ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ELE-767
Apprentissage machine en intelligence artificielle
Remis à Chakib Tadj

Laboratoire 2 :
Réseau multicouche à rétropropagation des erreurs

Guillermo Alberto Martinez	MARG04099307
Sébastien Tardif	TARS11099708
Jordan Perus	PERJ09059707

Montréal, le 20 Mars 2022

Introduction

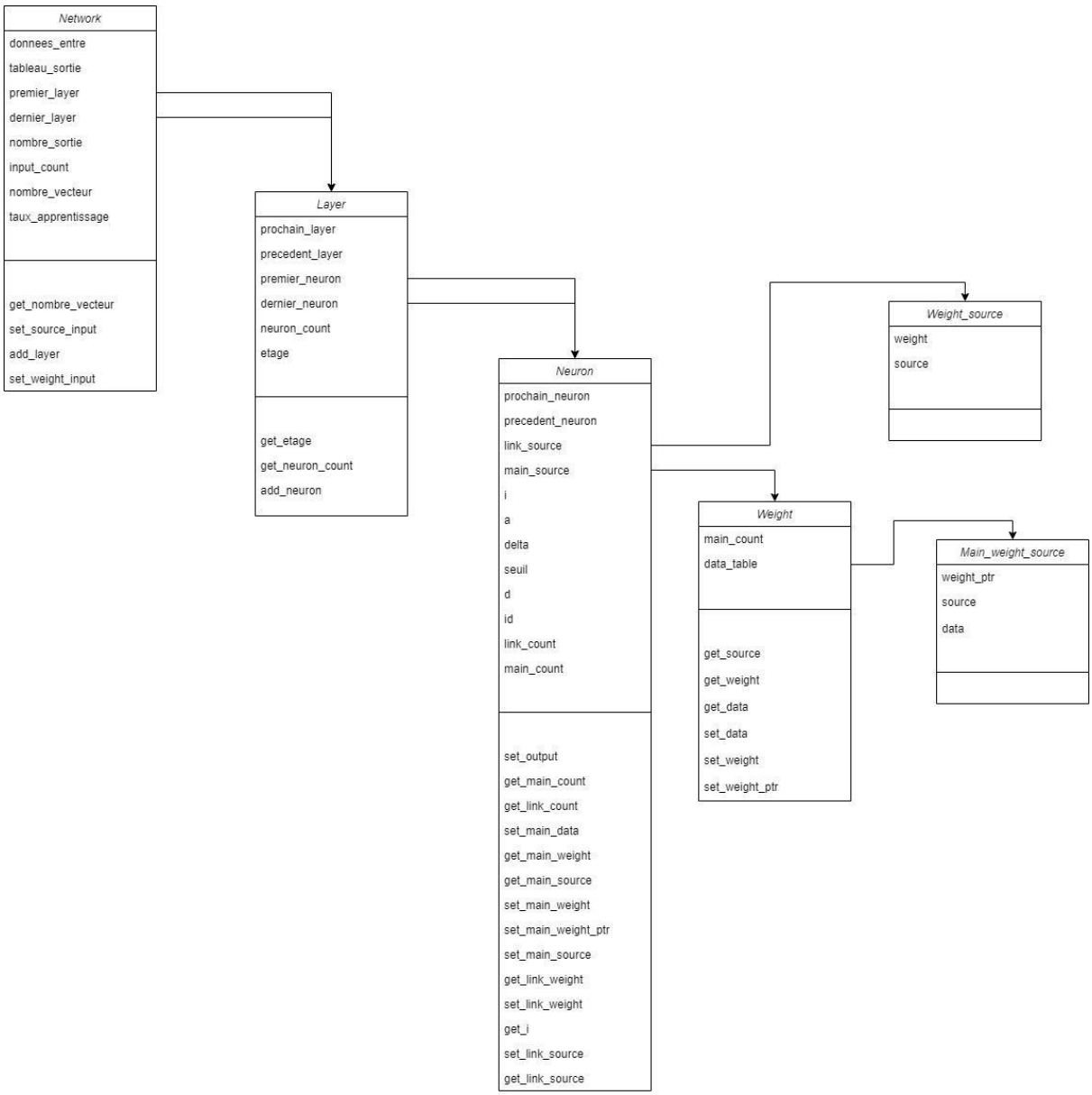
Ce laboratoire consiste en la création d'un réseau de neurones permettant la reconnaissance vocale. Le réseau à concevoir sera un réseau à rétropropagation du gradient de l'erreur.
Le langage de programmation choisi sera le C++, par facilité, car tous les membres de l'équipe peuvent gérer du code écrit en C++. Nous utiliserons donc Visual Studio 2017 ainsi que le système de Contrôle de Code Source très populaire, Git qui nous aidera dans la conception en équipe.
Nous avons certaines contraintes à respecter pour réaliser le réseau de neurones attendu. Ces contraintes sont inscrites dans le document d'explication du laboratoire.
À part ces contraintes, nous avons le champ libre pour réaliser ce projet.

