Laurent Prost

TIPE :

Apprendre à un réseau de neurones formels à jouer au Démineur

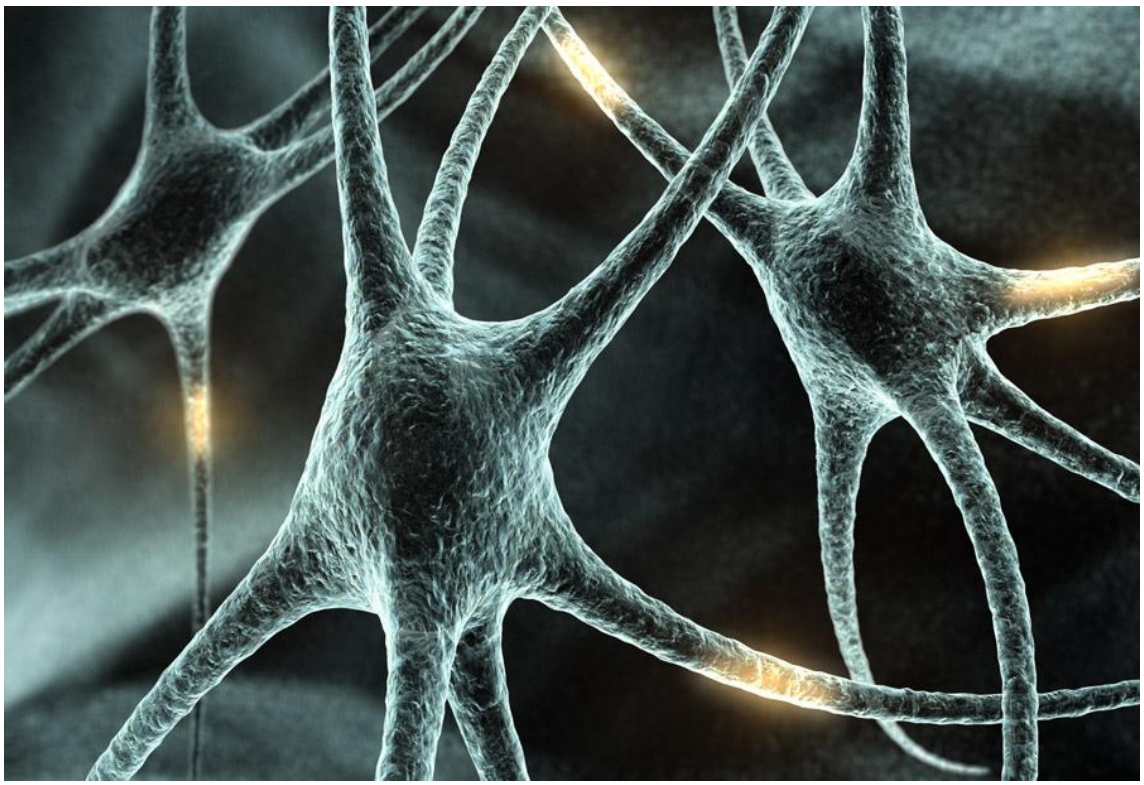

Table des matières

1 Introduction

Dans le cadre des TIPE, j'ai choisi de m'intéresser aux réseaux de neurones formels, et je me suis plus précisément attaché à réaliser un programme permettant d'entraîner un réseau de neurones formels à jouer au Démineur, un jeu livré avec tous les PCs fonctionnant sous Windows. L'objectif de cet exposé n'est pas de démontrer la supériorité des réseaux de neurones formels sur les algorithmes classiques, car le problème du Démineur se résout relativement bien de manière algorithmique, mais plutôt de découvrir sur un exemple original quelques types de réseaux et d'algorithmes d'apprentissage.

2 Le concept de réseau de neurones formels

Avant toute chose, introduisons rapidement l'objet principal de notre étude : le réseau de neurones formel. Prenons pour cela un exemple très simple de réseau, celui de la *FIG. 1* . Ce réseau possède 3 neurones connectés entre eux : 2 neurones d'entrée (numéro 1 et 2) qui reçoivent les informations de l'utilisateur et les transmettent, et 1 neurone de sortie (numéro 3), qui reçoit les informations des autres neurones et calcule la sortie du réseau. Il existe également des réseaux (comme celui de la *FIG. 16*) possédant des neurones entre l'entrée et la sortie : ces neurones sont dits "neurones cachés" et leur ensemble forme la "couche cachée" du réseau. Les connexions entre les neurones sont orientées, ce qui permet de parler d'entrées et de sorties pour chaque neurone. Le neurone 3 possède ainsi deux entrées et une sortie. On affecte de plus un poids à chacune des connexions.

Comment calculer la sortie d'un neurone en fonction de ses entrées ? Lorsqu'il s'agit d'un neurone d'entrée, comme les neurones 1 et 2 qui reçoivent les valeurs de l'utilisateur, la sortie du neurone est simplement égale à son entrée. La sortie de 1 est donc x, et celle de 2 est y. Sinon, pour un neurone comme le neurone 3, la sortie est une fonction de la somme des sorties de 1 et 2 pondérée par les poids des connexions. En clair, la sortie de 3 est égale à $f(w_{13}x + w_{23}y) = f(x + y)$, où f est une fonction dite fonction d'activation. La fonction choisie ici est la fonction seuil, qui vaut 1 sur les nombres strictement positifs et 0 sur les nombres négatifs.

