Licence 3 de Sciences, mention Informatique

Intelligence Artificielle

Rapport du projet : Perceptron et perceptron multi-couches avec Neuroph

Présenté par

Victor Constans
Luigi Coniglio

Table des matières

1 Fonctions booléennes

Dans la première section de ce rapport on se propose d'implementer des fonctions booléennes a l'aide des réseaux de neurones.

1.1 "Et" logique

En matematique la conjonction logique $\wedge$ est un connecteur logique que, étant donné deux propositions a et b forme une nouvelle propositions $a \wedge b$ qui est vrai seulement si a et b sont vraies.

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1 – Table de vérité de $a \wedge b$

1.1.1 Un perceptron pour la fonction "et"

Est il possible d'utiliser un perceptron pour apprendre la fonction logique "et" ? Bien sur, en effet même un perceptron mono-couche est suffisant pour obtenir ce resultat.

Le perceptron peut être utilisé comme un classificateur linéaire, c'est-à-dire l'algorithme du perceptron permettant de reconnaitre et donc classifier des données (ou points) linéairement séparables.

Un ensemble de points dans le plan caractérisé par deux sous-classes disjointes de points, est dit linéairement séparable si il éxiste une droite qui sépare complétement les deux sous-classes.

En trois dimension, un ensemble est linéairement séparable si il éxiste un plan qui sépare les deux classes. En quatre ou plus dimensions on ne parlera plus de plan mais d'hyperplan.

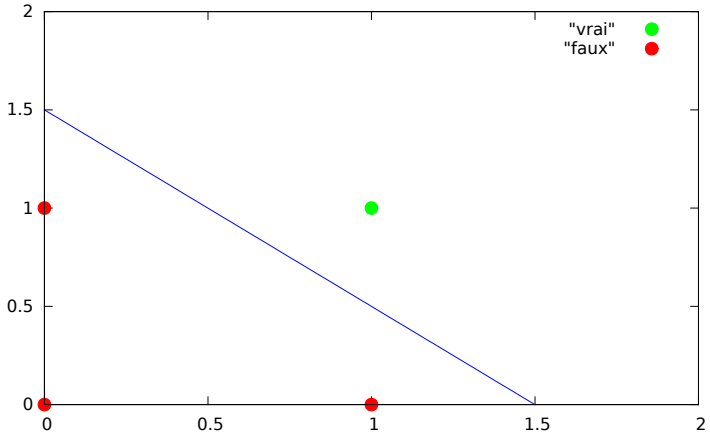

FIGURE 1 – La fonction logique "et" est linéairement séparable

Comme montre la figure 1 la fonction "et" est bien linéairement séparable. Etant donné que tout ensemble linéairement séparable peut être discriminé par un perceptron monocouche, on est sûr

de pouvoir trouver un perceptron monocouche qui engendre la fonction booléenne $\wedge$.

Le perceptron créé avec Neuroph qu'on utilisera pour cette fonction est très simple et est illustré dans la figure 2. Le data set pour l'apprentissage contiens les mêmes valeurs que la table 1.

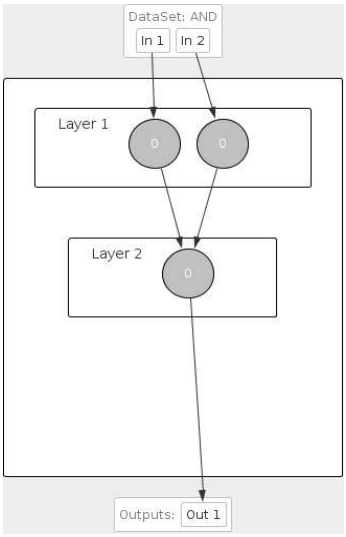

FIGURE 2 – Perceptron pour la fonction "et"

Pendant l'entrainement du perceptron (dont les poids ont été initialisés aléatoirement) avec un taux d'apprentissage de 0.2 (valeur par défaut) on obtient une courbe d'apprentissage qui converge toujours à zero. La forme de celle-ci dépendra essentiellement des valeurs initiales des poids.

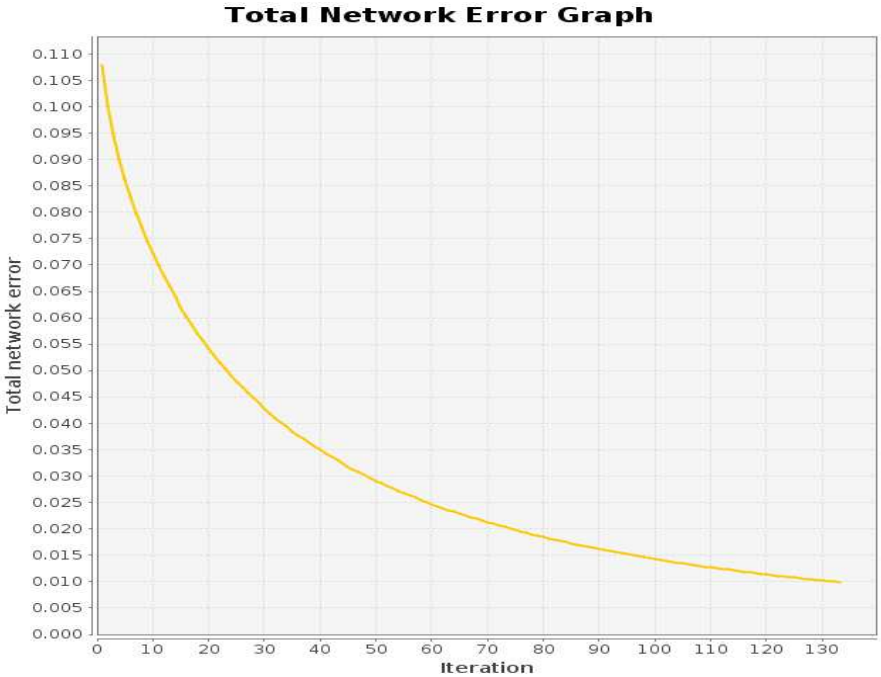

FIGURE 3 – Exemple de courbe d'apprentissage pour la fonction "et"

La figure 3 montre un exemple de courbe d'apprentissage obtenu avec les valeurs susmentionnées. L'apprentissage peut être pensé comme le processus pendant lequel le réseau de neurons cherche de s'approcher le plus possible du plus petit taux d'erreur.

En effet, la meilleure configuration (valeurs des poids, biais, etc.) pour notre réseau de neurones correspond au point de minimum globale de la fonction E qui associe à chaque configuration de notre réseau de neurones un coût C.

E mesure la différence entre le valeur attendu et celle produite et est souvent calculé comme étant $E(y, y') = \frac{1}{2}\|y - y'\|^2$ ou y est le valeur attendu et y' est la valeur produite par le réseau. Mais pourquoi $\|y - y'\|$ ne suffit simplement pas ? En effet, la choix d'utiliser une fonction quadratique n'est pas un hasard. Un des problèmes les plus importants dans l'apprentissage supervivée d'un réseau de neurones qui se base sur l'algorithme du gradient (gradient descent) est celui posé par les points de minimum locales de E. Le fait que E soit quadratique permet d'exploiter la convexité des fonctions quadratique pour minimiser ce probleme. Le $\frac{1}{2}$ permet de simplifier la fonction lors de sa dérivation.