1. Les différentes étapes du développement du laboratoire

Plusieurs étapes ont été nécessaires pour la réalisation du réseau de neurones.
Le langage de programmation qu'on a utilisé pour ce projet est le langage C++.

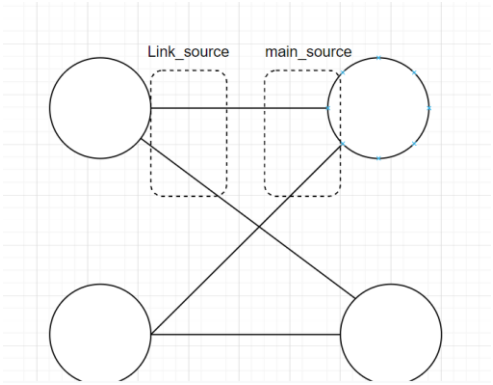
La première étape est de construire un réseau de neurones. On a utilisé plusieurs classes pour séparer chaque élément. On a séparé comme suit: Class Neuron, Class Network, Class weight, Class Layer.

Voici un bref schéma de notre architecture :



Dans cette architecture, il y a un network qui contient plusieurs layer et chaque layer contient plusieurs neurones. Network contient le premier et dernier layer. Pour accéder à d' autre layer il faut parcourir une liste doublement chaînée. La même logique s'applique avec les neurones que chaque layer à son premier et dernier neurone. Chaque neurone contient

un tableau link_source et main_source. Link Source contient les poids qui sont reliés à ce neurone sur sa sortie et main source contient les poids qui sont reliés sur son entrée. Comme représenté ci-dessous :



Une fois qu'on a déterminé l'architecture, il fallait créer une fonction pour créer tous les liens des poids entre chaque neurone. Cette fonction qui permet de faire cela est *creation_MLP()*.

Par la suite, il fallait récupérer les données dans un fichier texte et comprendre comment les données étaient réparties. Nous savons que les données étaient réparties en 26 unités, dont 12 unités statiques et 12 unités dynamiques. On a décidé de travailler seulement avec les unités statiques. Les valeurs dynamiques on aller les enlever. On a créé une fonction qui permet de régénérer un fichier texte qui contient seulement les valeurs statiques pour chaque entrée. Cette fonction s'appelle *pretraitement_basedonne()*. Il fallait créer une autre fonction qui permet d'extraire le fichier texte qui a été prétraité. Cette fonction est parser_basedonne*().*

Une fois qu'on a récupéré les données, il fallait faire une fonction pour faire la rétropropagation. Cette fonction est *calcul_retropopagation()*. Il fallait aussi faire la fonction *update_mlp()* qui permet de faire la mise à jour d'une nouvelle entrée et sortie du MLP après la rétropropagation. Aussi, la fonction *evaluation_MLP()* permettra de voir si on avait des bons résultats avec la sortie désirée de la dernière couche et les valeurs de sortie obtenues.

À la dernière étape, on a incorporé une interface graphique. Cette interface graphique sert à choisir la configuration qu'on souhaite pour l'apprentissage. Nous avons choisi la librairie FLTK pour sa rapidité et parce qu'elle ne prend pas trop de place. Elle nous permet d'avoir des boutons et des champs de textes d'entrés et de sortis, ce qui est tout ce dont on avait besoin pour ce laboratoire.

2. Les deux fonctions d'activation utilisées

Dans le cadre de notre laboratoire, nous avons mis deux fonctions d'activation .
La première fonction d'activation est le sigmoïde

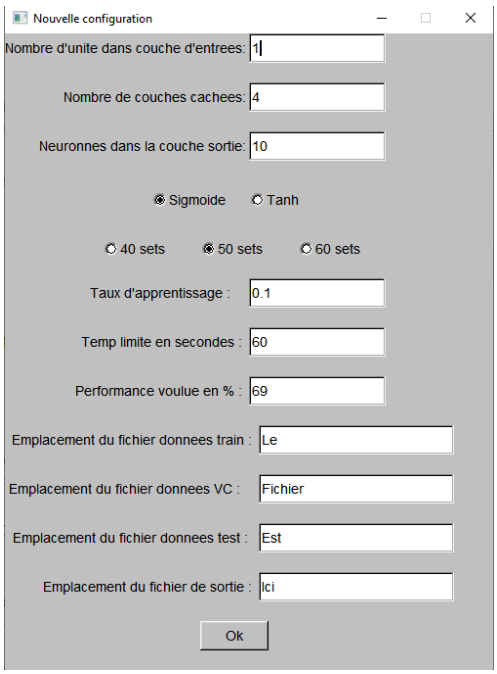
$$sigmoide(i) = \frac{1}{1 + e^{-i}}$$

La deuxième fonction d'activation est tangente hyperbolique.

