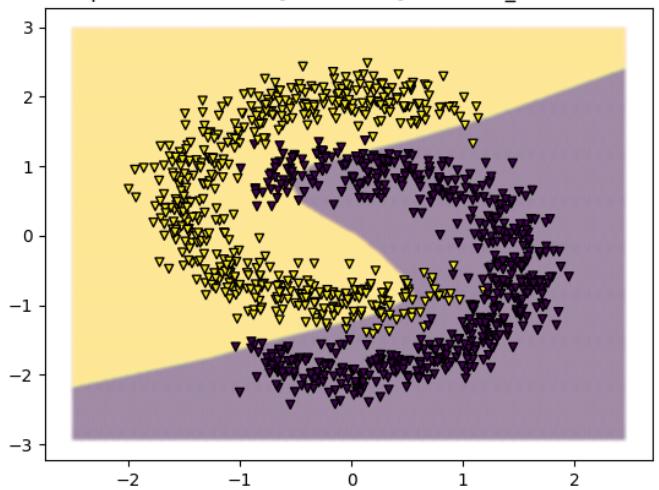
Error = 1.72727272727%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0, 0, 0.5, 0], K = 25, l_rate = 0.1



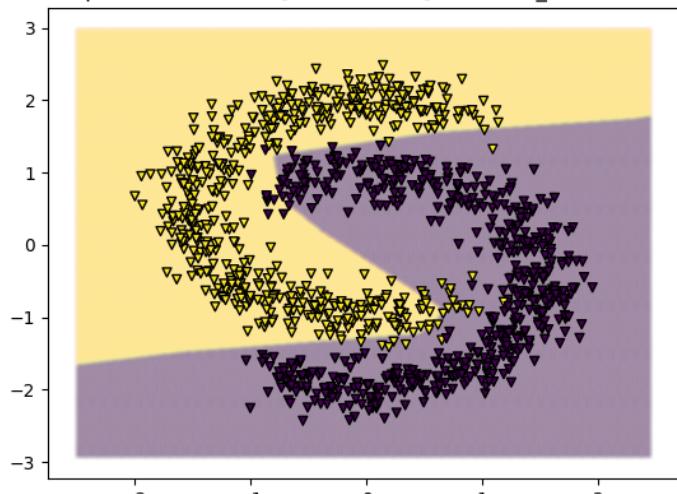
Error = 6.0%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0, 0.5, 0, 0], K = 25, l_rate = 0.1



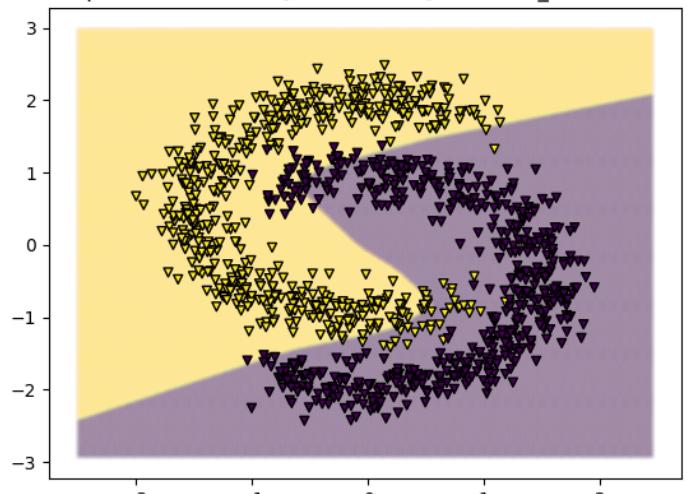
Error = 8.0%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0, 0, 0, 0.5], K = 25, l_rate = 0.1



Error = 3.63636363636%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0.5, 0.5, 0, 0], K = 25, l_rate = 0.1