Comme attendu, dans notre cas le perceptron n'a aucun problème à trouver le minimum globale de E, mais dans ce premiere exemple on se contente d'entrainer le réseau de neurones jusqu'à obtenir un taux d'erreur moyen de seulement 1%. La figure 4 montre les résultats de l'entrainement :

```
Input: 0; 0;  Output: 0.0061; Desired output: 0;  Error: 0.0061;
Input: 1; 0;  Output: 0.1401; Desired output: 0;  Error: 0.1401;
Input: 0; 1;  Output: 0.1445; Desired output: 0;  Error: 0.1445;
Input: 1; 1;  Output: 0.8185; Desired output: 1;  Error: -0.1815;
Total Mean Square Error: 0.01837336781134471
```

FIGURE 4 – Test du perceptron après l'apprentissage de la fonction "et"

En utilisant la definition de E on peut vérifier que le coût moyen est bien proche de 0.01 :

$$\begin{aligned}&\frac{1}{4} \times (E(0, 0.0061) + E(0, 0.1401) + E(0, 0.1445) + E(1, 0.8185)) \\ &\quad = \frac{1}{4} \times (\frac{0.0061^2}{2} + \frac{0.1401^2}{2} + \frac{0.1445^2}{2} + \frac{0.1815^2}{2}) = 0.009185965\end{aligned}$$

Le resultat obtenu est très proche de celui souhaité, en effet il suffit d'introduire un seuil pour obtenir un output qui soit à 0 ou à 1. Dans Neuroph cela se traduit par l'utilisation de la fonction d'activation step pour le neurone d'output.

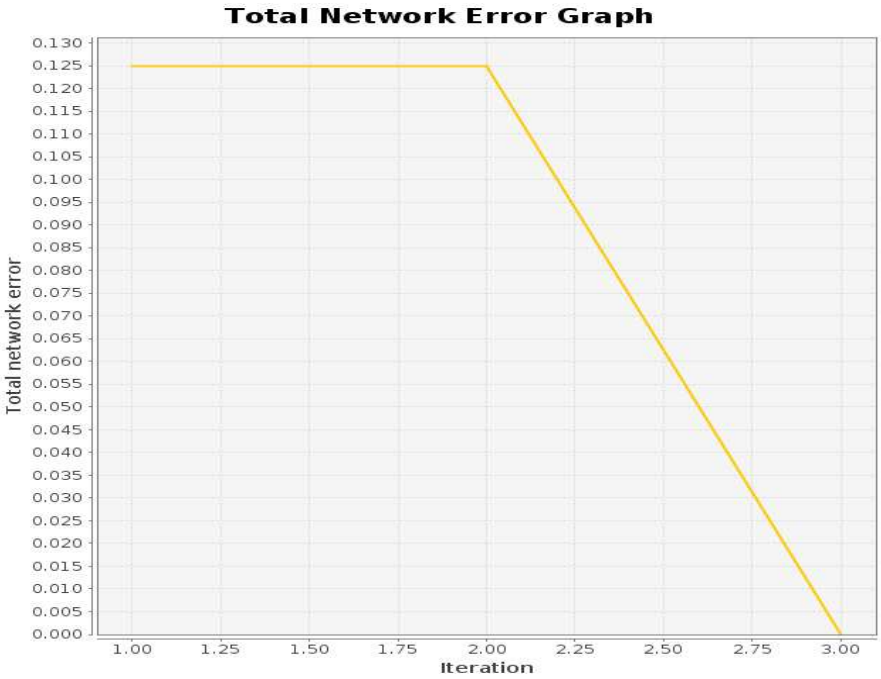

FIGURE 5 – Courbe d'apprentissage de "et" en utilisant la fonction step

Cette technique nous permet d'obtenir une sortie parfaite qui ne présente aucun taux d'erreur.

```
Input: 0; 0;  Output: 0; Desired output: 0;  Error: 0;
Input: 1; 0;  Output: 0; Desired output: 0;  Error: 0;
Input: 0; 1;  Output: 0; Desired output: 0;  Error: 0;
Input: 1; 1;  Output: 1; Desired output: 1;  Error: 0;
Total Mean Square Error: 0.0
```

FIGURE 6 – Test d'erreur après l'apprentissage de la fonction "et"

1.1.2 Differents pas d'apprentissage

Le pas d'apprentissage détermine la vitesse (pas) avec laquelle notre réseau cherche à s'approcher du point d'erreur minimum. A chaque époque (*batch learning*) ou itération (*incremental learning*) les poids du réseau sont mise à jour en utilisant une combinaison du pas d'apprentissage et du gradient ∇E ce qui indique la direction du taux d'accroissement le plus élevé de E et sa pente.

Le pas d'apprentissage joue un rôle très important dans la recherche du minimum globale d'une fonction. Si un pas d'apprentissage est trop grand, cela risque de ne pas atteindre le minimum globale. Inversement, un pas trop petit augmente le temps nécessaire pour entrainer le réseau et risque de se bloquer dans un point de minimum locale.

Dans notre cas le changement du pas d'apprentissage n'a aucun effet que celui de changer la vitesse d'apprentissage.
Avec un pas de 1 (figure 8) l'erreur totale du perceptron tombe en-dessous de 0.01 après seulement 17 itérations. En revanche avec un pas de 0.1 le nombre d'itérations nécessaires pour obtenir le même résultat monte à plus de 155 itérations (figure 7).

