|  |
| --- |
|  |
| GPA749 |
| Laboratoire 2 |
|  |
| **Flavien Deschaux – Adrien Vassal** |
|  |

|  |
| --- |
|  |

Table des matières

[1. Le réseau MLP 2](#_Toc433353040)

[1.1. Architecture 2](#_Toc433353041)

[1.2. Entrainement du réseau 3](#_Toc433353042)

[1.3. Notre cas 4](#_Toc433353043)

[2. Implémentation du réseau MLP 5](#_Toc433353044)

[3. Expérience 1 : impacte du paramètre d’apprentissage 5](#_Toc433353045)

[4. Expérience 2 : impacte de l’apprentissage par batch 5](#_Toc433353046)

[5. Expérience 3 : visualisation du surentrainement 5](#_Toc433353047)

[6. Expérience 4 : impacte du surentrainement 5](#_Toc433353048)

[7. Expérience 5 : impacte des hyper-paramètres sur les résultats 5](#_Toc433353049)

[7.1. La construction des batch 5](#_Toc433353050)

[7.2. Nombre de neurones de la couche cachée 5](#_Toc433353051)

[7.3. Nombre de neurones de la couche d’entrée 5](#_Toc433353052)

[8. Bonus 6](#_Toc433353053)

Introduction

L’objectif de ce laboratoire est d’entrainer un réseau de neurones à reconnaitre les contours d’une image.  
Dans notre cas, il s’agira d’entrainer un réseau MLP en testant différente valeur pour l’ensemble de hyper-paramètres ce celui-ci.  
Ainsi le but sera de définir l’impact de ces hyper-paramètres sur la solution obtenue et de trouver les meilleurs paramètres possibles afin que notre réseau soit le plus performant.  
Les trois paramètres sur lesquels nous allons nous concentrer sont : la constante d’apprentissage, l’apprentissage par batch, et le nombre d’époque optimal.

# Le réseau MLP

Dans le précédent laboratoire nous avons étudiés le réseau perceptron et mis en lumière son plus gros défaut : il ne peut pas classifier que des données linéairement séparables.  
Or en pratique il se trouve que les données testées ne sont (pratiquement) jamais linéairement séparable.  
C’est suite à ce besoin de trouver une méthode de classification pour des données non linéairement séparable qu’est nait le MLP ou Multi Layer Perceptron (perceptron multicouche).

## Architecture

Un MLP comme son nom l’indique est constitué de plusieurs couche. L’entrée du MLP peut être de taille variable en fonction des données du problème.  
La couche de sortie du MLP est elle aussi de taille variable, mais on impose que sa fonction d’activation soit dérivable, on utilisera la fonction sigmoïde.

Les couches situées entre l’entrée et la sortie sont appelées couches cachées. Elles sont composées d’un nombre variable de neurones de McCulloch&Pitts, elle aussi composée de fonction dérivable (encore une fois nous utiliserons la fonction sigmoïde).

L’architecture peut être décrite par le graphique suivant :

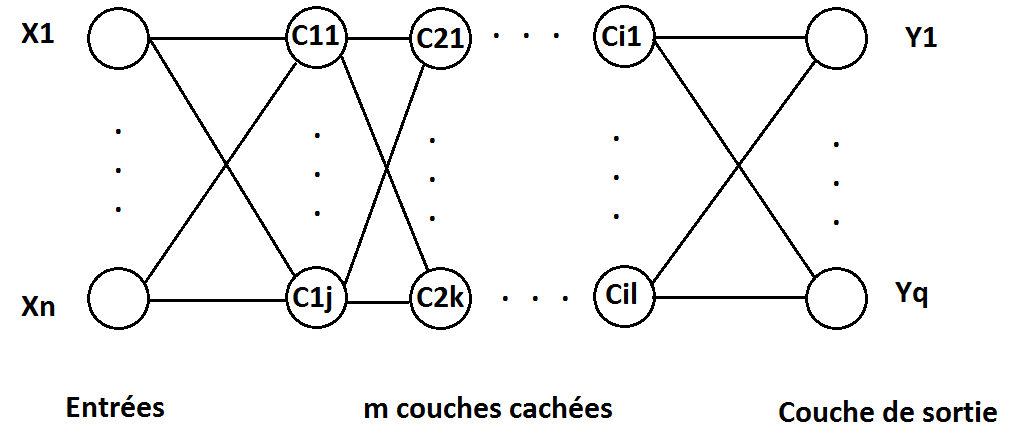


Figure 1 : Représentation graphique d'un MLP

Sur le graphique ci-dessus :

* le nombre d’entrée est variable
* le nombre de couches cachées est variable
* le nombre de neurones par couche cachée est variable
* le nombre de sortie est variable

Le réseau est à peine crée qu’il y a déjà plusieurs paramètres à définir. Généralement ces paramètres sont à définir en fonction du problème et des données d’entrées.