La connexion sortant de 3 ne pointe vers aucun autre neurone : la sortie du neurone 3 est donc la réponse du réseau aux entrées x et y, et on vérifie donc que ce réseau calcule donc le OU logique : si x et y sont des éléments de $[0, 1]$, la sortie vaut 0 si et seulement si $x = y = 0$.

Pour faire répondre un réseau donné à un problème, on a besoin d'un algorithme d'apprentissage, qui à partir d'exemples détermine quels sont les meilleurs poids. Ces algorithmes seront étudiés dans la suite.

3 Généralités sur le Démineur et les réseaux

3.1 Règles du jeu

Le principe du jeu du Démineur est très simple : le jeu se présente sous la forme d'une grille, non dévoilée au début de la partie (cf *FIG. 2*). Le joueur clique sur une case, ce qui a pour effet de dévoiler quelques cases (cf *FIG. 3*). Il doit ensuite utiliser les informations données par les cases dévoilées pour essayer de deviner où sont les mines : sur chaque case est indiqué le nombre de mines dans les cases entourant directement cette case. Lorsqu'il a repéré une mine, il pose un drapeau dessus en faisant un clic droit (cf *FIG. 4*) et peut dévoiler les cases où il est sûr qu'il n'y a pas de mine. Le jeu s'arrête soit lorsque le joueur a cliqué sur une mine (dans ce cas le joueur perd), soit lorsque toutes les mines ont été correctement repérées (dans ce cas le joueur gagne).

3.2 Les modalités d'utilisation des réseaux

Idéalement, le réseau qui joue au Démineur devrait prendre en entrée une grille pour ressortir les coordonnées d'une case à jouer et une action à y effectuer (clic gauche ou droit). Cependant, cette approche présente plusieurs inconvénients majeurs : elle ne fonctionne que pour une grille de taille donnée, elle demanderait une structure certainement très complexe donc un apprentissage très long, les choses étant encore compliquées par le fait qu'il n'y a pas de solution unique au problème "quelle case jouer ?".

Nous suivrons donc une approche moins ambitieuse : pour le réseau, il s'agira uniquement de déterminer si, dans une case donnée, il y a une mine ou pas. Pour cela, nous mettrons en entrée du réseau les deux rangées de cases entourant la case, ces informations étant le plus souvent suffisantes pour déterminer si la case contient une mine.

Cependant, on ne peut pas toujours déterminer à coup sûr le contenu d'une case. Etant donné qu'il est plus aisé d'utiliser des réseaux à sortie binaire, nous utiliserons deux réseaux. Le premier réseau, le "*réseau oui*", doit répondre "oui" s'il est sûr que la case contient une mine, "non" sinon. A l'inverse, le "*réseau non*" doit répondre "oui" s'il est sûr que la case ne contient pas de mine, et "non" s'il n'est pas sûr. En combinant les réponses des deux réseaux, on peut donc avoir une réponse plus fiable qu'avec un seul.

4 Le programme de jeu

Pour entraîner et utiliser les réseaux, je me suis attaché à concevoir un programme permettant à un utilisateur de jouer au Démineur avec l'"assistance" des réseaux : il peut, au choix, jouer selon son idée, et donc fournir des exemples aux réseaux qui apprennent, ou alors demander l'avis des réseaux sur une case de son choix, et jouer en conséquence.

4.1 Choix du langage

J'ai dans un premier temps essayé de réaliser le programme en Maple, mais pour des raisons pratiques (jouer en entrant les coordonnées de la case à jouer est extrêmement lent et contraignant, surtout lorsqu'il s'agit de fournir de nombreux exemples) je me suis tourné vers la combinaison HTML/JavaScript/PHP/MySQL, qui permettait de construire facilement une interface avec l'utilisateur beaucoup plus efficace. Le programme de jeu se présente donc sous la forme d'une page web dont l'intégralité du code source est en annexe de cet exposé.

4.2 Les fonctionnalités

Avant tout, le programme permet de jouer au Démineur de manière tout à fait normale. Il suffit de rafraîchir la page pour commencer une nouvelle partie. En plus des fonctionnalités usuelles, le programme affiche des statistiques sur les réseaux, et à chaque coup joué, des informations sont enregistrées dans la base de données : les poids des deux réseaux, et leur réussite sur le coup qui vient d'être joué, ce qui permet l'établissement de statistiques sur plusieurs parties. On peut également activer une option qui enregistre des exemples en vue d'un apprentissage "automatisé".

L'utilisation du JavaScript a permis de coder de manière simple et pratique l'avis des réseaux : à chaque fois que l'utilisateur pointe sa souris sur une case, deux boîtes de texte affichent l'avis des réseaux sur la case pointée. Un bouton permet de jouer au hasard, ce qui évite aux réseaux de croire qu'on est dans une situation où on peut décider s'il y a une mine lorsque ce n'est pas le cas.

De plus, le programme a été conçu de manière à ce qu'il n'y ait que deux fonctions à changer (l'apprentissage et le calcul) pour changer de type de réseau. C'est pourquoi les fonctions "générales" sont toutes dans un même fichier, et les fonctions spécifiques à chaque réseau dans un fichier à part.