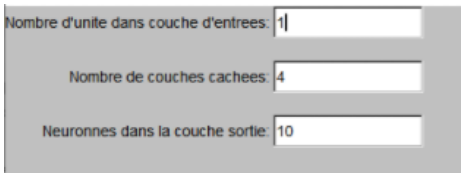
$$tanh(i) = \frac{e^{2i} - 1}{e^{2i} + 1}$$

3. Les contraintes complémentaires ajoutées

Dans les contraintes complémentaires, on a ajouté une interface graphique permettant de modifier les nombres de couches, le nombre de neurones, la fonction d'activation, le temps limite de l'apprentissage, le taux d'apprentissage et la performance de la validation croisée qu'on souhaite s'arrêter.



Deuxième contrainte, comme mentionnée, on peut exécuter le programme avec plusieurs couches cachées (2 couches cachées et plus) et par la suite on va devoir rentrer le nombre de neurones dans chacune des couches cachées.



Troisième contrainte, on peut dans notre programme améliorer la règle d'apprentissage en modifiant le taux d'apprentissage, la durée d'apprentissage et modifier les poids dans l'initialisation avec un fichier de sauvegarde.

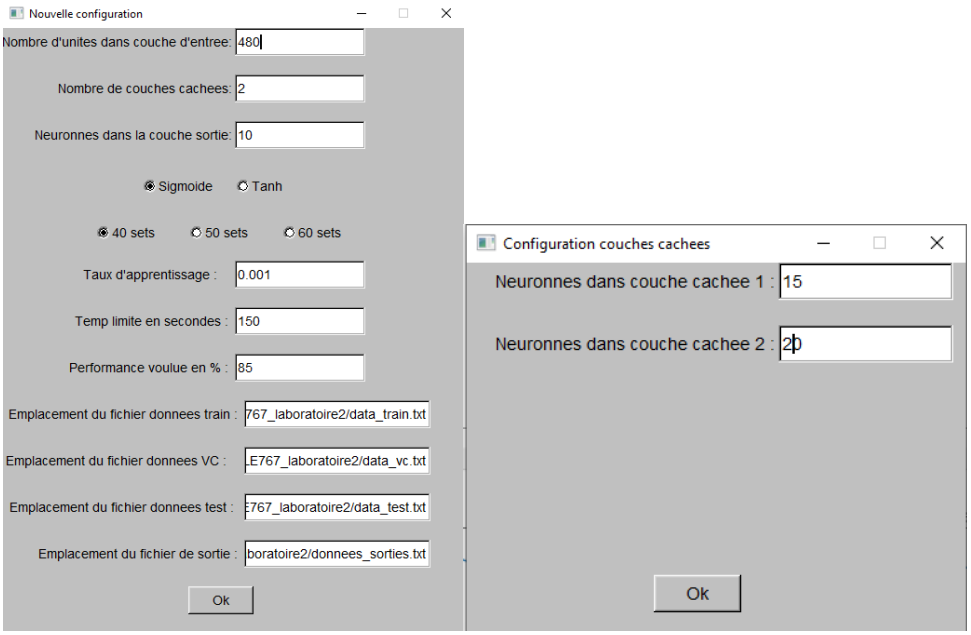
4. Les améliorations apportées et leur efficacité

En plus des points mentionnés ci-haut, pour améliorer la performance de notre réseau de neurones, nous avons utilisé un langage de programmation assez bas niveau. Ainsi on peut avoir un code plus rapide. Une expérience qui aurait pu être intéressante aurait été de faire le même réseau avec un langage de programmation de plus haut, et de plus bas niveaux. Par exemple en C et en Python, pour ensuite comparer si le langage offre une performance différente sur les réseaux neuronaux et leur vitesse.