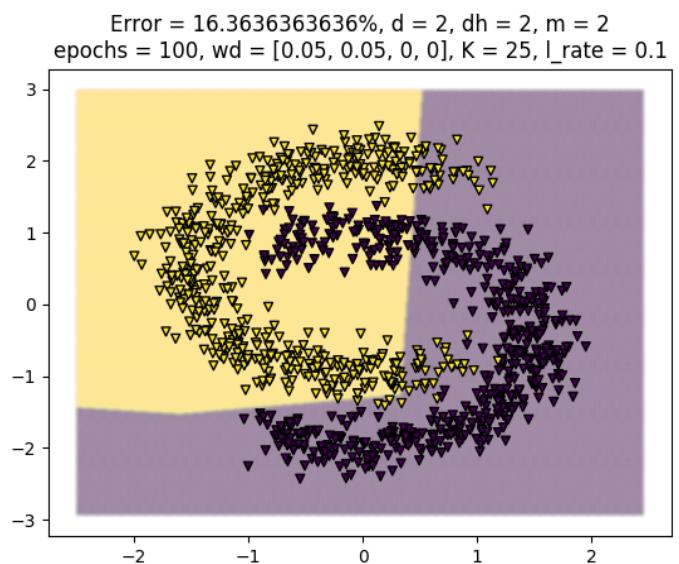
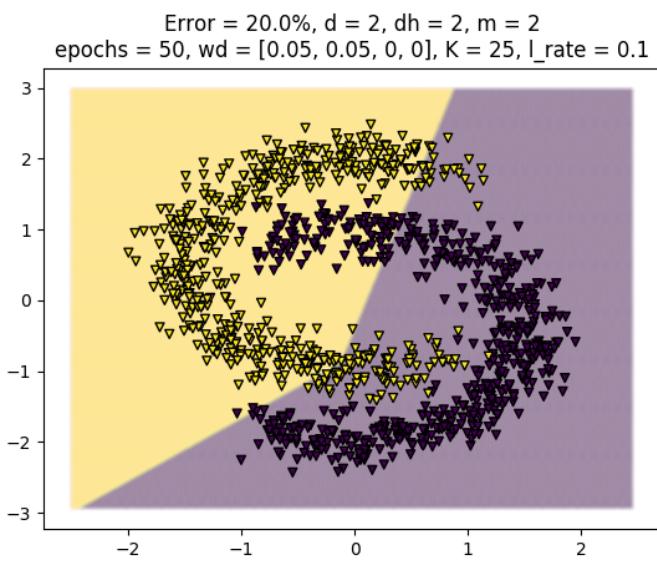
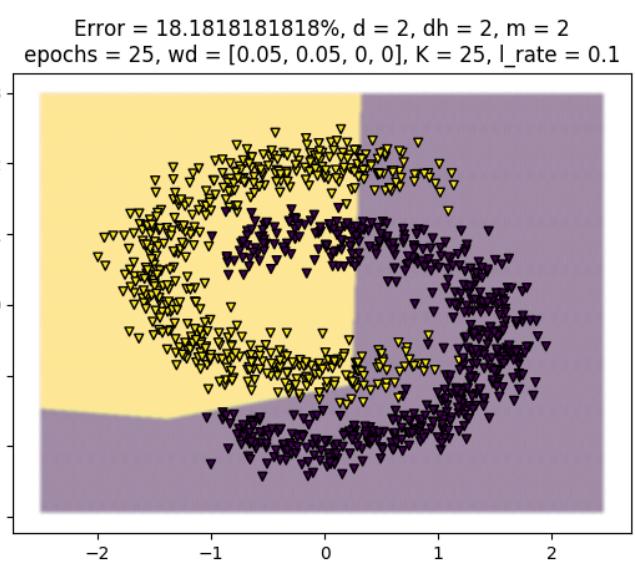
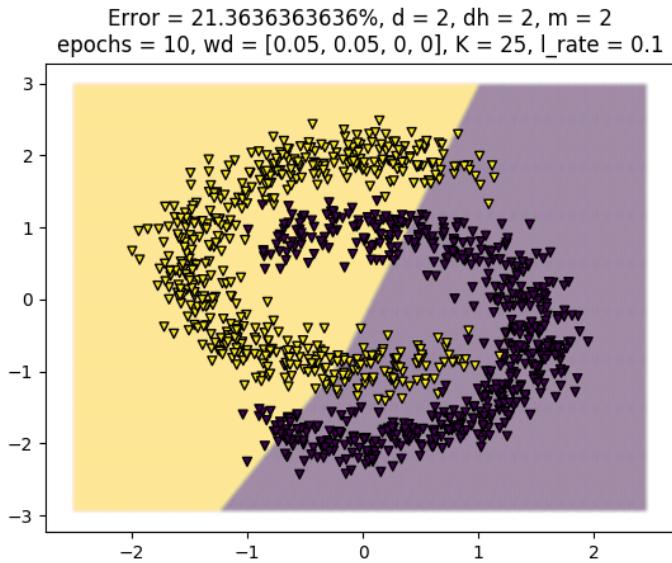


Error = 6.90909090909%, d = 2, dh = 20, m = 2
epochs = 25, wd = [0, 0, 0.5, 0.5], K = 25, l_rate = 0.1

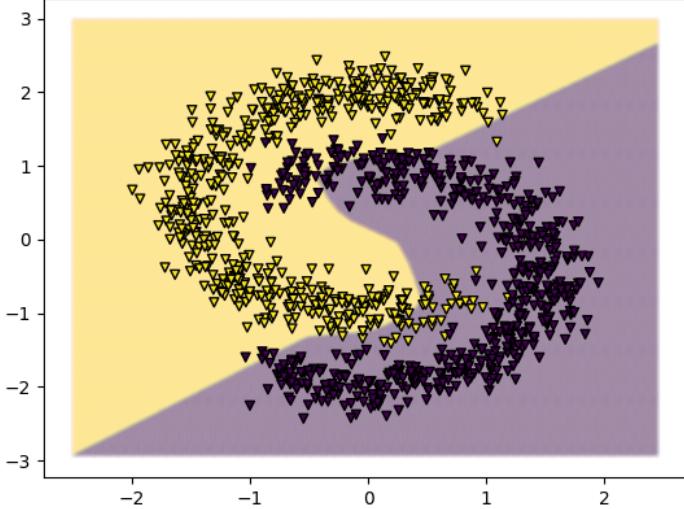


Les graphiques ci-dessous ainsi que les graphiques en page 8 et 9 nous permettent de constater qu'un plus grand nombre de neurones dans la couche cachée permet d'obtenir un meilleur apprentissage en moins d'époques, et permet donc une plus grande capacité de notre modèle. Également, on observe que l'augmentation du nombre neurones finit par ne plus avoir d'influence dans la capacité de notre modèle, il ne servirait donc à rien d'utiliser un trop grand nombre de neurones cachés pour cette tâche.