Tout ceci permet d'entraîner les réseaux de la manière la plus naturelle possible : l'apparence du jeu est exactement similaire à celle du Démineur "original", le jeu est aussi rapide (voire plus), ce qui permet de jouer facilement de nombreuses parties.

Voir la *FIG. 5* pour une copie d'écran du programme en marche.

5 Premier essai : le réseau à simple couche

5.1 Structure du réseau

Le réseau est ici le plus simple possible : les neurones d'entrée sont directement reliés à la sortie (cf *FIG. 6* et *FIG. 7*), avec une fonction d'activation "seuil" : la sortie vaut 1 si la somme pondérée des entrées est strictement positive, 0 sinon.

5.2 L'algorithme d'apprentissage

Sur chaque exemple présenté, le réseau commet une erreur (qui peut être éventuellement nulle), et le but de l'apprentissage est de rendre cette erreur minimale. Etant donné que les réseaux apprennent au coup par coup, et non sur une banque d'exemples prédéfinie, il faut, à chaque présentation d'exemple, modifier les poids de manière à réduire l'erreur sur l'exemple donné sans toutefois perdre le bénéfice des apprentissages précédents. Dans ce but, j'ai retenu l'algorithme de Widrow-Hoff, qui s'appuie sur le principe de la descente du gradient :

Etant donnée une fonction f d'une variable réelle x, de classe au moins C^1 et possédant un minimum global, on se propose de construire une suite convergeant vers ce minimum. On choisit ϵ réel, fixe, puis on choisit x_0 au hasard. x_n étant choisi, on pose

$$x_{n+1} = x_n - \epsilon f'(x_n)$$

Essayons de comprendre cette formule (cf *FIG. 8*) : par exemple, lorsque f' est positive, alors la fonction est croissante : il faut donc se déplacer "vers la gauche" (c'est-à-dire choisir $x_{n+1} < x_n$) pour se diriger vers un minimum, et ce d'autant plus que $f'(x_n)$ est grand (car plus on se rapproche d'un minimum, plus la dérivée est petite). Le facteur ϵ sert à "modérer" le déplacement : il ne doit être ni trop petit (sinon la convergence sera extrêmement lente, et on risque de rester "piégé" dans un minimum local), ni trop grand (sinon il y a risque de divergence). Son choix est donc empirique.

On généralise ce principe pour une fonction de plusieurs variables (x est alors un vecteur), avec la règle de modification suivante :

$$x_{n+1} = x_n - \epsilon \overrightarrow{grad}_{x_n}(f)$$

Pour un exemple dont les entrées sont x_i et dont la valeur de sortie attendue est c (pour "consigne"), on définit donc la fonction d'erreur suivante pour le réseau :

$$\boxed{E(w_1, w_2, ..., w_n) = \frac{1}{2}(c - \sum_i w_i x_i)^2}$$

Cette fonction est de classe C^∞ et possède bien un, et même plusieurs, minima globaux (situés sur un hyperplan affine). On peut donc appliquer le principe de la descente du gradient pour modifier chacun des poids. En projetant la relation obtenue plus haut sur chacune des coordonnées, la modification d'un poids donné est :

$$\Delta w_i = -\epsilon \frac{\partial E}{\partial w_i}$$

Comme on a :

$$\frac{\partial E}{\partial w_i} = (c - \sum_i w_i x_i)(-x_i)$$

On obtient la modification :

$$\boxed{\Delta w_i = \epsilon(c - \sum_i w_i x_i)x_i}$$

Remarquons toutefois que la fonction d'erreur n'est pas ici définie directement à partir de la sortie (qui est $f(\sum_i w_i x_i)$, où f est la fonction seuil, ou fonction de Heaviside). Ceci pour une raison très simple : f n'est pas dérivable partout, et lorsqu'elle l'est, sa dérivée vaut 0 ! On considère donc pour l'apprentissage que la sortie est $\sum w_i x_i$, et il suffit alors d'apprendre au réseau à donner comme réponse 1 si sa réponse est "oui", et -1 si sa réponse est "non".

Pourquoi ne prend-on pas directement un n-uplet $(w_1, w_2, ..., w_n)$ situé sur l'hyperplan affine d'équation $\sum w_i x_i = c$ pour minimiser immédiatement la fonction d'erreur ? Il ne faut pas perdre le bénéfice des apprentissages précédents : rendre l'erreur nulle sur un exemple donné peut nécessiter une modification importante des poids, et donc altérer gravement les résultats sur d'autres exemples. Ceci d'autant plus que l'exemple présenté peut être faux ! La méthode de descente du gradient permet donc aux réseaux de neurones d'acquérir une caractéristique qui fait défaut aux algorithmes classiques : ils peuvent s'accomoder des erreurs.

5.3 Résultats

Le réseau a été principalement entraîné sur une grille 10x10 comportant 9 mines, ce qui correspond au mode débutant du Démineur de Windows. Les poids sont tous initialisés à 0 au départ. En "Apprentissage manuel", on relève les statistiques de réussite après environ 1000 coups joués, où les poids sont mis à jour à chaque coup. En "Apprentissage automatique", on fait tourner l'algorithme d'apprentissage un certain nombre de fois sur une base de 500 exemples, puis on joue environ 1000 coups sans modifier les poids obtenus, et on relève les statistiques de réussite sur ces 1000 coups.