5. Les différentes simulations effectuées

Fonction sigmoïde

Simulation avec un petit facteur de correction et peu d'époques

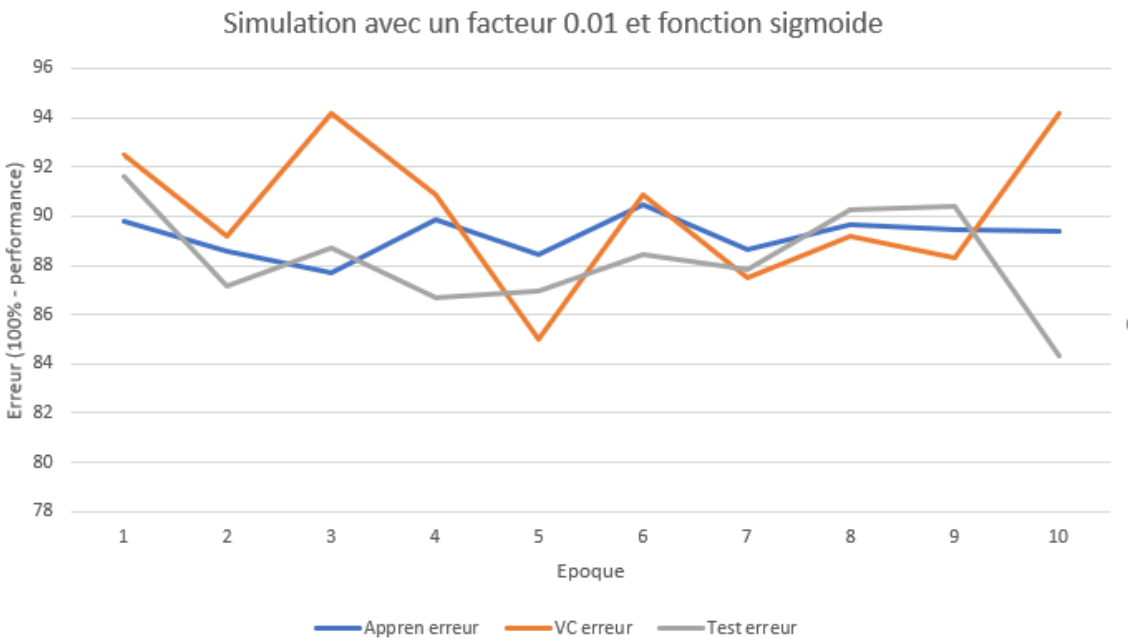


C:\Users\Marti\Desktop\Hiver-2022\ELE767\Laboratoire2\ELE767_laboratoire2\Lab2\x64\Debug\Lab2.exe

```
Epoque 1 performance apprentissage :    10.2239  elapsed time(seconde): 8
Epoque 1 performance vc :               7.5      elapsed time(seconde): 8
Epoque 1 performance test :             8.46154  elapsed time(seconde): 11
Epoque 2 performance apprentissage :    11.0448  elapsed time(seconde): 20
Epoque 2 performance vc :               10.8333  elapsed time(seconde): 20
Epoque 2 performance test :             12.8205  elapsed time(seconde): 23
Epoque 3 performance apprentissage :    12.3134  elapsed time(seconde): 32
Epoque 3 performance vc :               5.83333  elapsed time(seconde): 32
Epoque 3 performance test :             11.2821  elapsed time(seconde): 35
Epoque 4 performance apprentissage :    10.1493  elapsed time(seconde): 43
Epoque 4 performance vc :               9.16667  elapsed time(seconde): 44
Epoque 4 performance test :             13.3333  elapsed time(seconde): 47
Epoque 5 performance apprentissage :    11.5672  elapsed time(seconde): 55
Epoque 5 performance vc :               15       elapsed time(seconde): 56
Epoque 5 performance test :             13.0769  elapsed time(seconde): 59
Epoque 6 performance apprentissage :    9.55224  elapsed time(seconde): 67
Epoque 6 performance vc :               9.16667  elapsed time(seconde): 68
Epoque 6 performance test :             11.5385  elapsed time(seconde): 71
Epoque 7 performance apprentissage :    11.3433  elapsed time(seconde): 79
Epoque 7 performance vc :               12.5     elapsed time(seconde): 80
Epoque 7 performance test :             12.1795  elapsed time(seconde): 83
Epoque 8 performance apprentissage :    10.3731  elapsed time(seconde): 91
Epoque 8 performance vc :               10.8333  elapsed time(seconde): 92
Epoque 8 performance test :             9.74359  elapsed time(seconde): 95
Epoque 9 performance apprentissage :    10.597   elapsed time(seconde): 103
Epoque 9 performance vc :               11.6667  elapsed time(seconde): 103
Epoque 9 performance test :             9.61538  elapsed time(seconde): 107
Epoque 10 performance apprentissage :   10.597   elapsed time(seconde): 115
Epoque 10 performance vc :              5.83333  elapsed time(seconde): 115
Epoque 10 performance test :            15.641   elapsed time(seconde): 119
Epoque 11 performance apprentissage :   9.85075  elapsed time(seconde): 127
Epoque 11 performance vc :              10.8333  elapsed time(seconde): 127
Epoque 11 performance test :            9.10256  elapsed time(seconde): 130
Epoque 12 performance apprentissage :   11.4179  elapsed time(seconde): 139
Epoque 12 performance vc :              8.33333  elapsed time(seconde): 139
Epoque 12 performance test :            10.1282  elapsed time(seconde): 142
```