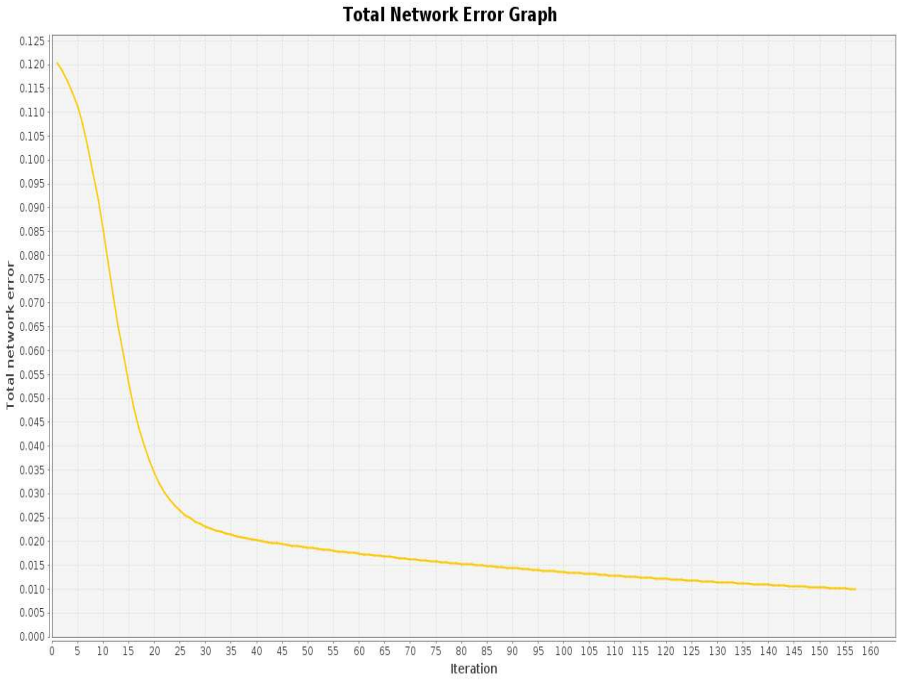

FIGURE 7 – Apprentissage fonction "et" avec un pas de 0.1

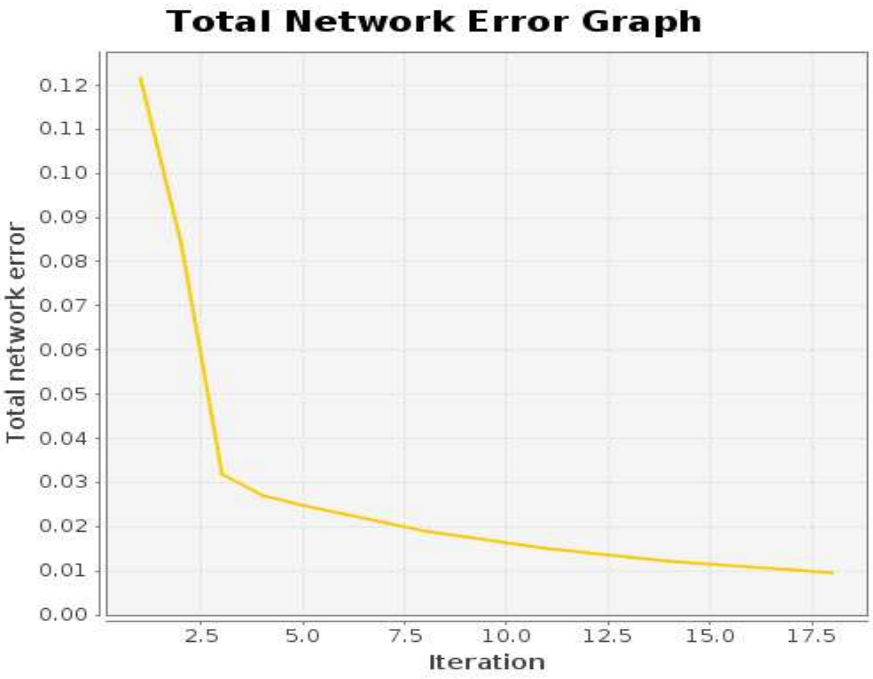

FIGURE 8 – Apprentissage fonction "et" avec un pas de 1

1.2 Equivalence logique

En mathématique deux propositions sont dites équivalentes si elles s'impliquent l'une l'autre : $a \Leftrightarrow b$. La table de vérité de l'équivalence logique est illustrée par la table 2. Cette table de vérité nous servira de data set pour l'entrainement de tout les perceptrons de cette partie.

a	b	$a \Leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

TABLE 2 – Table de verite de $a \Leftrightarrow b$

1.2.1 Un perceptron monocouche pour l'équivalence

Comme pour la fonction booléenne "et" précédente, nous allons essayer de faire apprendre l'équivalence à un perceptron monocouche. Pour cela, nous utiliserons le meme perceptron monocouche que l'on a utilisée pour la fonction "et" (avec deux inputs et un output). Entrainons ce perceptron sur 1000 itérations :

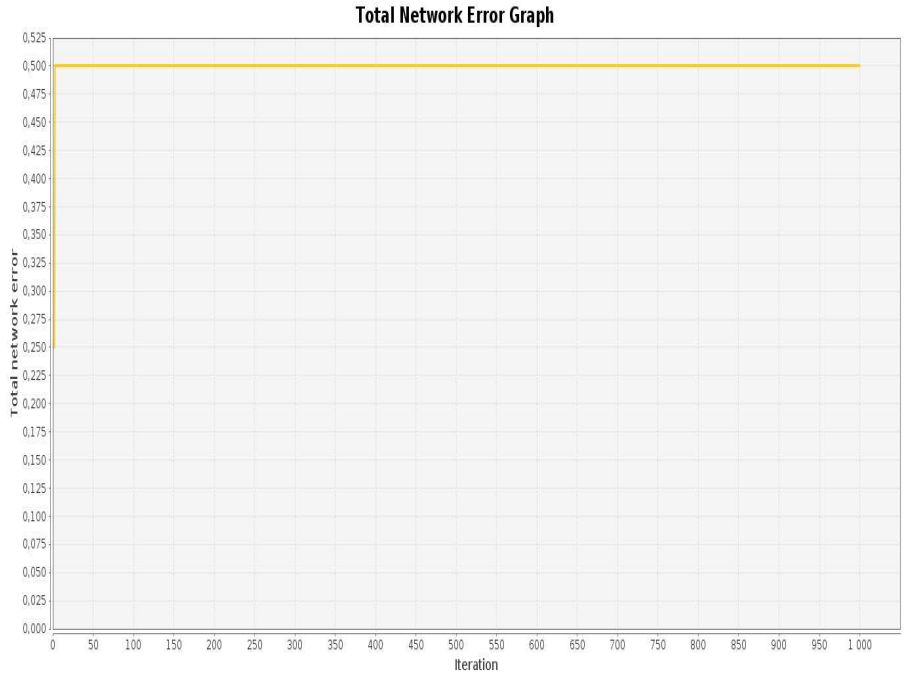

FIGURE 9 – Apprentissage de l'équivalence

Nous pouvons observer que le taux d'erreur du perceptron ne converge pas vers zero en laissant les paramètres par défaut. Nous pouvons changer le pas d'apprentissage pour essayer d'obtenir une convergence. Voyons se que cela donne avec des pas d'apprentissage de respectivement 0.1 , 0.01 et 0.001 (figure 10).
Nous pouvons encore observer que cela ne permet pas de conduire à une convergence. Le perceptron monocouche ne semble donc pas adapté pour l'apprentissage de la fonction d'équivalence. En effet, nous pouvons remarquer que l'équivalence n'est pas une fonction linéairement séparable (figure 11), ceci explique pourquoi un perceptron monocouche ne peut pas apprendre de manière correcte la fonction d'équivalence.

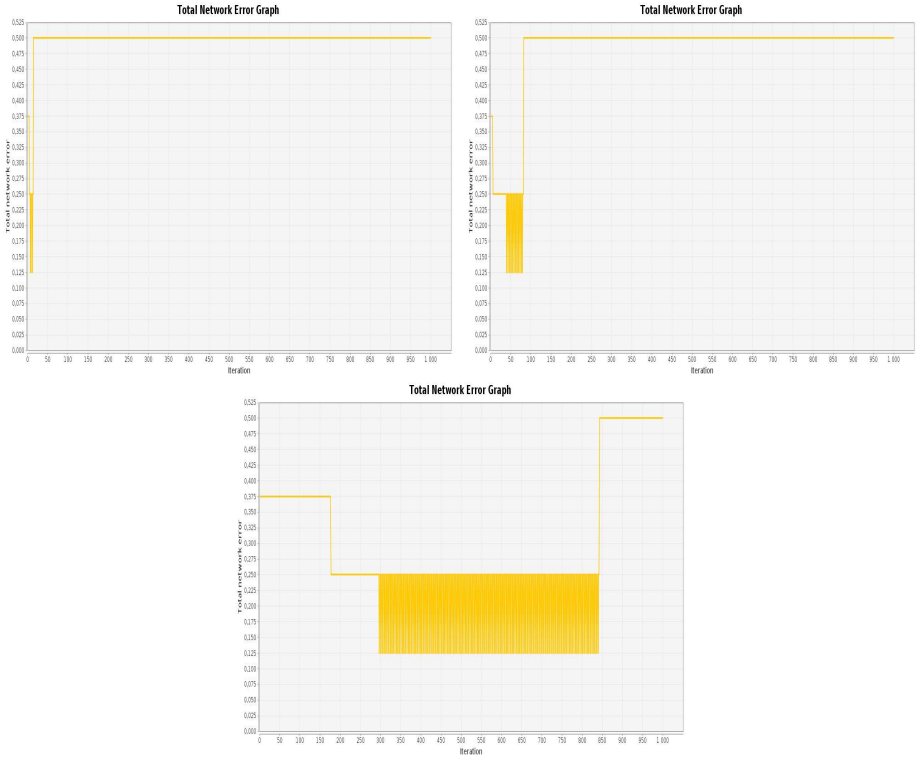

FIGURE 10 – Apprentissage équivalence avec un pas de 0.1, 0.01 et 1

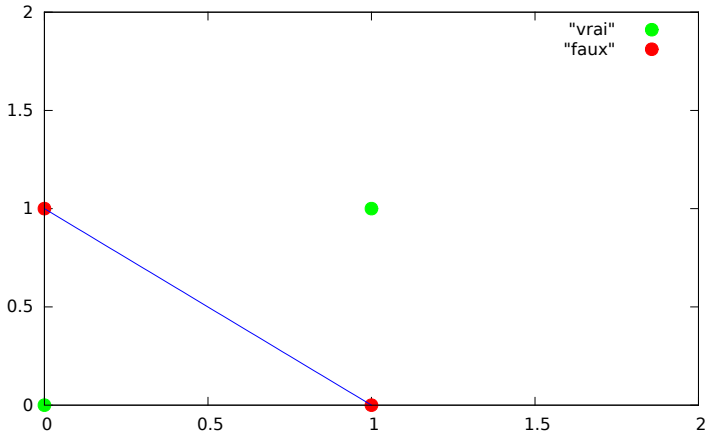

FIGURE 11 – L'équivalence n'est pas linéairement separable

1.2.2 Un perceptron multicouche pour l'équivalence

Pour permettre l'apprentissage de l'équivalence par un perceptron, il faut donc que celui-ci soit un perceptron multicouche. Nous allons donc créer un perceptron multicouche qui contiendra une couche cachée de 3 neurones, et avec toujours 2 inputs et un un output (figure 12).