## Entrainement du réseau

Pour entrainer un tel réseau il faut procéder en deux phases :

* La propagation directe

On présente une entrée au réseau, on calcul sa sortie et on en déduit l’erreur obtenue.

* La rétropropagation

C’est cette étape qui va nous permettre de définir une façon de modifier les poids synaptiques de notre réseau.

Basée sur l’algorithme delta généralisé, l’idée est de minimiser l’erreur quadratique totale (aussi appelé Mean Square Error) faite par notre réseau.

Pour ce faire nous allons réaliser la rétropropagation du gradient de l’erreur.   
Le gradient étant défini (en un point) comme la pente la plus forte qui point vers le maximum local.   
Ainsi pour minimiser l’erreur quadratique, il faut se déplacer dans le sens inverse du gradient de cette erreur.  
 Fort de ces observations, il ne reste plus qu’à exprimer l’impact de chaque neurone sur la sortie, autrement dit exprimer le gradient de l’erreur en fonction des poids synaptiques associés à chaque neurone.

(Qui dit gradient dit dérivée, ce qui explique pourquoi il doit y avoir une fonction d’activation dérivable)

Ainsi après calcul nous avons les équations suivantes :

Où :

* k : le nombre de neurone de la couche de sortie
* j : le nombre de neurone de la couche cachée considérée
* wkj : le poids de la connexion entre le la sortie k et le neurone j
* : la variation à apporter au poids wkj
* : la variation à apporter au poids wij
* : le paramètre d’apprentissage

A noter que nous introduisons un nouveau paramètre à régler pour notre réseau : , ce paramètre sert à exprimer le fait que les poids synaptiques du réseau ne doivent pas changer du tout au tout à chaque rétropropagation .

## Notre cas

Dans notre cas nous allons utiliser un réseau possédant une unique sortie. En effet nous cherchons à savoir si un pixel est un contour ou non. Il s’agit donc d’une information binaire, ce qui explique pourquoi nous n’avons qu’une seule sortie.

Pour des questions de facilité d’implémentation, nous allons restreindre le nombre de couche cachée à 1.  
Cependant le nombre d’entrée et le nombre de neurone dans la couche cachée sont des paramètres que nous allons pouvoir modifier.

# Implémentation du réseau MLP

Deux approches sont possibles pour implémenter ce réseau :

Soit nous appliquons les formules avec des sommes et nous travaillons élément par élément, soit nous considérons des calculs matriciels.  
La deuxième solution étant bien plus optimal en termes de temps de calcul, c’est celle-là qui a été choisie.

# Expérience 1 : impacte du paramètre d’apprentissage

* **Objectif**

L’objectif de cette expérience est de définir l’impact du paramètre d’apprentissage sur la solution désirée.  
Le paramètre d’apprentissage permet de modérer la modification de poids lors de la rétropropagation du gradient de l’erreur.

Prenons un exemple pour expliquer son utilité:  
Un réseau de neurones arrive parfaitement à classifier 100 exemples donnés, mais aux 101 exemples, il fait une erreur. S’il n’y avait pas de paramètre d’apprentissage, la valeur des poids synaptiques serait fortement impactée lors de la rétropropagation. Or notre réseau a réussit à classer une très large majorité d’exemple sans se tromper, ce qui signifie qu’il était relativement bien entrainé et donc que les poids synaptiques trouvés était correctes. Ainsi ne pas mettre de constante d’apprentissage viendrais détruire le travail d’apprentissage effectué jusque-là.

* **Expérience**

Pour définir l’impact du paramètre d’apprentissage, nous fixons tous les autres paramètres du réseau et nous entrainons notre réseau pour le jeu de paramètre défini ci-dessous :

En plus de comprendre l’impact de la constante d’apprentissage, le but de cette expérience est de trouver un intervalle dans lequel donne des résultats correct afin de relancer la même expérience sur un intervalle plus petit.