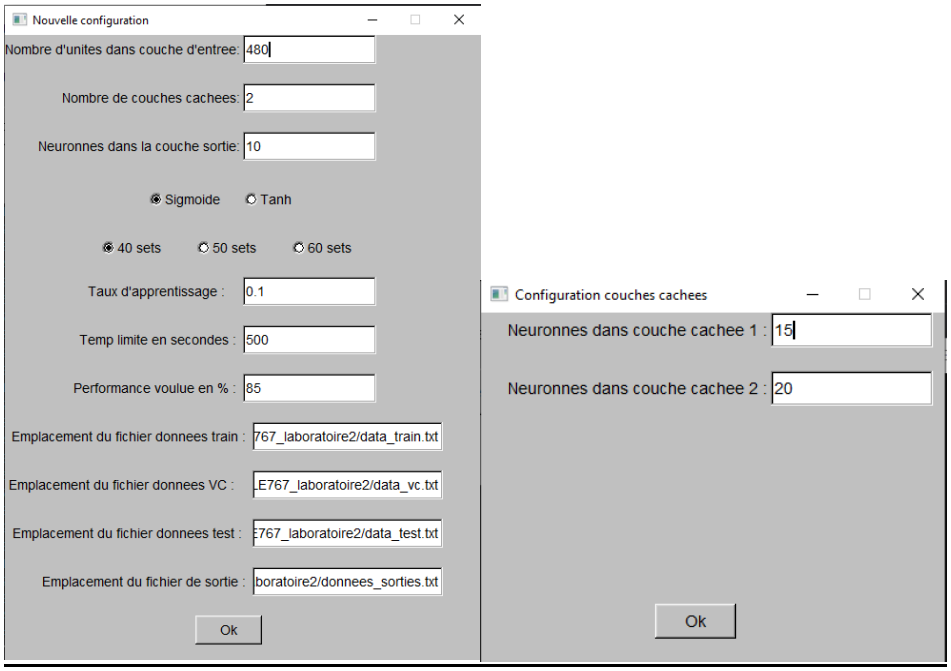
Performance

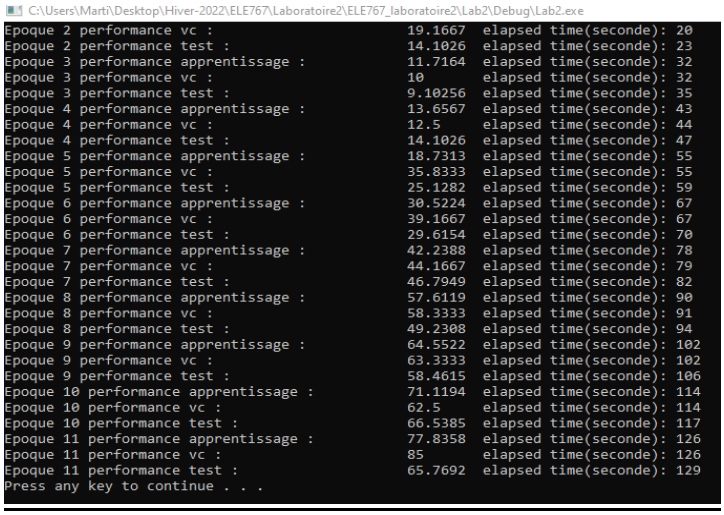
Temp en seconde

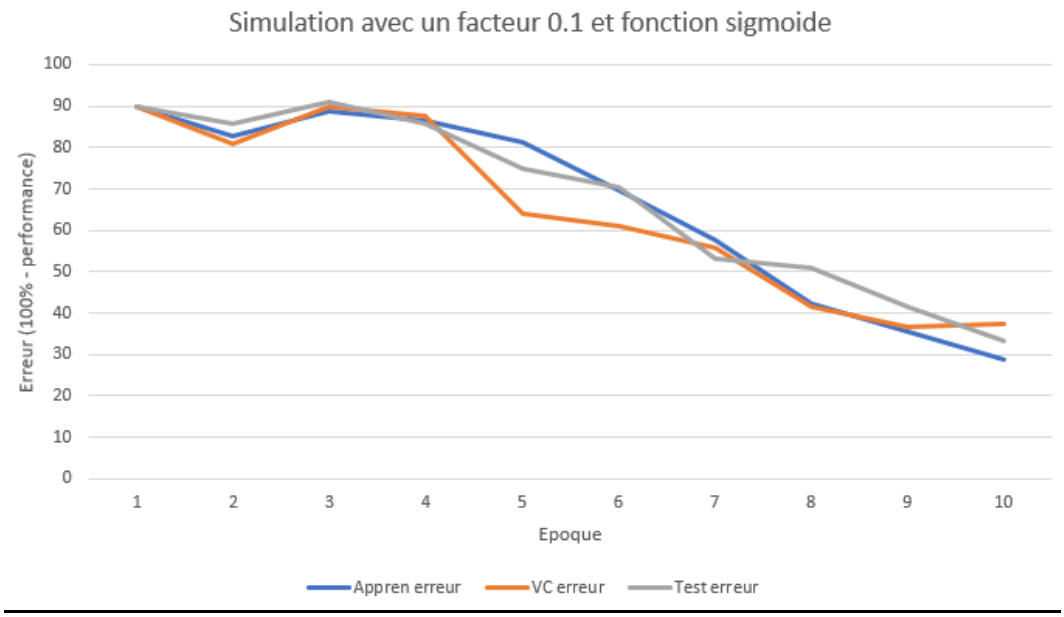


On peut voir qu'en utilisant peu d'époques et un petit facteur de correction, l'apprentissage ne se fait pas aussi bien. Avec un facteur de correction de 0.001, il faut alors faire plus d'époques pour arriver à un apprentissage parabolique.