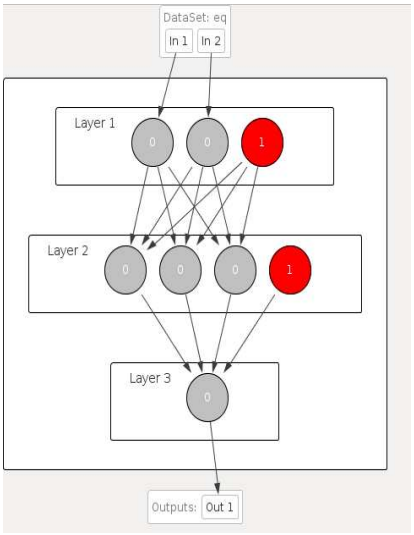

FIGURE 12 – Perceptron multi-couche

Testons ce perceptron avec les paramètres par défaut :

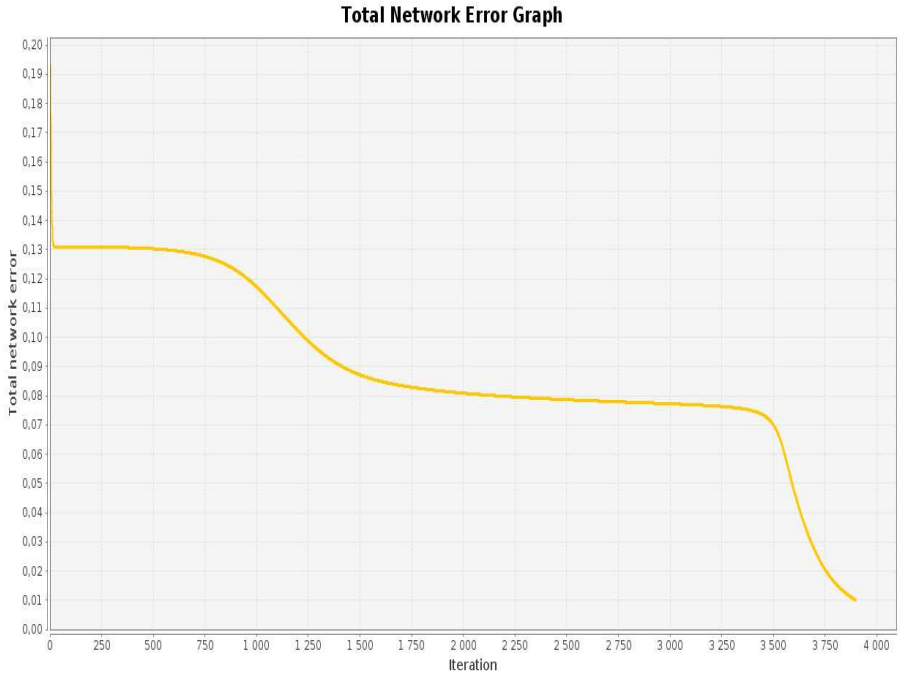

FIGURE 13 – Apprentissage équivalence avec perceptron multi-couche

Notre perceptron multi-couche converge en dessous d'un taux d'erreur de 0.1 après environ 3900 itérations. Nous allons maintenant voir comment le changement de certains paramètres influence la courbe d'apprentissage. Voyons tout d'abord les effets d'un changement du pas d'apprentissage en essayant avec des pas d'apprentissages de respectivement 0.1 , 0.5 et 1 (figure 14).