Pour comparer les différents résultats entre eux, nous allons nous baser sur deux données : le temps de stabilisation et l’évolution de l’erreur quadratique dans le temps.

* **Résultats**

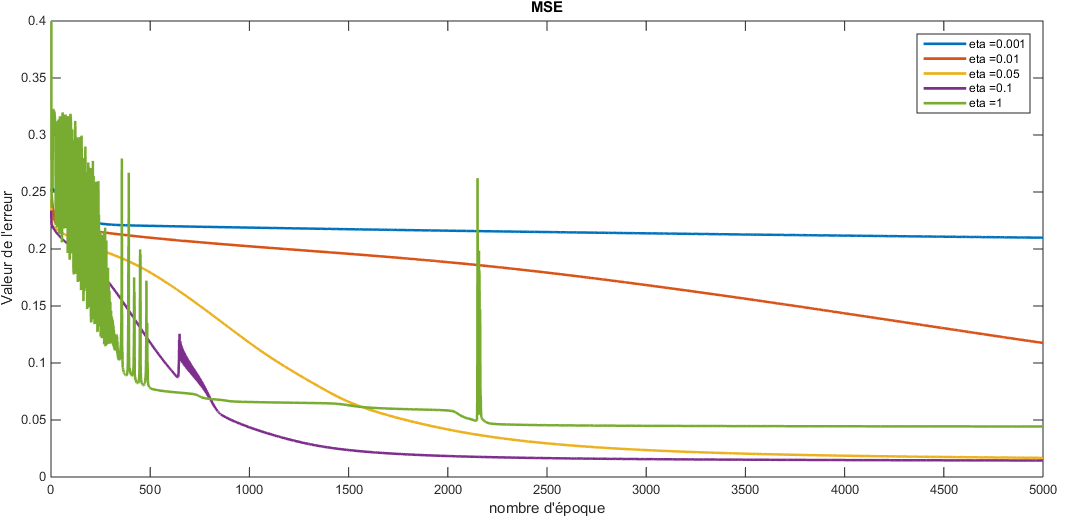
En utilisant le vecteur défini précédemment nous obtenons les courbes d’erreur suivantes :

Figure : Evolution du MSE en fonction du paramètre d'apprentissage

Il apparait clairement que nous avons trois cas distinct :

* Cas  : Dans ce cas la constante d’apprentissage est trop grande. Cela se remarque par le fait que l’erreur quadratique peut augmenter entre deux époques.  
  Ce phénomène vient du fait que lorsque la descente du gradient se rapproche du ou d’un minimum local, la modification des poids à chaque itération est tellement forte que l’algorithme oscille autour du minimum sans arriver à se stabiliser sur celui-ci.
* Cas  : C’est le cas où nous obtenons les résultats les plus prometteurs, il n’y a pas d’oscillation sur l’erreur quadratique qui se stabilise autour d’une valeur fixe dans un laps de temps acceptable (bien que le cas soit limite au niveau du temps de stabilisation).
* Cas  : Pour une valeur trop petite la valeur de l’erreur quadratique n’a pas le temps de se stabiliser autour d’une valeur en seulement 5000 itérations.

Pour mieux comprendre l’aspect temporelle lié à la constante d’apprentissage nous allons entrainer notre réseau sur un plus grand nombre d’époque (100 000) et étudier l’impact de la valeur de sur le temps de stabilisation de l’erreur quadratique.  
Les données obtenues après entrainement sont résumées dans le tableau suivant

Tableau : Temps de stabilisation du MSE en fonction de eta

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Valeur du MSE final | / | 0 | 0 |
| Temps de stabilisation (en nombre d’époque) | / |  |  |

(Ces valeurs peuvent changer d’un apprentissage à l’autre mais c’est l’écart entre les temps de stabilisation qui nous intéresse).

D’après les données obtenue, plus le paramètre d’apprentissage est petit plus le temps de stabilisation du MSE est long. Nous sommes dans le cas où chaque modification des poids synaptiques est tellement petite que l’algorithme de descente du gradient est très lent à converger vers le minimum. Ainsi pour un trop petit notre réseau devient très lent pour apprendre, nécessitant beaucoup plus d’itération pour que le MSE se stabilise autour d’une valeur finale.