$\epsilon = \mathbf{0.01}$:

Divergence très rapide des poids , résultats non significatifs

$\epsilon = \mathbf{0.002}$:

Cette valeur d'ϵ est la valeur limite pour la divergence des poids. Pour cette valeur, les poids divergent encore.

$\epsilon = \mathbf{0.001}$:

En apprentissage manuel :

- Réussite du *réseau oui* : 58%
- Réussite du *réseau non* : 53%
- Réussite des deux réseaux ensemble : 40%
- Echec des deux réseaux ensemble : 29%

L'évolution des poids lors de cet apprentissage est représentée à la *FIG. 9* . On remarque que la courbe est constituée de paliers entrecoupés de sauts brusques, ce qui signifie que les poids n'arrivent pas à se stabiliser.

En apprentissage automatique (après 20 boucles) :

- Réussite du *réseau oui* : 68%
- Réussite du *réseau non* : 43%
- Réussite des deux réseaux ensemble : 35%
- Echec des deux réseaux ensemble : 24%

Curieusement, l'apprentissage en boucle donne des résultats à priori moins bons. Ceci est dû au fait qu'avec les poids issus de l'apprentissage automatique, le *réseau non* pense qu'il n'y a pas de mine si on pointe sur une case d'une grille vide, alors que le *réseau oui* répond (à juste titre) qu'il n'est pas sûr qu'il y ait une mine. On constate cependant (cf *FIG. 10*) que l'apprentissage automatique donne lieu au bout de quelques itérations à une évolution cyclique. Les poids n'arrivent donc toujours pas à se stabiliser.

$\epsilon = \mathbf{0.0001}$:

En apprentissage manuel :

- Réussite du *réseau oui* : 69.5%
- Réussite du *réseau non* : 61%
- Réussite des deux réseaux ensemble : 48%
- Echec des deux réseaux ensemble : 18%

En regardant les résultats bruts, c'est bien meilleur ! Il y a pourtant un "mais" : si le *réseau non* équilibre bien ses réponses, le *réseau oui* répond 98% du temps qu'il n'est pas sûr ! Etant donné qu'on dévoile plus de cases qu'on ne pose de drapeaux, cela fait artificiellement monter les statistiques du réseau. Cependant, l'évolution des poids représentée à la *FIG. 11* suggère qu'il faudrait plus d'exemples.

En apprentissage automatique (après 50 boucles) :

– Réussite du *réseau oui* : 70.7%
– Réussite du *réseau non* : 55.6%
– Réussite des deux réseaux ensemble : 49.5%
– Echec des deux réseaux ensemble : 23%

Les résultats sont sensiblement identiques, cependant on constate le même phénomène que précédemment : avec les poids issus de l'apprentissage automatique, le *réseau non* répond qu'il n'y a pas de mine sur une case d'une grille entièrement non dévoilée, ce qui fait baisser ses statistiques par rapport au *réseau oui*, lequel équilibre enfin ses réponses. Le graphe d'évolution des poids lors de l'apprentissage automatique à la *FIG. 12* montre que cette fois, on semble bien se diriger vers une convergence, et également qu'il était utile de présenter plus d'exemples.

6 Second essai : le réseau à couche cachée

6.1 Intérêt de la couche cachée

Les réseaux à simple couche sont un premier modèle simple, mais ils possèdent malgré tout une limitation importante : ils ne peuvent reconnaître que les échantillons séparables par un hyperplan affine. Un exemple extrêmement simple est celui de la fonction XOR : il est impossible de la calculer à l'aide d'un réseau de neurones formels à simple couche (cf *FIG. 13*), alors qu'un réseau à couche cachée (comme celui de la *FIG. 14*) peut le faire. Tentons donc de rajouter une couche cachée à notre premier réseau !

6.2 Structure du réseau

Etant donné qu'une technique souvent utilisée par les joueurs humains pour repérer si une case possède une mine est de regarder une des 8 cases en "contact" avec la case centrale, de compter le nombre de mines qu'il y a autour de cette case (ce qui fait 7 cases à examiner) et d'en déduire si la case centrale possède une mine ou pas. On forme donc 8 groupes de 8 cases (dont 2 sont représentés à la *FIG. 15*), et on relie chaque groupe à des 8 neurones de la couche cachée. Les 8 neurones de la couche cachée sont à leur tour reliés à la sortie. (cf *FIG. 16*).

6.3 L'algorithme d'apprentissage

Les réseaux à couche cachée introduisent une difficulté nouvelle : il n'est plus possible de contourner le problème de non-dérivabilité de la fonction d'activation seuil comme pour le réseau à simple couche. Il nous faut donc changer de fonction d'activation. On utilise donc une approximation infiniment dérivable de la fonction seuil : la fonction sigmoïde σ définie par

$$\boxed{\sigma(x) = \frac{1}{1+\exp(-x)}}$$

Cette fonction possède également l'avantage d'avoir une dérivée très simple :

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

La fonction d'erreur sur un exemple donné est la même que que celle définie précédemment :

$$\boxed{E(w_1, w_2, ..., w_n) = \frac{1}{2}(c-s)^2}$$

Où c est la valeur attendue (consigne) et s la sortie effective du réseau. En conservant le principe de la méthode du gradient, il nous faut donc évaluer $\frac{\partial E}{\partial w_i}$. Afin de rendre les calculs plus compréhensibles, nous adopterons les notations suivantes :

– w_{ij} représente le poids de la connexion allant du neurone i vers le neurone j

– s_i représente la sortie du neurone i
– e_i représente l'entrée du neurone i ($e_i = \sum_k w_{ki} s_i$).