Simulation en augmentant le facteur de correction

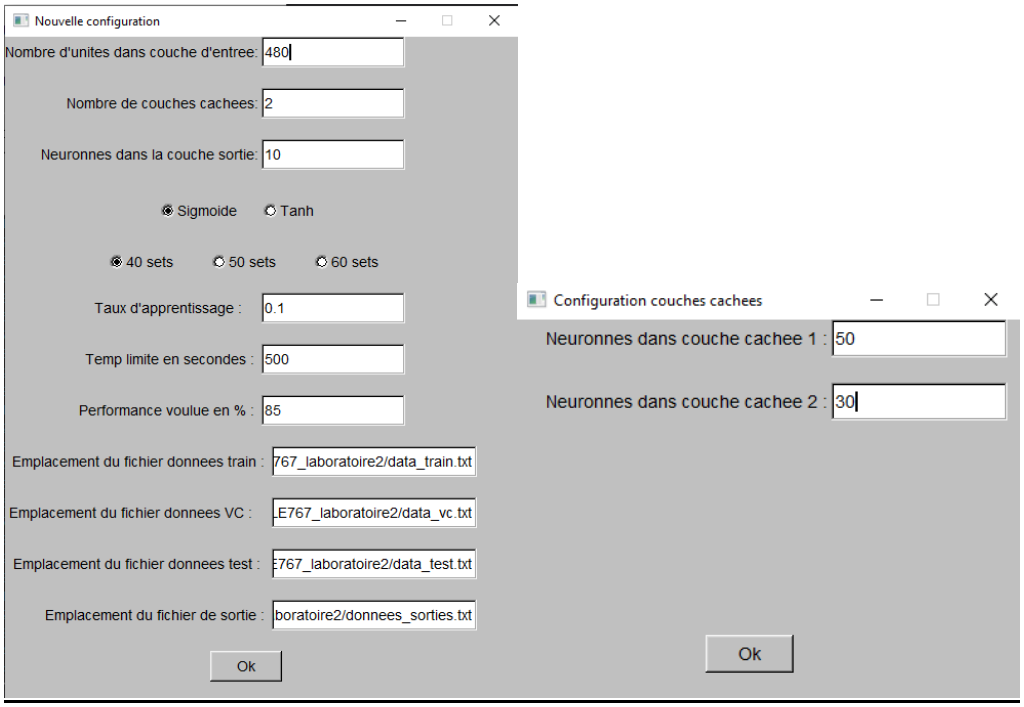


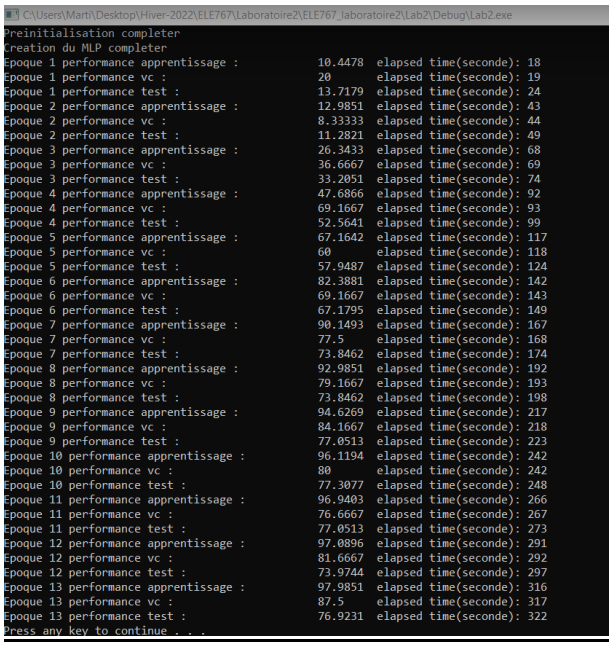


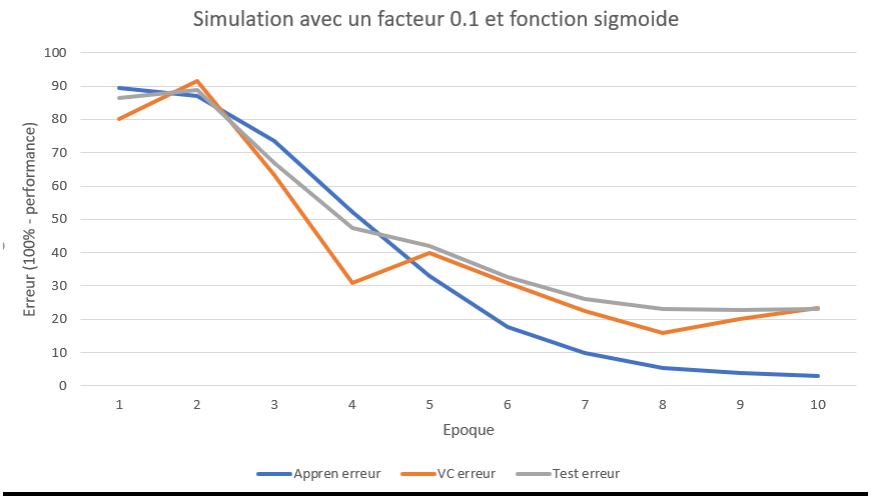


On peut voir qu'en augmentant le facteur de correction, l'apprentissage se fait plus rapidement. Cela est dû au fait que le facteur de correction est plus grand et donc on trouve plus rapidement le point d'optimisation.