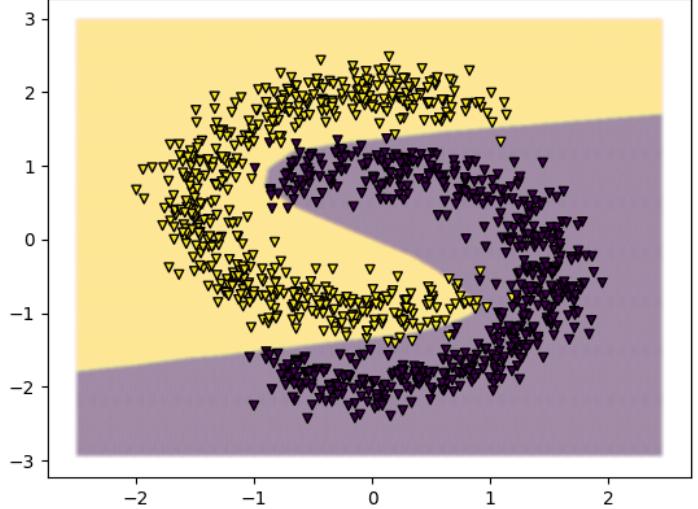
Le nombre d'époques a une influence sur le résultat pour un même nombre de neurones cachés. Plusieurs époques permettent une meilleure optimisation de l'erreur d'entraînement, ce qui permet d'augmenter la capacité de notre modèle.



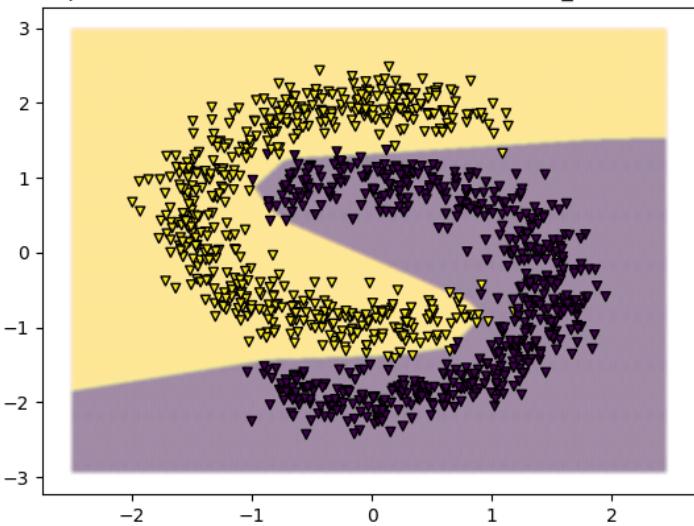
Error = 9.36363636364%, d = 2, dh = 20, m = 2
epoches = 10, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



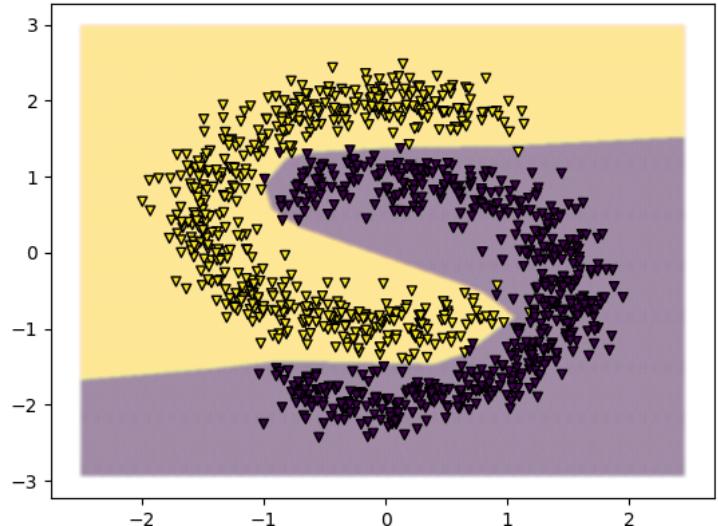
Error = 1.90909090909%, d = 2, dh = 20, m = 2
epoches = 25, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