Etant donné que w_{ij} n'intervient que dans le calcul de la quantité e_j, on a :

$$\boxed{\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial w_{ij}} = \frac{\partial E}{\partial e_j} s_i}$$

Premier cas : Supposons que i soit un neurone de la couche cachée. j est alors le neurone de sortie. On a alors

$$\frac{\partial E}{\partial e_j} = -\frac{\partial s}{\partial e_j}(c - s)$$

Comme

$$\frac{\partial s}{\partial e_j} = \sigma'(e_j) = s(1 - s)$$

On a, en posant $\delta_j = -\frac{\partial E}{\partial e_j} = s(1 - s)(c - s)$:

$$\boxed{\frac{\partial E}{\partial w_{ij}} = -s_i \delta_j}$$

Deuxième cas : Supposons que j est un neurone de la couche cachée. On a alors, comme e_j n'intervient que dans le calcul de la quantité e_{34} (le neurone de sortie est le neurone numéro 34) :

$$\frac{\partial E}{\partial e_j} = \frac{\partial E}{\partial e_{34}}\frac{\partial e_{34}}{\partial e_j}$$

Comme

$$e_{34} = \sum_k w_{k,34} s_k = \sum_k w_{k,34}\sigma(e_k)$$

Où k décrit l'ensemble des neurones de la couche cachée, on a

$$\frac{\partial e_{34}}{\partial e_j} = w_{j,34}\sigma'(e_j) = w_{j,34} s_j (1 - s_j)$$

Nous voyons donc qu'il est nécessaire de calculer $\frac{\partial E}{\partial e_{34}}$, c'est-à-dire $-\delta_{34}$. On a à ce moment :

$$\frac{\partial E}{\partial w_{ij}} = -s_i s_j (1 - s_j) w_{j,34}\delta_{34}$$

Ainsi, en posant $\delta_j = -s_j(1 - s_j) w_{j,34}\delta_{34}$, on retrouve :

$$\boxed{\frac{\partial E}{\partial w_{ij}} = -s_i \delta_j}$$

Remarque : Cette règle de modification se généraliserait dans le cas d'un réseau à plusieurs couches cachées, et où les neurones possèdent plusieurs sorties, en posant

$$\delta_j = -s_j(1-s_j)\sum_k w_{jk}\delta_k$$

Où k décrit l'ensemble des neurones possédant une entrée provenant de j. C'est pourquoi cet algorithme s'appelle l'"algorithme de rétropropagation du gradient" : en effet, pour le mettre en oeuvre, il faut d'abord calculer les δ des neurones de sortie, puis ceux de la première couche cachée, puis ceux de la seconde couche cachée, et ainsi de suite...

Ceci nous permet donc d'énoncer l'algorithme de modification des poids :

- Calculer δ_{34}
- Calculer δ_i pour tout i entre 26 et 33
- Pour toute connexion (i, j), faire $w_{ij} \leftarrow \epsilon s_i \delta_j$

On remarque une fois de plus que si c = s, alors aucune modification n'est effectuée.

6.4 Résultats

Le protocole est le même que pour le réseau à simple couche.

$\epsilon = \mathbf{0.01}$

En apprentissage manuel :

- Réussite du *réseau oui* : 70.8%
- Réussite du *réseau non* : 52.7%
- Réussite des deux réseaux ensemble : 25%
- Echec des deux réseaux ensemble : 1%

Les statistiques sont ici encore à mettre en parallèle avec la variété des réponses des réseaux : les réseaux n'étaient pas sûrs plus de 98% du temps. Ces résultats montrent donc qu'il est nécessaire de présenter bien plus d'exemples, ce que confirme l'examen de l'évolution des poids (cf *FIG. 17*).

En apprentissage automatique (après 200 boucles) :

- Réussite du *réseau oui* : 75.2%
- Réussite du *réseau non* : 70%
- Réussite des deux réseaux ensemble : 62%
- Echec des deux réseaux ensemble : 17%

Les résultats sont ici nettement meilleurs, et cette fois-ci les réseaux varient leurs réponses ! Remarquons cependant que les 200 boucles (soit 100 000 exemples) n'étaient ici pas de trop, sachant qu'avec 100 boucles on observait toujours le même phénomène que précédemment. La *FIG. 18* suggérerait qu'il faudrait faire plus d'itérations pour avoir une meilleure convergence, mais des itérations supplémentaires ne montrent pas d'amélioration significative des résultats, pas plus qu'un changement d'ϵ.

7 Conclusion

Bien que les résultats soient dans un certain sens un peu décevants, car ils ne permettent pas de faire des réseaux suffisament fiables pour jouer de manière autonome, cette étude nous aura malgré tout permis de découvrir sur un exemple le fonctionnement des réseaux de neurones formels et la difficulté de leur mise en oeuvre, notamment au niveau du choix du type de réseau et de l'algorithme d'apprentissage. Mon approche n'est bien entendu pas exempte de failles, et pour améliorer les résultats, on pourrait encore tester l'influence de facteurs comme le codage des informations en entrée du réseau (par quel nombre représenter une case vide ? une case non dévoilée ? ...), essayer d'améliorer l'algorithme d'apprentissage en utilisant un ϵ variable, essayer d'autres structures de réseau... Mais tout cela n'est pas faisable en un seul TIPE !