Simulation en augmentant le nombre de neurones dans chaque couche



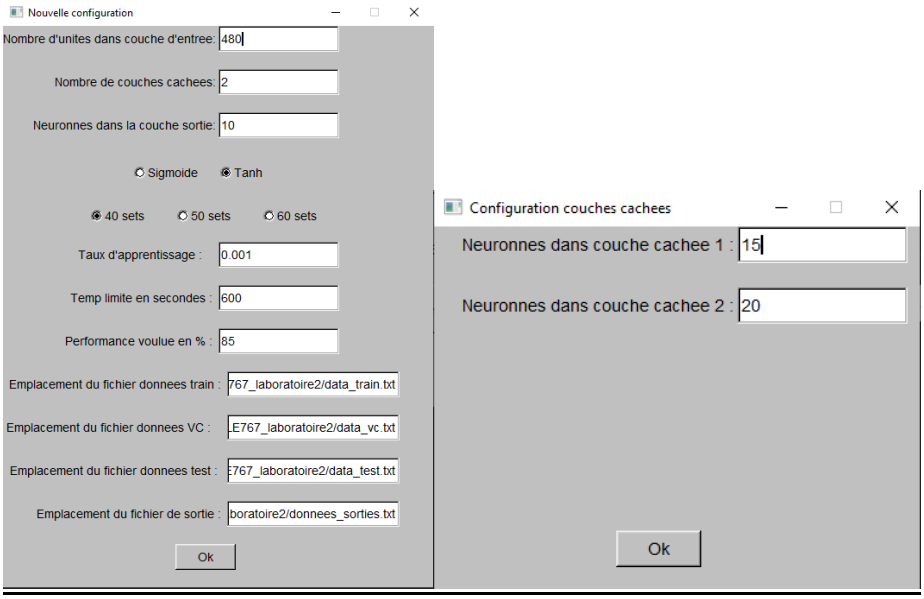




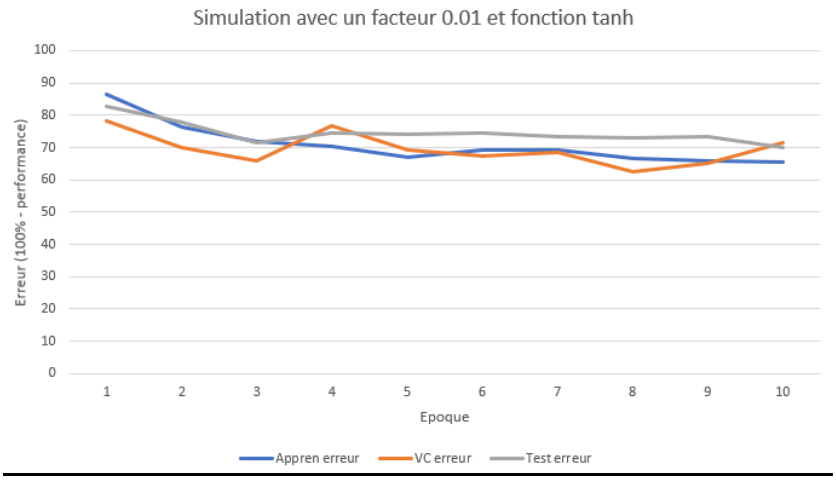
Dans cette simulation, on a augmenté le nombre de neurones dans la première couche de 50 et 30 dans la deuxième couche. Comme on peut voir dans les résultats, on arrive à atteindre des performances de test de 77%.