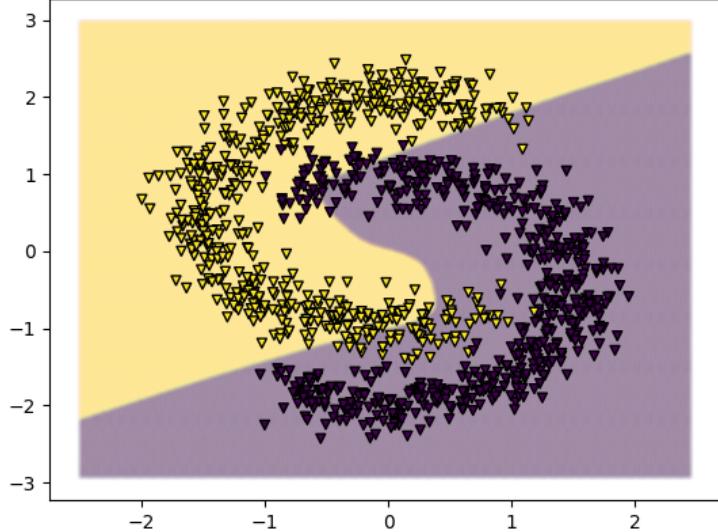
Error = 1.45454545455%, d = 2, dh = 20, m = 2
epoches = 50, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



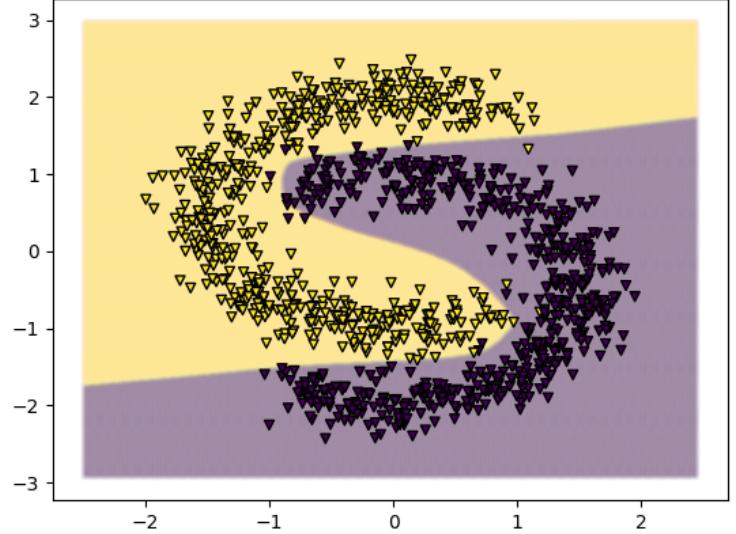
Error = 0.727272727273%, d = 2, dh = 20, m = 2
epoches = 100, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



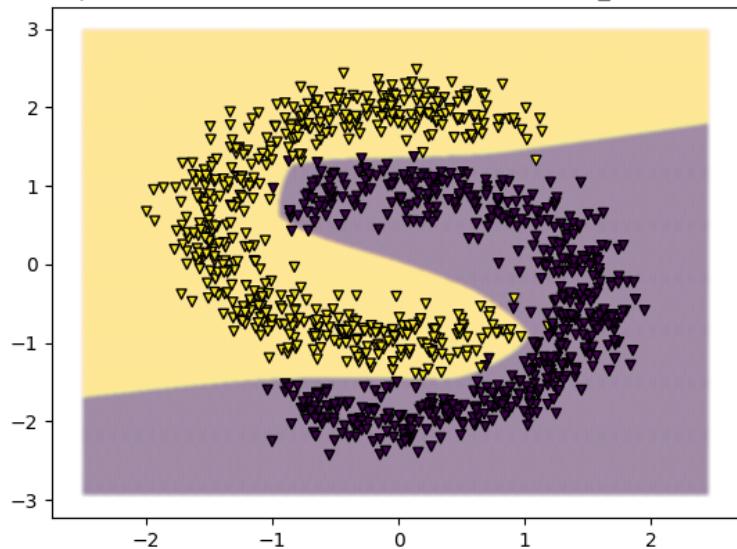
Error = 10.4545454545%, d = 2, dh = 200, m = 2
epoches = 10, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



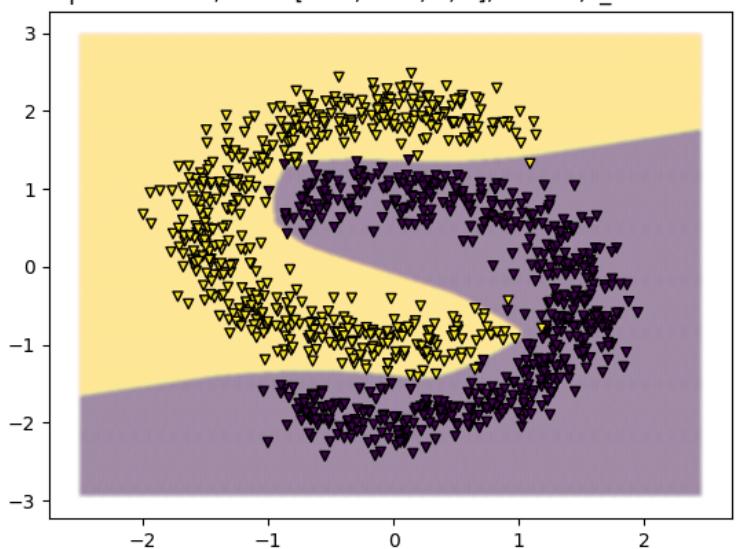
Error = 1.18181818182%, d = 2, dh = 200, m = 2
epoches = 25, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