8 Bibliographie

Références sur les réseaux de neurones :

- http ://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html
- Ben Kröse et Patrick Van der Smagt - An introduction to Neural Networks (University of Amsterdam - 1996)

Références en programmation :

- Le méga livre JavaScript (Sybex)
- Manuel PHP officel (http ://www.php.net/docs.php)

Je me suis aidé de la documentation pour comprendre le principe du réseau de neurones et des algorithmes d'apprentissage. Tout le reste (la mise en oeuvre du programme, des réseaux, les interprétations des résultats...) est strictement personnel.

9 Illustrations

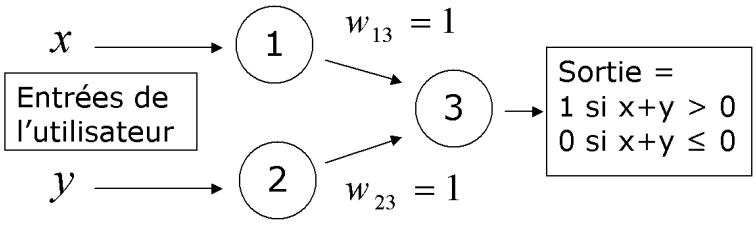

FIG. 1 – Un réseau calculant le OU logique

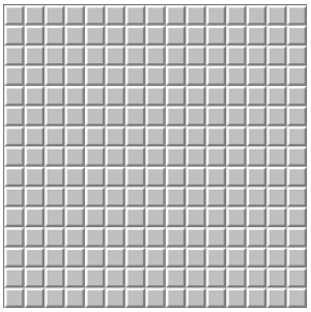