Fonction tangente hyperbolique

Simulation avec un petit facteur de correction et peu époque



```
C:\Users\Marti\Desktop\Hiver-2022\ELE767\Laboratoire2\ELE767_laboratoire2\Lab2\Debug\Lab2.exe
Epoque 1 performance apprentissage :            13.6567  elapsed time(seconde): 8
Epoque 1 performance vc :                       21.6667  elapsed time(seconde): 8
Epoque 1 performance test :                     17.3077  elapsed time(seconde): 11
Epoque 2 performance apprentissage :            23.5821  elapsed time(seconde): 19
Epoque 2 performance vc :                       30       elapsed time(seconde): 20
Epoque 2 performance test :                     22.1795  elapsed time(seconde): 23
Epoque 3 performance apprentissage :            28.0597  elapsed time(seconde): 31
Epoque 3 performance vc :                       34.1667  elapsed time(seconde): 32
Epoque 3 performance test :                     28.5897  elapsed time(seconde): 35
Epoque 4 performance apprentissage :            29.6269  elapsed time(seconde): 43
Epoque 4 performance vc :                       23.3333  elapsed time(seconde): 43
Epoque 4 performance test :                     25.641   elapsed time(seconde): 46
Epoque 5 performance apprentissage :            32.9851  elapsed time(seconde): 55
Epoque 5 performance vc :                       30.8333  elapsed time(seconde): 55
Epoque 5 performance test :                     26.0256  elapsed time(seconde): 58
Epoque 6 performance apprentissage :            30.7463  elapsed time(seconde): 66
Epoque 6 performance vc :                       32.5     elapsed time(seconde): 67
Epoque 6 performance test :                     25.3846  elapsed time(seconde): 70
Epoque 7 performance apprentissage :            30.8209  elapsed time(seconde): 78
Epoque 7 performance vc :                       31.6667  elapsed time(seconde): 78
Epoque 7 performance test :                     26.6667  elapsed time(seconde): 82
Epoque 8 performance apprentissage :            33.3582  elapsed time(seconde): 90
Epoque 8 performance vc :                       37.5     elapsed time(seconde): 90
Epoque 8 performance test :                     27.0513  elapsed time(seconde): 93
Epoque 9 performance apprentissage :            33.9552  elapsed time(seconde): 101
Epoque 9 performance vc :                       35       elapsed time(seconde): 102
Epoque 9 performance test :                     26.7949  elapsed time(seconde): 105
Epoque 10 performance apprentissage :           34.5522  elapsed time(seconde): 113
Epoque 10 performance vc :                      28.3333  elapsed time(seconde): 114
Epoque 10 performance test :                    30.1282  elapsed time(seconde): 117
Epoque 11 performance apprentissage :           36.9403  elapsed time(seconde): 125
Epoque 11 performance vc :                      36.6667  elapsed time(seconde): 125
Epoque 11 performance test :                    30.3846  elapsed time(seconde): 129
Epoque 12 performance apprentissage :           37.6866  elapsed time(seconde): 137
Epoque 12 performance vc :                      28.3333  elapsed time(seconde): 137
Epoque 12 performance test :                    27.8205  elapsed time(seconde): 140
```



Ici on a utilisé la fonction d'activation tangente hyperbolique. On peut voir qu'il est plus facile de faire l'entraînement. Cette fonction est plus performante que celle du sigmoïde, car même si on exécute avec le même nombre d'époques et le même facteur de correction, on aura des meilleurs résultats dans la fonction tangente.

6. Analyse du comportement du réseau de neurones développé

Avec ces résultats, on peut voir que l'apprentissage varie dépendant du nombre de neurones, du nombre de couches cachées, facteur de correction et du nombre d'époques.

Si on fait l'apprentissage avec beaucoup d'époques, plus l'apprentissage se fait mieux. On a aussi remarqué que le taux d'apprentissage fait varier les performances. Si le facteur de correction est petit, on va devoir faire plus d'époques pour avoir des bonnes performances.

On a réussi à obtenir un taux de test allant jusqu'à 77%, d'apprentissage de 98% et de VC de 87.5% en utilisant deux couches, avec 50 neurones sur la première et 30 sur la deuxième. Le temps pour cet apprentissage était d'environ 5 minutes 20 secondes.

7. Recommandations

L’apprentissage n’est pas relativement rapide, car on utilise un cœur de notre processeur pour rouler la séquence de notre code. On peut faire une amélioration si on fait du multitâches que chaque cœur d’un processeur prend une tache de code. Cela pourrait améliorer la rapidité de l'exécution de notre code.

8. Conclusions

Ce laboratoire a permis de mettre en pratique différents sujets étudiés en cours. Nous avons créé un Réseau de Neurones à rétropropagation du gradient de l’erreur, utilisant la règle d’apprentissage du delta généralisé. Ce réseau de neurones est écrit en C++.

Plusieurs contraintes à respecter, on était énoncées dans le document présentant le laboratoire, comme d’avoir 2 fonctions d'activation ou le fait que l’utilisateur puisse choisir combien de couches d’entrée, cachée, et de sortie il veut.

Le but de ce réseau est de détecter dans un signal vocal humain, un message inclus dans ce signal vocal, grâce à des caractéristiques extraites du signal.
Pour ce faire, il a fallu entraîner le réseau de neurones grâce à des caractéristiques extraites d’un signal afin qu’il puisse retrouver de lui-même des résultats désirés. La base de données contenant ces caractéristiques provient de Texas Instruments. Grâce à celle-ci, nous avons réalisé l'entraînement ou l'apprentissage de notre réseau.
Par la suite, il a fallu vérifier si ce que le réseau a appris est applicable à d’autres exemples et si le réseau est capable de donner le résultat désiré.

Pour finir, la mise en pratique du laboratoire nous a permis de revoir le langage C++. Il aura permis d’appliquer à une utilisation pratique, la théorie vue en classe sur les réseaux de neurones. Nous nous sommes également bien mieux familiarisés avec l’IDE Visual Studio 2017, ainsi qu’avec le système de contrôle de code source GitHub.