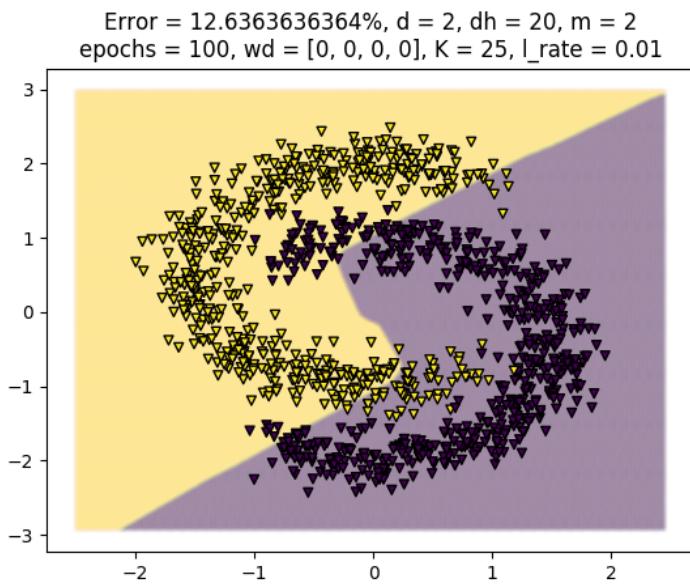
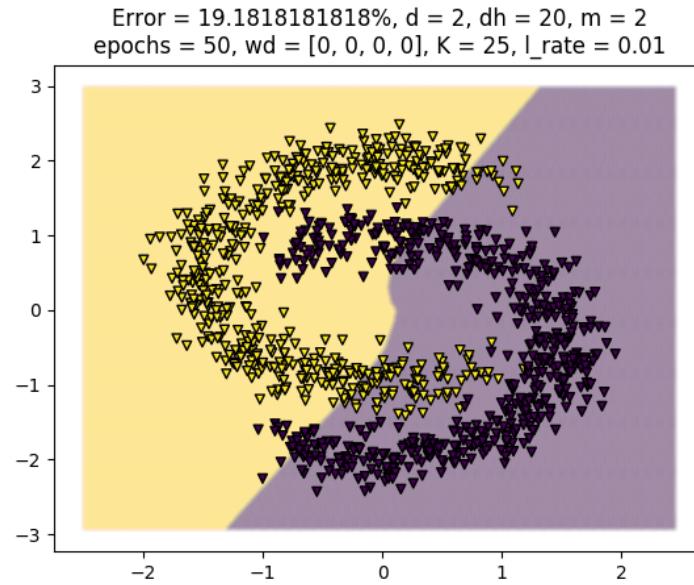
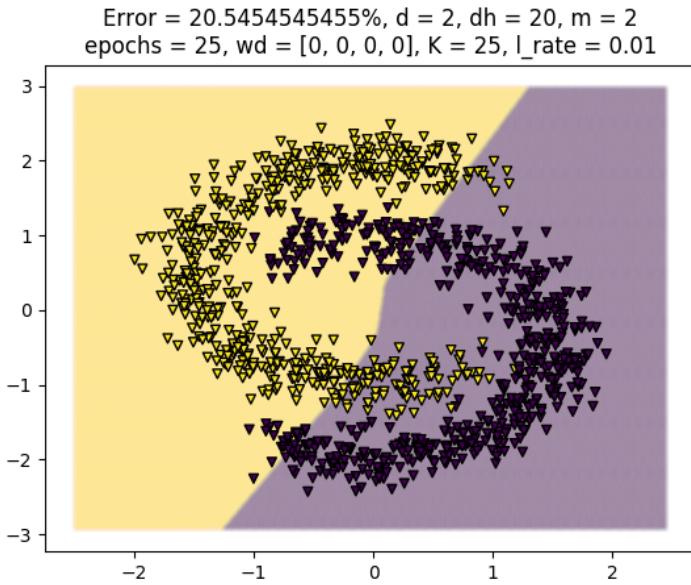
Error = 0.818181818182%, d = 2, dh = 200, m = 2
epoches = 50, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