FIG. 2 – Une grille de Démineur vide

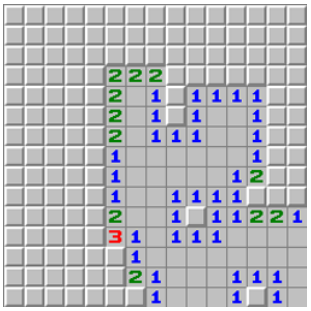

FIG. 3 – Une grille de Démineur après un premier coup

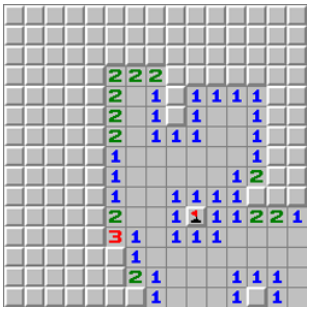

FIG. 4 – La même grille où on a posé un drapeau

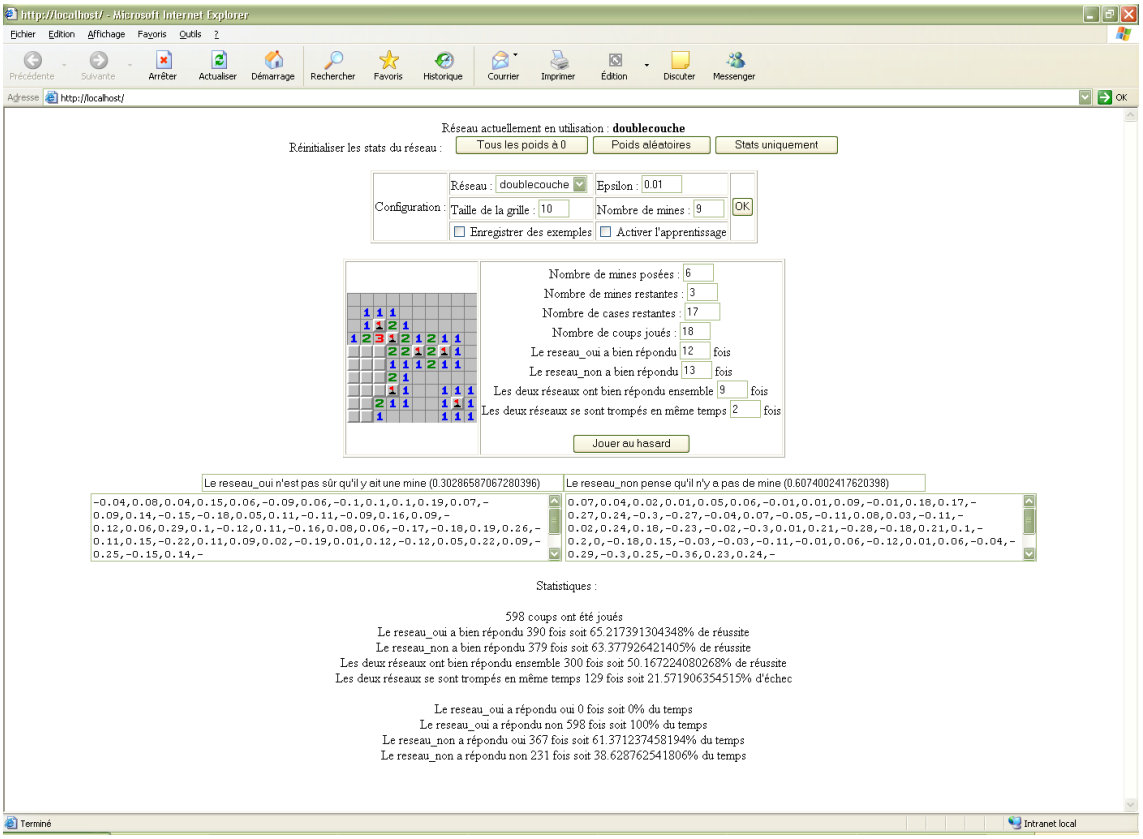

FIG. 5 – Le programme de jeu en action

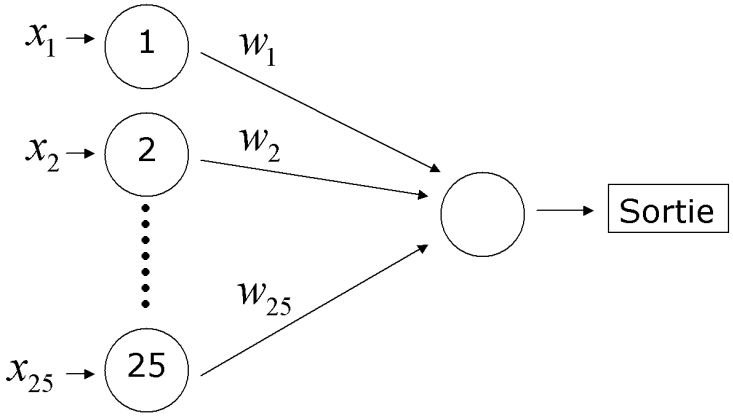

FIG. 6 – La structure du réseau à simple couche

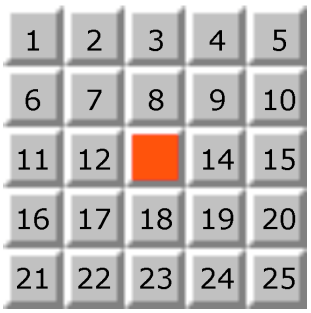

FIG. 7 – La numérotation des cases d'entrée

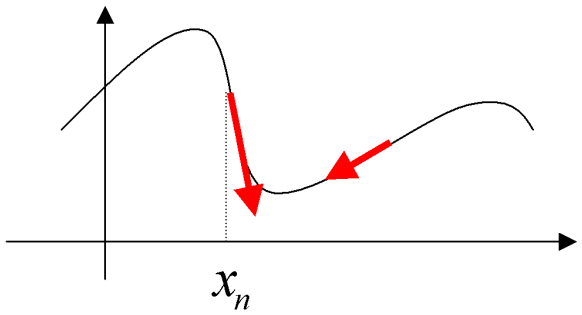

FIG. 8 – La descente du gradient

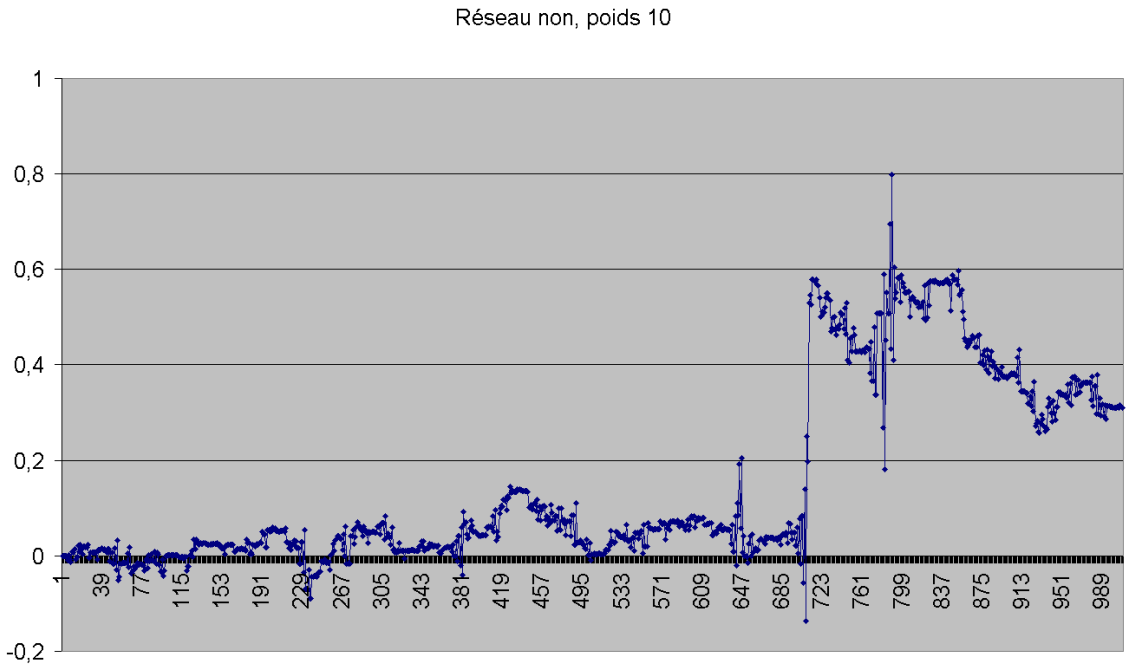

FIG. 9 – Suivi d'un poids du réseau à simple couche pour $\epsilon = 0.001$ pendant l'apprentissage manuel