* **Conclusion**

Pour entrainer notre réseau convenablement, il faut choisir une valeur de ni trop grande (pour éviter les oscillations), ni trop petite (pour éviter un temps d’apprentissage trop lent). Pour trouver la valeur optimale pour il faut refaire l’expérience précédente pour un plus petit intervalle admissible (Andrew conseille de faire varier le pas de 0.03 entre chaque valeur de ):

Nous obtenons ainsi les courbes suivantes :

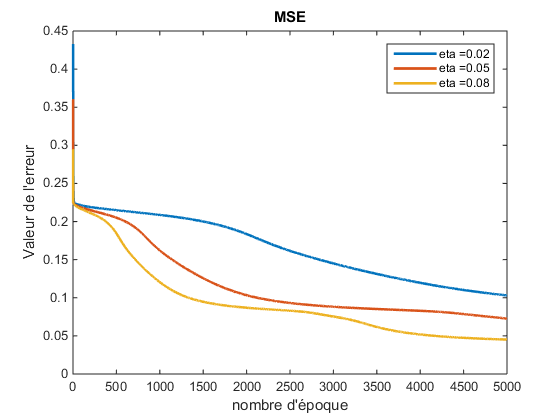


Figure : Evolution du MSE pour des valeurs de bonnes valeurs de eta

L’ensemble des résultats ci-dessus sont ne présente aucune oscillation et ne sont pas trop lent, ainsi nous choisirons le milieu de l’intervalle à savoir pour le reste du laboratoire.

* **Amélioration possible**

Une des améliorations possible serait d’attribuer à chacun des poids une constante d’apprentissage qui lui est propre.  
Il se peut que certain poids se règle plus vite que d’autre. Ainsi en observant l’évolution du gradient de l’erreur associé à chacun des poids nous pourrions régler au mieux la constante d’apprentissage pour qu’elle s’adapte à son poids associé.

* Si la valeur du gradient de l’erreur change de signe : nous sommes en train d’osciller autour du minimum, il faut donc baisser la valeur de pour stopper les oscillations et atteindre ce minimum.
* Si la valeur du gradient ne change pas de signe c’est que nous sommes encore loin du minimum, nous pouvons donc nous permettre d’augmenter la valeur de pour accélérer l’apprentissage.

# Expérience 2 : impacte de l’apprentissage par batch

* **Objectif**

Lorsqu’un réseau MLP est en phase d’entraînement il y a trois grandes façons de l’entrainer par époque :

* Incrémental : on présente un exemple à l’entrée du réseau, on calcul la propagation directe, la rétropropagation et on applique le changement de poids. On répète ce cycle jusqu’à ce que toutes les données du set d’entrainement soit passés.
* Set total : on présente le set d’entrainement complet au réseau, celui-ci va calculer le changement de poids liés à chaque exemple, additionner ces changements et modifier les poids du réseau une fois tous les exemples du set d’entrainement passés.
* Par batch : on décompose le set d’entrainement en batch (dont la taille est à définir). On présente les batch les uns après les autres au réseau et pour chaque batch on applique la modification de poids calculé avant de présenter le nouveau batch.

Notre objectif est de quantifier l’impact de la taille des batch sur l’erreur obtenue, le temps de convergence de cette erreur ainsi que le temps d’apprentissage du réseau.

* **Expérience**

Pour définir l’impact de la taille des batch nous fixons l’ensemble des autres paramètres et nous calculons l’erreur moyenne pour l’ensemble des batch suivants:

(La première valeur représente le cas incrémental, tandis que la dernière représente le set total).

A noter que dans tous les cas, les données d’entrée seront prises dans un ordre aléatoire, et que chacun des batch est créé de tel sorte que la proportion de vecteur d’apprentissage appartenant à chaque classe (contour ou non) soit respecté. Une explication plus détaillé de ce procédé sera présenté dans l’expérience 5.