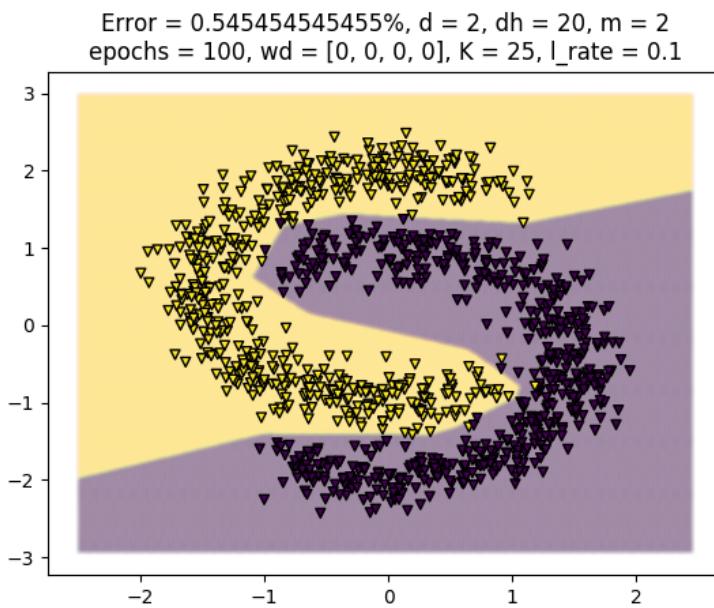
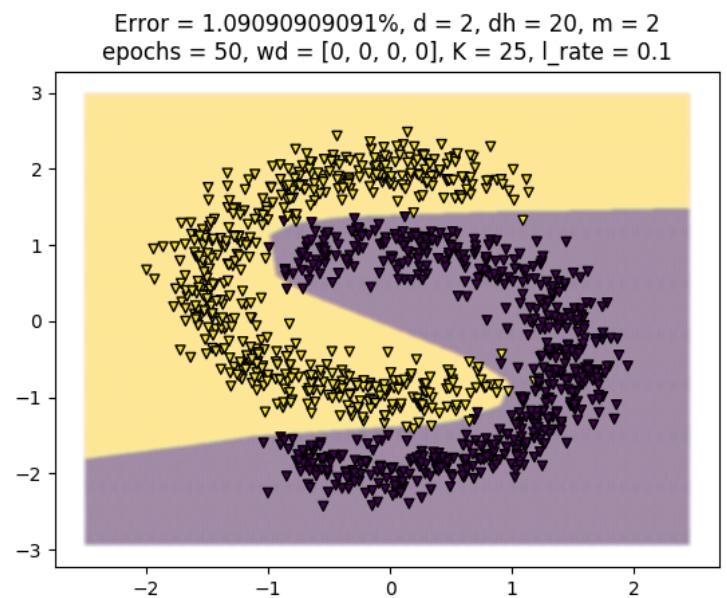
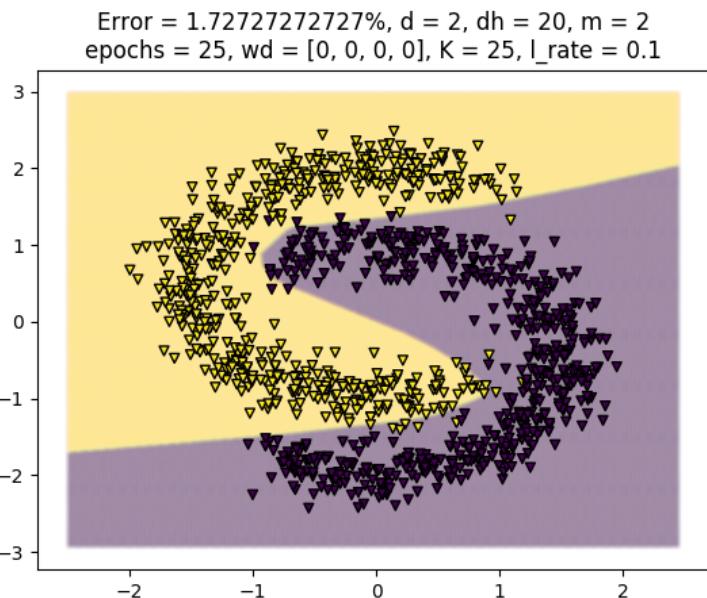
Error = 0.818181818182%, d = 2, dh = 200, m = 2
epoches = 100, wd = [0.05, 0.05, 0, 0], K = 25, l_rate = 0.1



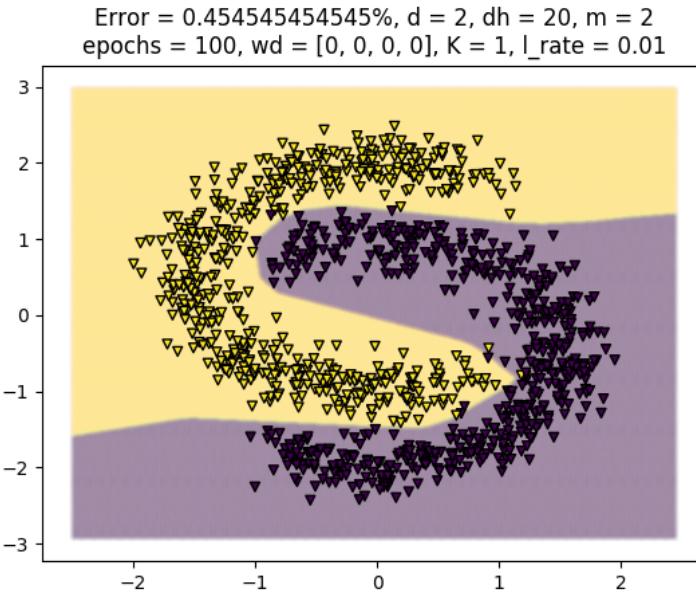
On a également remarqué lors de nos tests que pour un petit learning_rate, plus la taille du mini-batch (hyper-paramètre K) est grande, plus le nombre d'époques nécessaires pour optimiser l'ensemble d'entraînement doit être grand comme nous le montre les trois graphiques ci-dessous.