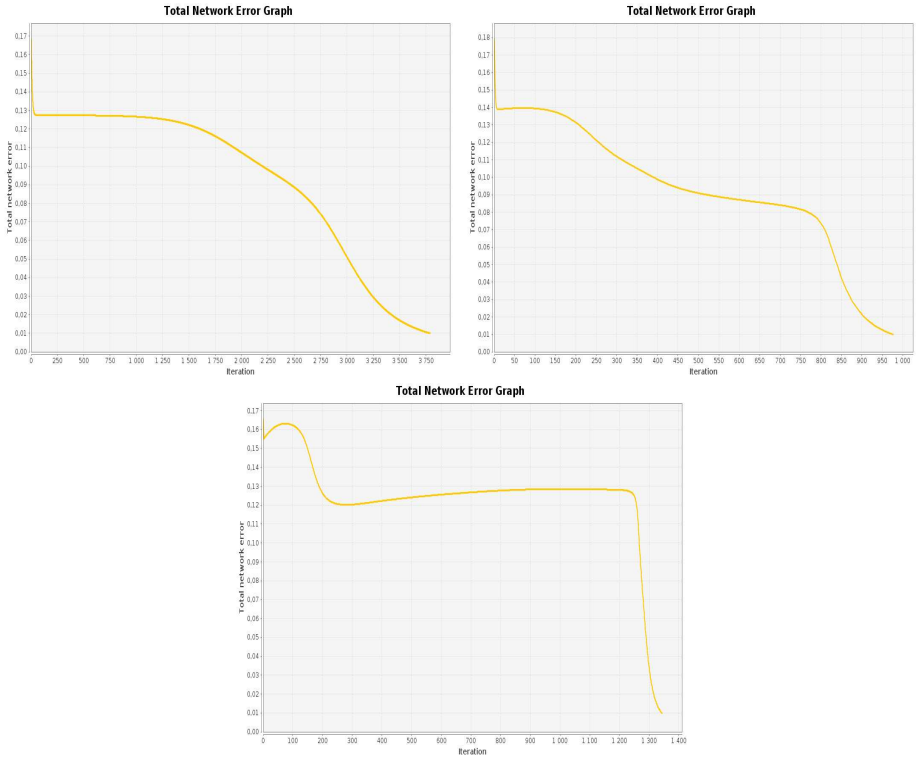

FIGURE 14 – Apprentissage équivalence avec un pas de 0.1, 0.5 et 1

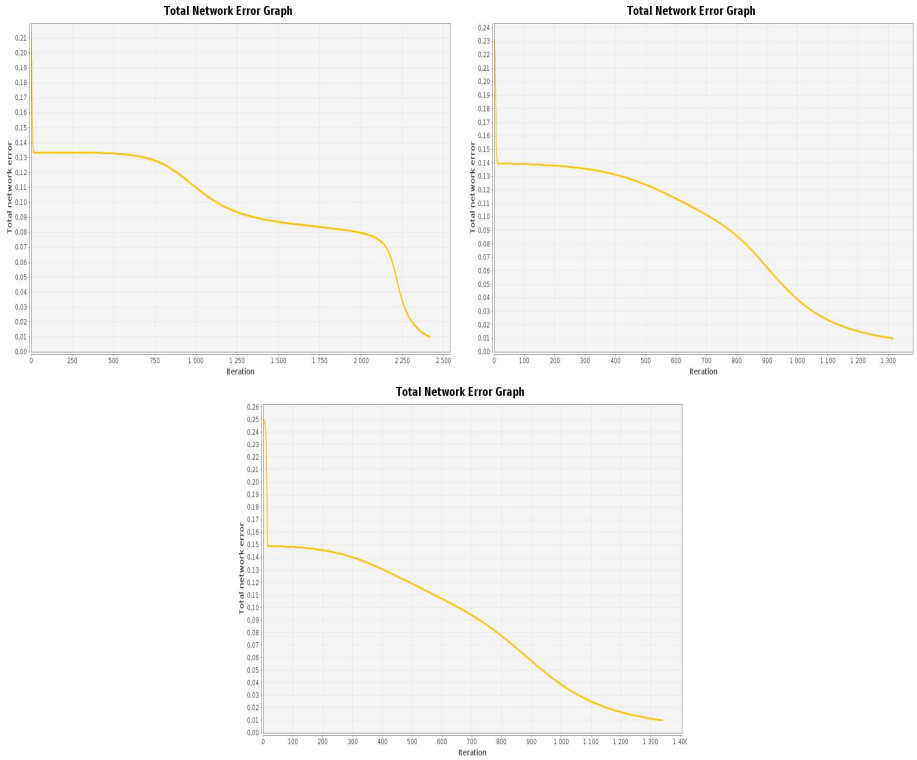

FIGURE 15 – Apprentissage équivalence avec 5, 10 et 20 neurones cachés

Ces 3 graphiques permettent de mettre en évidence le faite que plus le pas d'apprentissage est grand, plus les variations ont de chance d'etre soudaine ; et plus le pas est petit, plus la courbe sera uniforme (avec des variations moins prononcées). Cela correspond bien à nos attentes étant donnée que le pas d'apprentissage est utilisé pour déterminer la vitesse de déplacement sur la courbe d'erreur.
Changeons à présent le nombre de neurone dans la couche caché. Nous allons partir d'un réseau de 5 neurones, et les doubler à chaque fois. Nous pourrons ainsi voir assez rapidement si le nombre de neurone influe grandement sur les résultats. La figure 15 représente donc respectivement les résultats pour des perceptrons de 5, 10 et 20 neurones dans une couche caché. Augmenter le nombre de neurones dans la couche caché semble diminuer le nombre d'itération nécessaire pour que la courbe converge. La courbe semble aussi être plus "uniforme".
Essayons maintenant d'ajouter plus de couche cachées. Voyons la courbe que l'on obtient avec un perceptron de deux couches cachées avec 3 neurones chacune.

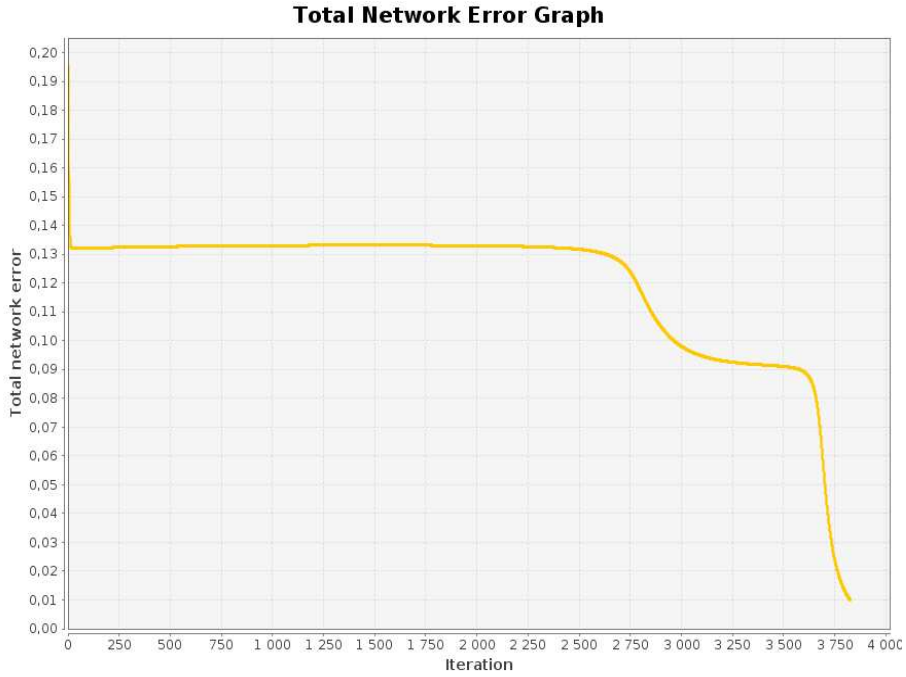

FIGURE 16 – Apprentissage équivalence avec 2 couches cachées de 3 neurones chacunes

Rajouter une couche cachée supplémentaire ne semble pas améliorer le nombre d'itération nécessaire par rapport à un perceptron à une seule couche cachée. Essayons de rajouter plus de neurones dans le couches cachées (figures 17 et 18).Rajouter des neurones dans les couche intermédiaire n'améliore pas le résultat non plus.
Enfin comme vu précédement, le changement de la fonction d'activation par la fonction step nous permet d'introduire un seuil et donc d'obtenir un résultat binaire (figure 19) qui correpsond bien à la nature de notre fonction.

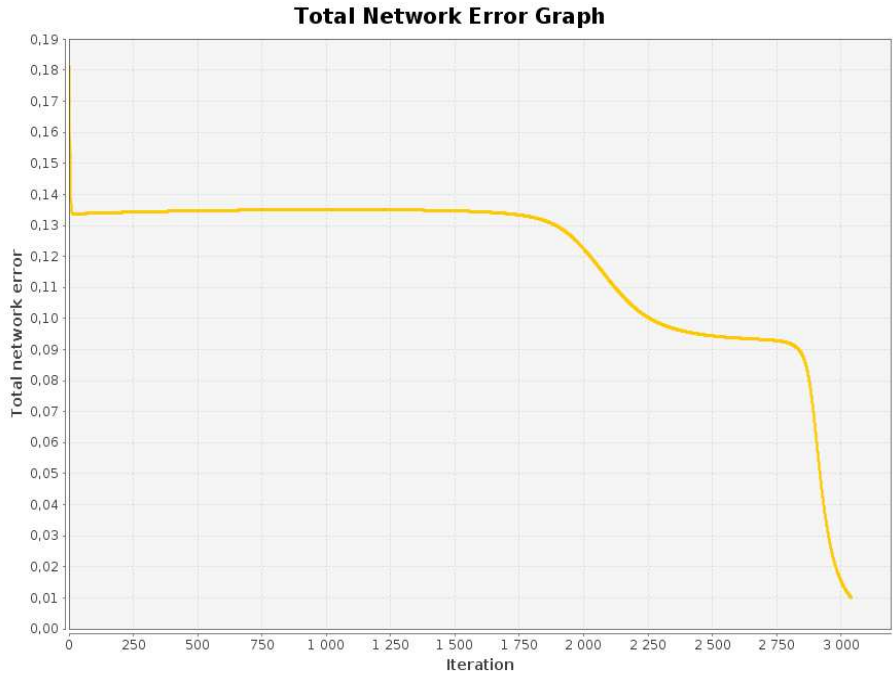

FIGURE 17 – Apprentissage équivalence avec 2 couches cachées de 4 neurones chacunes

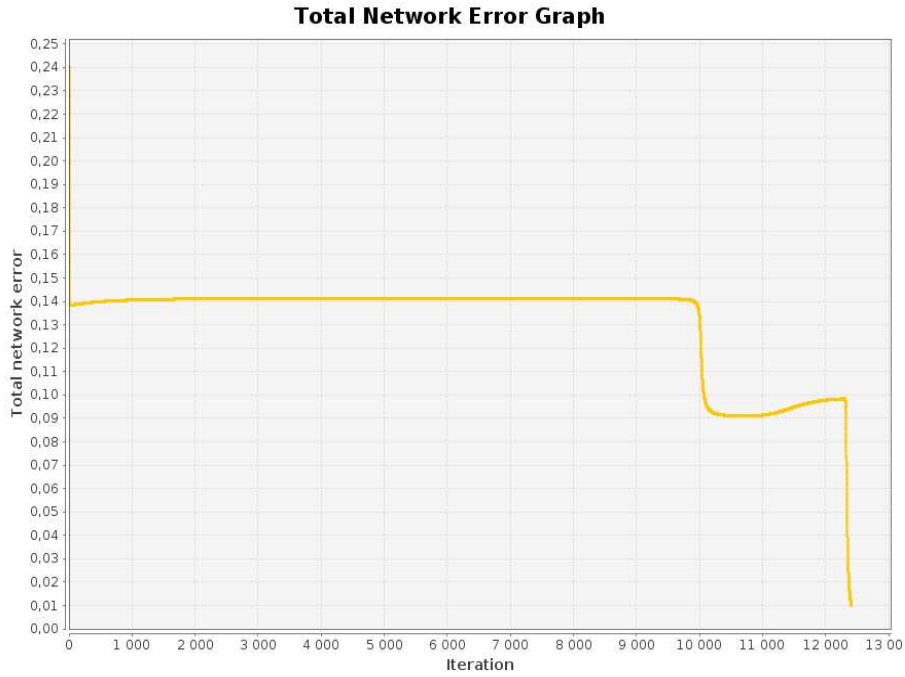

FIGURE 18 – Apprentissage équivalence avec 2 couches cachées de 6 neurones chacunes

```
Input: 0; 0;  Output: 1; Desired output: 1;  Error: 0;
Input: 0; 1;  Output: 0; Desired output: 0;  Error: 0;
Input: 1; 0;  Output: 0; Desired output: 0;  Error: 0;
Input: 1; 1;  Output: 1; Desired output: 1;  Error: 0;
Total Mean Square Error: 0.0
```