A chaque époque les batch sont régénérés aléatoirement

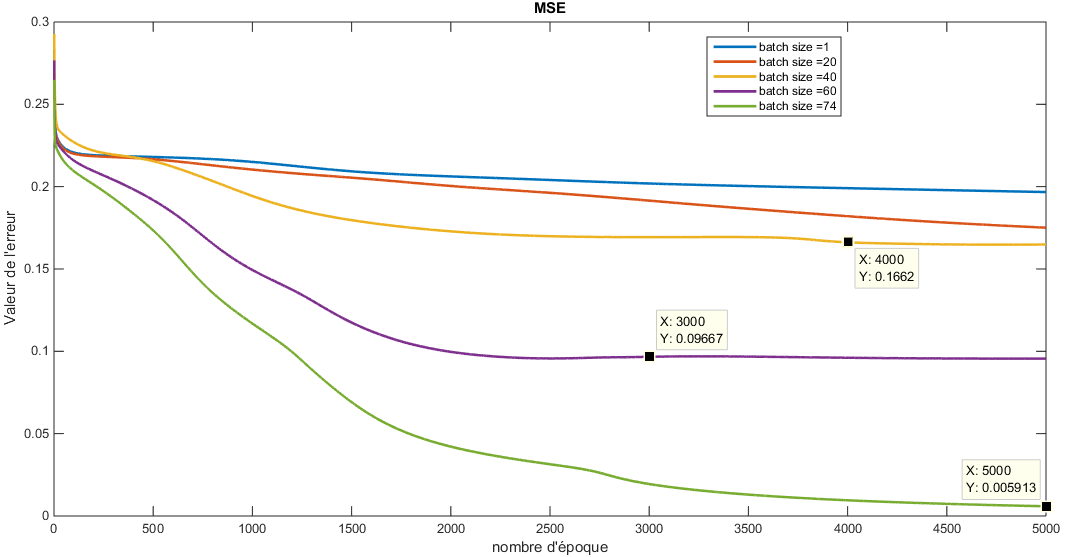
* **Résultat**

Figure : Evolution du MSE en fonction de la taille des batch

Tableau : Temps de stabilisation et valeur finale du MSE en fonction de la taille des batch

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Taille des batch | 1 | 20 | 40 | 60 | nbTotal |
| Temps de stabilisation du MSE (en nombre d’époque) | / | / |  |  |  |
| Valeur finale du MSE | / | / |  |  |  |

Tableau : Temps d'apprentissage en fonction de la taille des batch

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Taille des batch | 1 | 20 | 40 | 60 | nbTotal |
| Durée de l’entrainement (en secondes) | 25.97 | 2.41 | 0.945 | 0.934 | 0.930 |

* **Conclusion**

Batch plus grosses = diminution temps d’entrainement, diminution du temps de stabilisation du MSE

Batch plus petit = (pas dans notre cas) réduit les chances de rester dans un min local, augmente les temps d’entrainement

Générale : S’applique sur une base volumineuse et redondante, dépend donc beaucoup des données d’entrée

Dans notre cas il est préférable de faire des tests sur la base entière, nous ne tombons pas dans des minimum locaux.

* **Amélioration possible**

Paralléliser le travail !

# Expérience 3 : visualisation du surentrainement

**Objectif**

Quand un réseau de neurones ne généralise pas bien les données, il peut se trouver dans deux cas distincts :

Sur apprentissage :

Sous apprentissage :

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

Avoir plus de données (voir 7.4) (pour éviter les que les hyper paramètres du réseau soit parfaitement adapté à la base d’entrainement, ce qui se généraliserait mal)

Adapter la complexité du modèle au problème (voir 7.2-7.3)

Faire la moyenne de plusieurs modèles

Pénaliser les poids ayant des valeurs élevée

# Expérience 4 : impacte du surentrainement

**Objectif**

Prévenir le surentrainement

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

Déjà vue à l’expérience 3

# Expérience 5 : impacte des hyper-paramètres sur les résultats

### La construction des batch

**Objectif**

Représentation égale des deux classes dans chacun des lots

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

### Nombre de neurones de la couche cachée

**Objectif**

Nb neurones couche caché = complexité du modèle considéré

Faire attention à un modèle trop complexe qui entraine à un sur entrainement

Faire attention à un modèle pas assez complexe qui sera incapable de trier correctement les données.

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

### Nombre de neurones de la couche d’entrée

**Objectif**

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

### Augmentation artificielle du nombre d’exemple d’entrainement

**Objectif**

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

### Mise en place du momentum

**Objectif**

**Expérience**

**Résultat**

**Conclusion**

**Amélioration possible**

### Mise en place d’un paramètre d’apprentissage propre à chaque poids

# Bonus

George POWER !