À l'inverse, pour un grand learning_rate, si K est grand, alors moins d'époques est nécessaires pour arriver à l'optimisation comme nous le montre les trois graphiques ci-dessous comparé aux trois graphiques ci-dessus.



Pour un petit learning_rate, si K petit, alors un grand nombre d'époques apporte une excellente optimisation avec risque de sur-apprentissage comme on peut le constater dans le graphique ci-dessous:



Pour un learning_grand, si K petit, alors il faut un petit nombre d'époques pour arriver à l'optimisation, mais celle-ci ne sera pas forcément précise étant donné la taille du pas.

Question 6

Dans un deuxième temps, faites une copie de votre implémentation en vue d'en faire une version modifiée efficace qui manipulera les lots de K exemples avec des calculs matriciels (plutôt qu'une boucle). Reprenez les expressions matricielles numpy établies dans la première partie, et adaptez-les au cas des mini-lots de taille K. Indiquez dans votre rapport comment vous les avez adaptées (précisez les anciennes et nouvelles expressions avec les dimensions de chaque matrice).

En reprenant les expressions numpy de la premier partie, nous avons établie que :

```
ha = np.dot(W1, x) + b1
hs = rect(ha)
oa = np.dot(W2, hs) + b2
os = softmax(oa)
```

```

grad_oa = os - numpy.array([0 if i != y - 1 else 1 for i in range(m)]).reshape(m,1)
grad_b2 = grad_oa
grad_W2 = np.dot(grad_oa, np.transpose(hs))
grad_hs = np.dot(np.transpose(W2), grad_oa)
grad_ha = grad_hs * numpy.where(ha > 0, [1], [0])
grad_b1 = grad_ha
grad_W1 = np.dot(grad_ha, np.transpose(x))

```

En les adaptant pour une utilisation matricielle, nous avons que:

$W^{(1)}$ est de dimension $d_h \times d$, x est de dimension $d \times n$ et $\text{np.repeat}(b1, n, \text{axis}=1)$ est de dimension $d_h \times n$, alors h^a de dimension $d_h \times n$:

$ha = \text{np.dot}(W1, x) + \text{np.repeat}(b1, n, \text{axis}=1)$

h^a de dimension $d_h \times n$, alors h^s est de dimension $d_h \times n$:

$hs = \text{rect}(ha)$

$W^{(2)}$ est de dimension $m \times d_h$, x est de dimension $d \times n$ et $\text{np.repeat}(b2, n, \text{axis}=1)$ est de dimension $m \times n$, alors o^a de dimension $m \times n$:

$oa = \text{np.dot}(W2, hs) + \text{np.repeat}(b2, n, \text{axis}=1)$

o^a de dimension $m \times n$, alors o^s est de dimension $m \times n$:

$os = \text{softmax}(oa)$

o^s est de dimension $m \times n$, y est de dimension $n \times 1$. La fonction onehot retourne une matrice de dimension $n \times m$ représentant une matrice d'encodage onehot pour toutes les positions y . Donc $grad_{oa}$ est de dimension $m \times n$:

$grad_{oa} = os - \text{np.transpose}(\text{onehot}(\text{np.transpose}(y), m))$

$grad_{b2}$ est de dimension $m \times 1$:

$grad_{b2} = \text{np.sum}(grad_{oa}, \text{axis}=1)$

$grad_{oa}$ est de dimension $m \times n$ et h^s est de dimension $d_h \times n$, alors $grad_{W2}$ est de dimension $m \times d_h$:

$grad_{W2} = \text{np.dot}(grad_{oa}, \text{np.transpose}(hs))$

`grad_W2` est de dimension $m \times d_h$ et `grad_oa` est de dimension $m \times n$, alors `grad_hs` est de dimension $d_h \times n$:

```
grad_hs = np.dot(np.transpose(W2), grad_oa)
```

`grad_hs` est de dimension $d_h \times n$, et `np.where(ha > 0, [1], [0])` de dimension $d_h \times n$, mais la multiplication se fait terme à terme, donc `grad_ha` est de dimension $d_h \times n$:

```
grad_ha = grad_hs * np.where(ha > 0, [1], [0])
```

`grad_b1` est de dimension $d_h \times 1$:

```
grad_b1 = np.sum(grad_ha, axis=1)
```

Pour finir, `grad_ha` est de dimension $d_h \times n$ et `x` est de dimension $d \times n$ donc `grad_W1` est de dimension $d \times n$:

```
grad_W1 = np.dot(grad_ha, np.transpose(x))
```

Question 7

Comparez vos deux implémentations (avec et sans boucle sur les exemples du lot) pour vérifier qu'elles donnent le même gradient total sur les paramètres, d'abord avec $K = 1$. Puis comparez-les avec $K = 10$. Joignez à votre rapport les affichages numériques effectués pour cette comparaison.