FIGURE 19 – Résultat du test avec la fonction d'activation step

2 Apprentissage de la fonction : $f(x) = x^2$

Il est temps de tester si un perceptron (mono-couche ou multi-couches) est capable d'apprendre une fonction un peu plus complexe que celle qu'on a vu jusqu'à présent, la fonction carré : $f(x) = x^2$

Plusieurs différences sont à notées par rapport aux fonctions précédentes. Si avant, le domaine de nos fonctions binaires étaient réduit à $0, 1$, cette fois nous sommes confrontés à une fonction continue définie dans tout $\mathbb{R}$. Une autre différence est liée au type et nombre d'inputs, et change considérablement le paradigme utilisé par notre réseau.
Un concept que nous n'avons pas introduit jusqu'à présent est l'éxistence de plusieur types de données pour l'apprentissage, et autant de paradigmes d'apprentissage qui en découlent. Les deux principaux types de données avec lesquels on doit faire face sont :

- des variables catégorielles (dit aussi qualitative) - où il y a généralement un ou plusieurs cas qui peuvent être regroupés en différentes catégories (par example : haut, bas ou rouge, vert, bleu, etc.)
- des données quantitatives qui se présentes comme une mesure numérique d'un attribut

Le type d'apprentissage qu'on utilise pour reconnaitre et classifier des données appartenant à la première famille est appellé "classification". En revanche l'apprentissage supervisé avec des données quantitatives est appellé "régression".[2]

Pour récapituler : la régression consiste à estimer ou à prédire une réponse, la classification est l'identification de l'appartenance à un groupe.

Dans notre example precedents ("et" logique et équivalence) on a simplement traité des classifications binaires. Cette fois on est confronté avec des données de type quantitative.

2.1 Un perceptron pour la fonction $f(x) = x^2$

Un perceptron n'est pas le moyen plus efficace pour calculer cette fonction, mais peut il le faire ?

Comme nous l'avons vu précédement, un perceptron mono-couche peut être utilisé sans aucun problème pour distinguer des données linéairement séparables, mais il échoue à reconnaitre des schémas un peu plus complexe. Il n'est donc probablement adapté à notre problème.

Le théorème de l'approximation universelle (aussi connu sous le nom de théorème de Cybenko) nous dis qu'un perceptron multi-couches avec une seule couche cachée (et un nombre fini de neurones) peut approximer n'importe quelle fonction continue dans un intervalle de $\mathbb{R}$.[1] [3]
Pour l'apprentissage de la fonction carré nous allons donc utiliser un perceptron multi-couche avec une seule couche cachée et on cherchera à lui faire apprendre la fonction carré dans un intervalle de $\mathbb{R}$.
Pour commencer on pourra utiliser un perceptron avec 3 neurones cachés comme celui que l'on a utilisé précédement pour l'équivalence. Notre but sera d'approximer la fonction carré dans l'intervalle $[1, 100]$.
Dans notre exemple chaque neurone de notre perceptron utilisera la fonction d'activation sigmoide, c'est pour cette raison que l'on normalisera les inputs d'apprentissage pour les borner entre l'intervalle $[0, 1]$ (codomaine de la fonction sigmoide).

1. La démonstration mathématique de ce théorème est assez longue et complexe. Une approche visuelle qui se prête à une demonstration beaucoup plus intuive est proposé à [1].

2.1.1 3 neurones cachés

Pour ne pas surcharger notre perceptron on pourrait commencer à tester ses capacités d'apprentissage avec un dataset de dimensions réduites (table 3). On rappelle que pour avoir un data set compris entre 0 et 1 nous avons normalisé notre data set.

x	$f(x)$
0	0
10	100
20	400
...	...
100	10000

TABLE 3 – Fontion carre : dataset 1

Avec les valeurs par défaut pour l'apprentissage, notre réseau arrive à obtenir un bon taux d'erreur dans un temps raisonable comme le montre la figure 20. Après ce premier test notre réseau nous offre déjà une approximation appréciable de la fonction carré (figure 21).

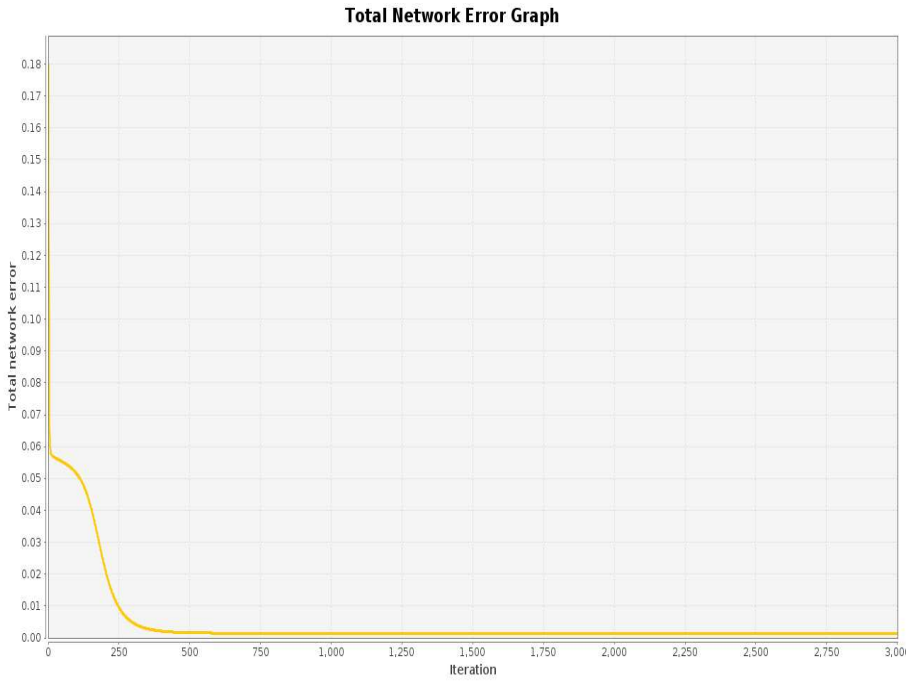

FIGURE 20 – Apprentissage fonction carré : 3 neurons, dataset 1

Maintenant nous pouvons essayer d'augmenter la précision de notre dataset pour essayer d'obtenir une meilleure approximation. Pour notre prochain test on essayera de doubler la précision de notre dataset. Les figures 22 et 23 nous montre une amélioration sensible des performances de notre réseau.
Par contre on se rend très vite compte que notre réseau commence à être aux limites de ces capacités. Après un certain temps, la vitesse d'apprentissage ralentit beaucoup, ce qui implique plusieurs milliés d'itérations pour améliorer de façon significative notre approximations (la figure 24 nous montre les résultats en poursuivant l'entrainement de notre réseau jusqu'à environ 30.000 époques).La fonction d'erreur est déjà assez proche de son minimum globale, c'est donc le moment de passer à un réseau un peu plus grand et donc avec plus de capacites de stockage.

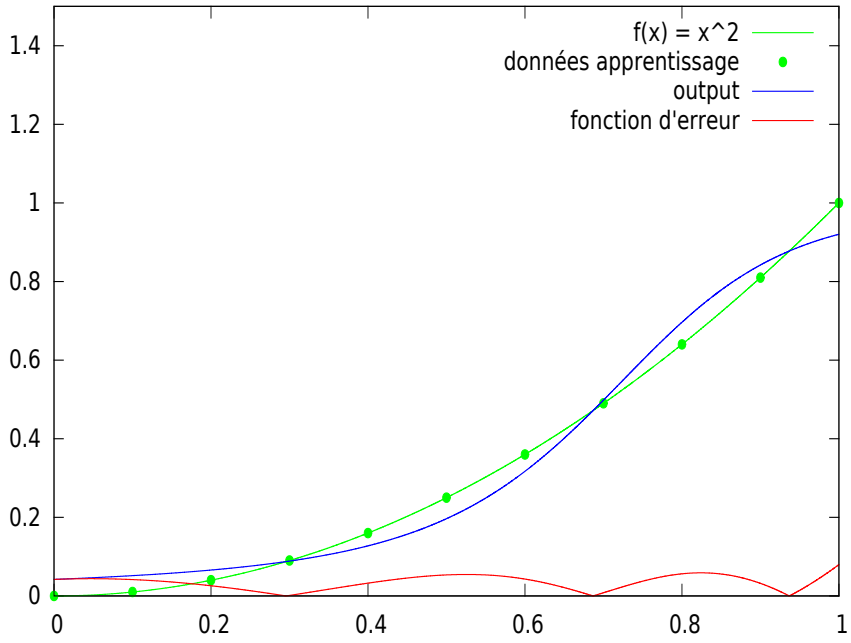

FIGURE 21 – Résultat apprentissage : 3 neurons, dataset 1

2.1.2 4 neurones cachés

Avec un réseau de 4 neurones cachés, plus ou moin 10.000 itérations sont suffisantes pour obtenir un resultat qui dépasse celui obtenu précedement avec une couche de 3 neuronés et 30.000 itérations (figures 25 et 26).
Avec un dataset encore plus grand (100 inputs) et après environ 30.000 itérations, on arrive à obtenir une approximation suffisament précise (figure 27) comme supposé par le théorème de Cybenko : un perceptron multicouche avec une seule couche cachée est suffisant pour approximer la fonction carré (comme n'importe quelle fonction continue) et la précision dépend strictement du nombre de neurones dans la couche caché.