$K = 1$: Gradients identiques

$K = 10$: Gradients identiques

Question 8

Mesurez le temps que prend une époque sur MNIST (1 époque = 1 passage complet à travers l'ensemble d'entraînement) pour $K = 100$ avec chacune des deux implémentations (mini-lot par boucle, et mini-lot avec calcul matriciel).

Sur MNIST, avec $d = 784$, $dh = 10$, $m = 10$, $wd = [0, 0, 0, 0]$, $K = 50$, $epoch = 1$, $learning_rate = 0.1$, on a obtenu les résultats suivants:

--- 7.8899230957 secondes --- pour 1 époque de MLP avec boucle.

--- 2.38572096825 secondes --- pour 1 époque de MLP avec calcul matriciel.

Question 9

Adaptez votre code pour qu'il calcule au vol, pendant l'entraînement, l'erreur de classification totale sur l'ensemble d'entraînement, en plus du coût optimisé total (somme des L encourus), ceci pour chaque époque d'entraînement, et qu'après chaque époque d'entraînement, il calcule aussi erreur et coût moyen sur l'ensemble de validation et de test. Faites en sorte qu'il les affiche après chaque époque les 6 nombres correspondants (erreur et coût moyen sur train, valid, test) et les écrive dans un fichier.

Voir le fichier error_and_cost_mlp_mat.csv et voir la console pour un affichage (ainsi que la méthode train du MLP pour les print et les calculs)

Question 10

Entraîner votre réseau sur les données de MNIST. Produisez les courbes d'entraînement, de validation et de test (courbes de l'erreur de classification et du coût en fonction du nombre d'époques d'entraînement, qui correspondent à ce que vous avez enregistré dans un fichier à la question précédente). Joignez à votre rapport les courbes obtenues avec votre meilleure valeur d'hyper-paramètres, c.à.d. pour lesquels vous avez atteint la plus basse erreur de classification sur l'ensemble de validation. On suggère deux graphiques : un pour les courbes de taux d'erreurs de classification (train, valid, test avec des couleurs différentes, bien précisées dans la légende) et l'autre pour la perte moyenne (le L moyen sur train, valid , test). Normalement vous devriez pouvoir atteindre moins de 5% d'erreur en test. Indiquez dans votre rapport la valeur des hyper-paramètres retenue et correspondant aux courbes que vous joignez. Points boni pour une erreur de test inférieure à 2%.

La valeur finale des hyper-paramètres retenus est:

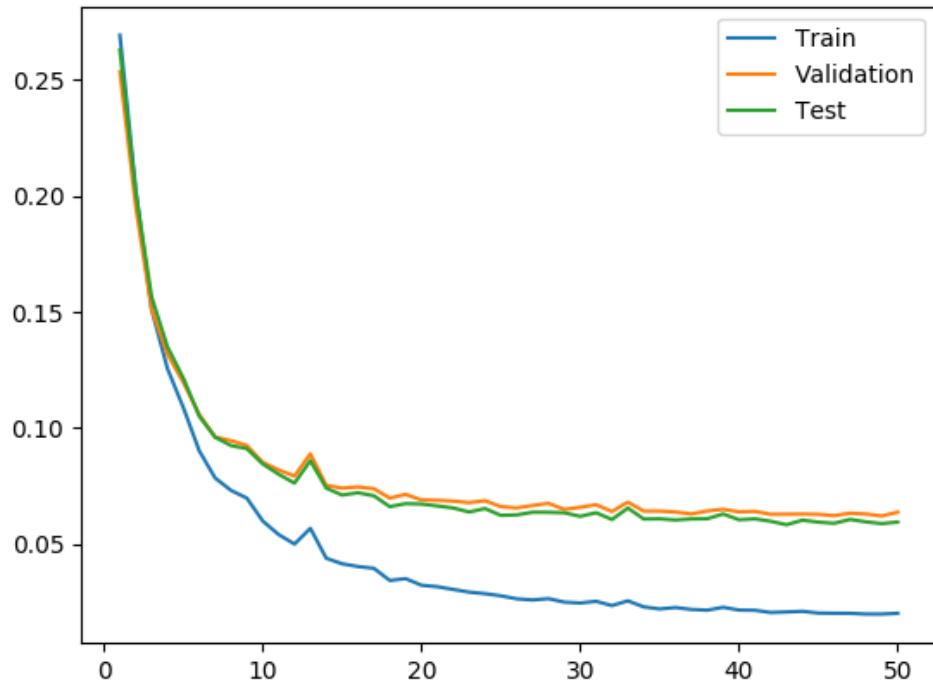
d = 784, dh = 784, m = 10, wd = [0, 0.005, 0, 0.005], K = 32, learning_rate = 0.05, epoch = 50

On obtient à l'époque 50:

NoEpoch, TrainErr, ValidErr, TestErr, TrainAvgLoss, ValidAvgLoss, TestAvgLoss

50, 0.156%, 1.76%, 1.82%, 0.0200190915627, 0.0636486300084, 0.0593582639471

Cout moyen - 50 époques



Erreur en % - 50 époques