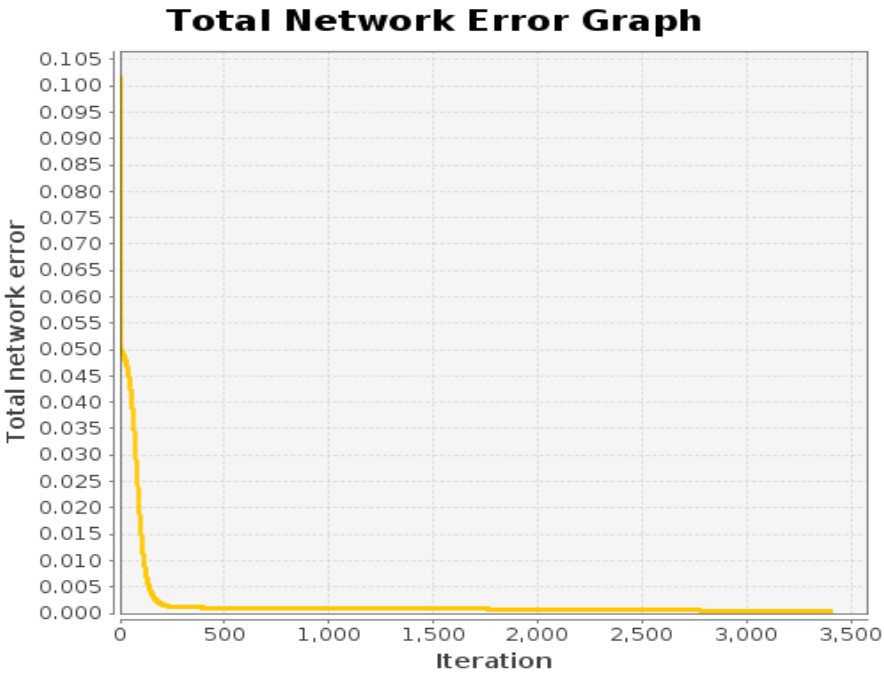

FIGURE 22 – Apprentissage fonction carré : 3 neurons, dataset 2

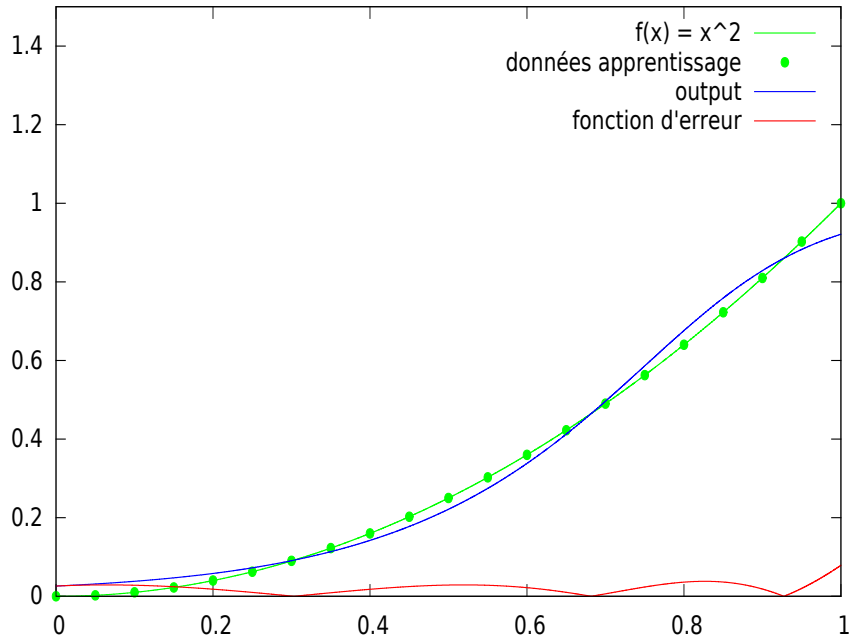

FIGURE 23 – Résultat apprentissage : 3 neurons, dataset 2

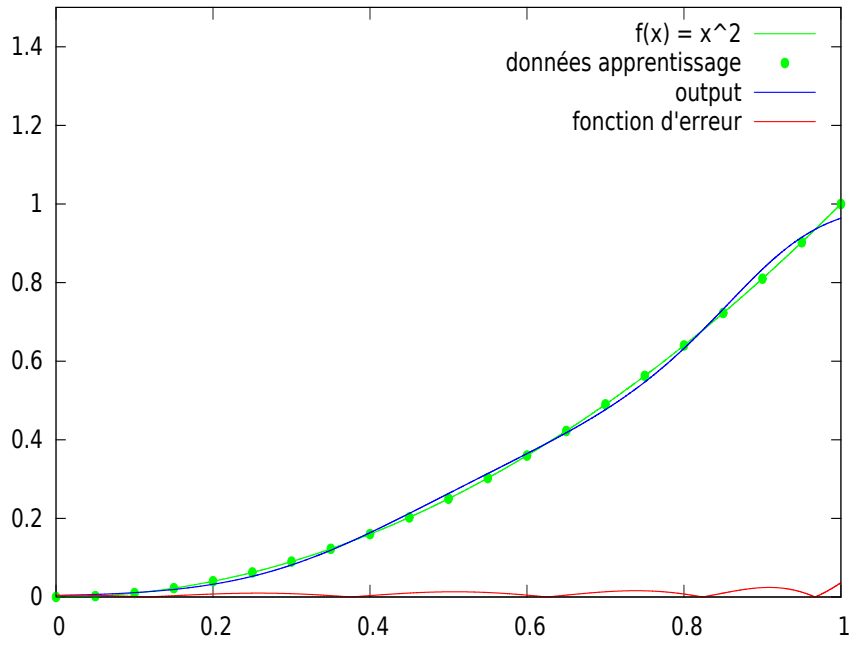

FIGURE 24 – Résultat apprentissage : 3 neurones, dataset 2, 30.000 iterations

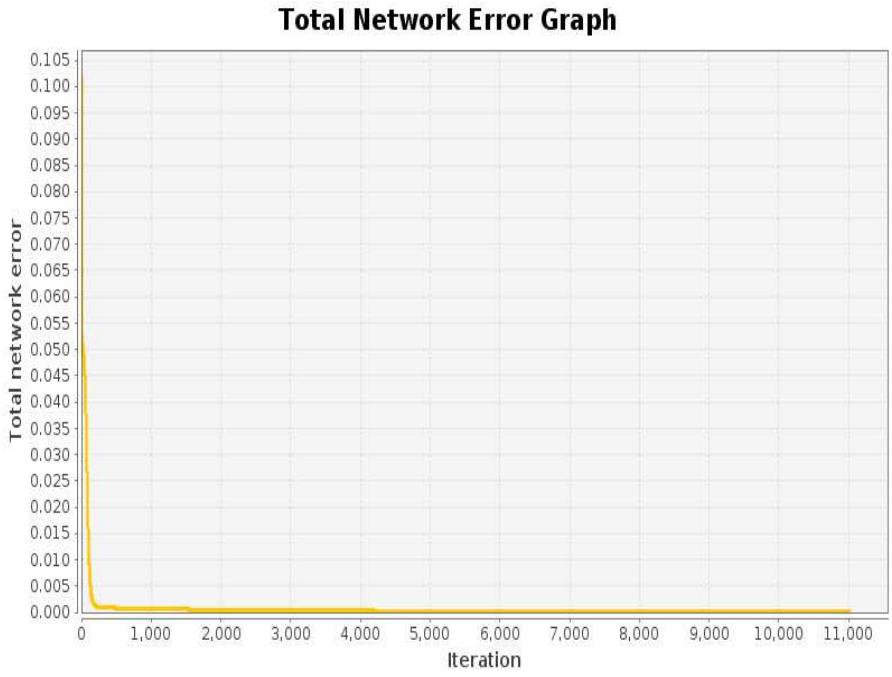

FIGURE 25 – Apprentissage fonction carré : 4 neurones, dataset 2, 10.000 itérations

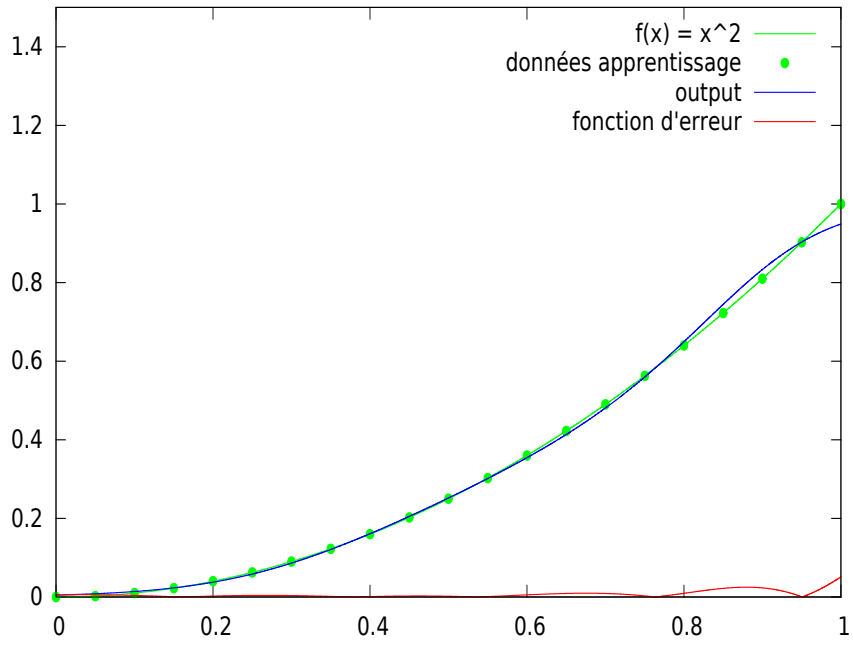

FIGURE 26 – Résultat apprentissage : 4 neurones, dataset 2, 10.000 itérations

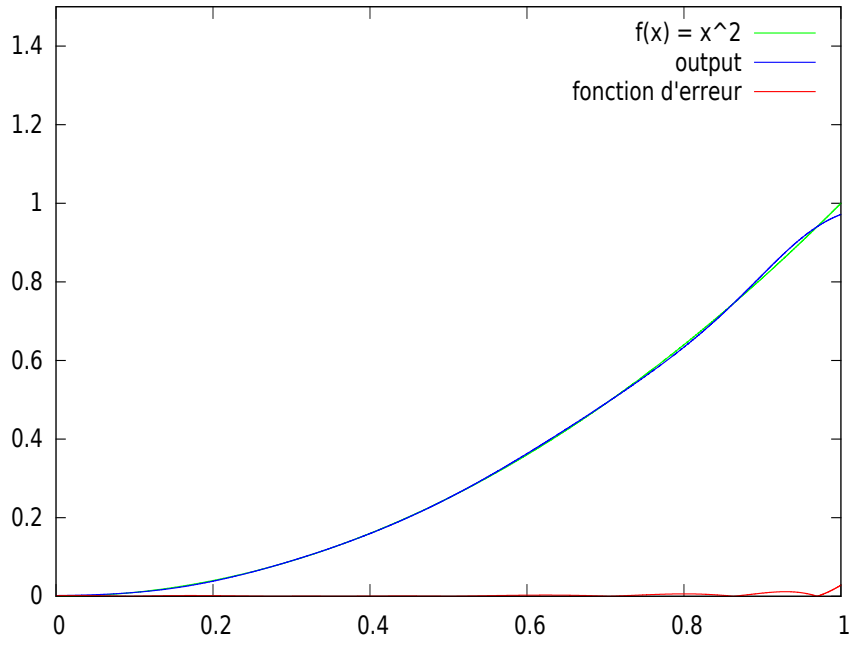

FIGURE 27 – Résultats apprentissage : 4 neurons, dataset 3, 30.000 itérations

3 Conclusion

Comme nous avons pu le constater dans la première partie de ce rapport, les perceptrons pouvent etre utilisés éfficacement dans la classification des données. En particulier un perceptron monocouche est suffisant pour déterminer une catagorisation entre des ensembles de données linéarement séparables.

Nous avons aussi pu voir que les perceptrons pouvent aussi etre utilisés dans le calcul et l'approximation de n'importe quelle fonction continue, ce résultat est résumé par le théorème de Cybenko. Une couche avec plusieurs neurones comporte une capacité de stockage supérieure et permet donc une meilleure approximation.

Pendant l'apprentissage on essaye de s'approcher le plus possible du minumim globale de la fonction d'erreur.

Le pas d'apprentissage détermine la taille de chaque avancement pendant la recherche du minumim dans la fonction d'erreur. Un grand pas d'apprentissage permet d'entrainer le perceptron plus rapidement mais risque de manquer le minimum global. En revanche un petit pas d'apprentissage permet de faire une recherche plus fine et de s'approcher plus précisement du minimum globale. Il risque cependant d'etre bloqué dans un minimum local.

Références

[1] Michael Nielsen. A visual proof that neural nets can compute any function. http://neuralnetworksanddeeplearning.com/chap4.html, 2017.

[2] Warren S. Sarle and Cary. How many kinds of nns exist ? ftp://ftp.sas.com/pub/neural/FAQ.html#A_kinds, 2002.

[3] Yousef Shajrawi and Fadi Abboud. Cybenko's theorem. https://cs.haifa.ac.il/~hhazan01/Advance%20Seminar%20on%20Neuro-Computation/2010/nn1.pdf.