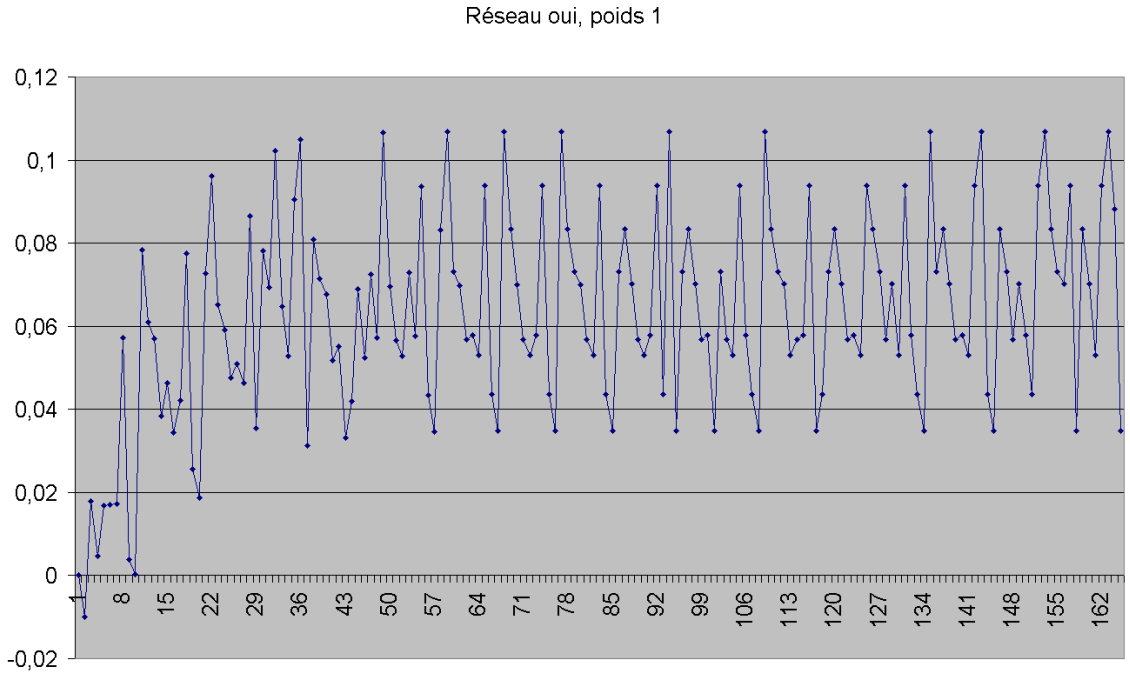

FIG. 10 – Suivi d'un poids du réseau à simple couche pour $\epsilon = 0.001$ pendant l'apprentissage automatique

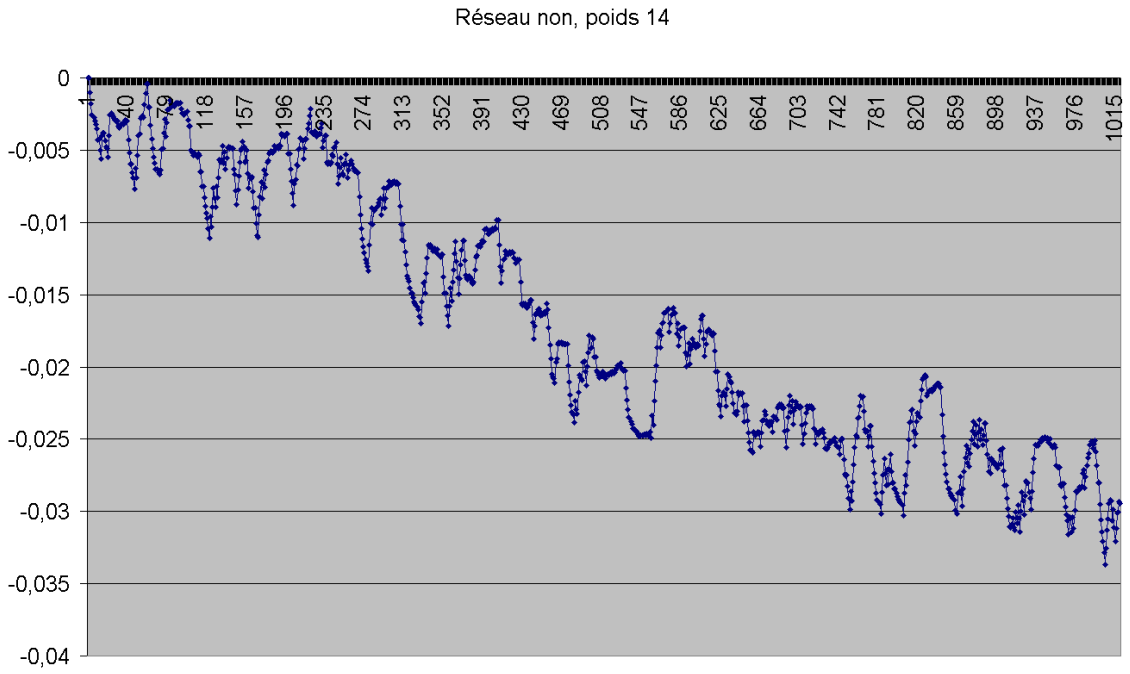

FIG. 11 – Suivi d'un poids du réseau à simple couche pour $\epsilon = 0.0001$ pendant l'apprentissage manuel

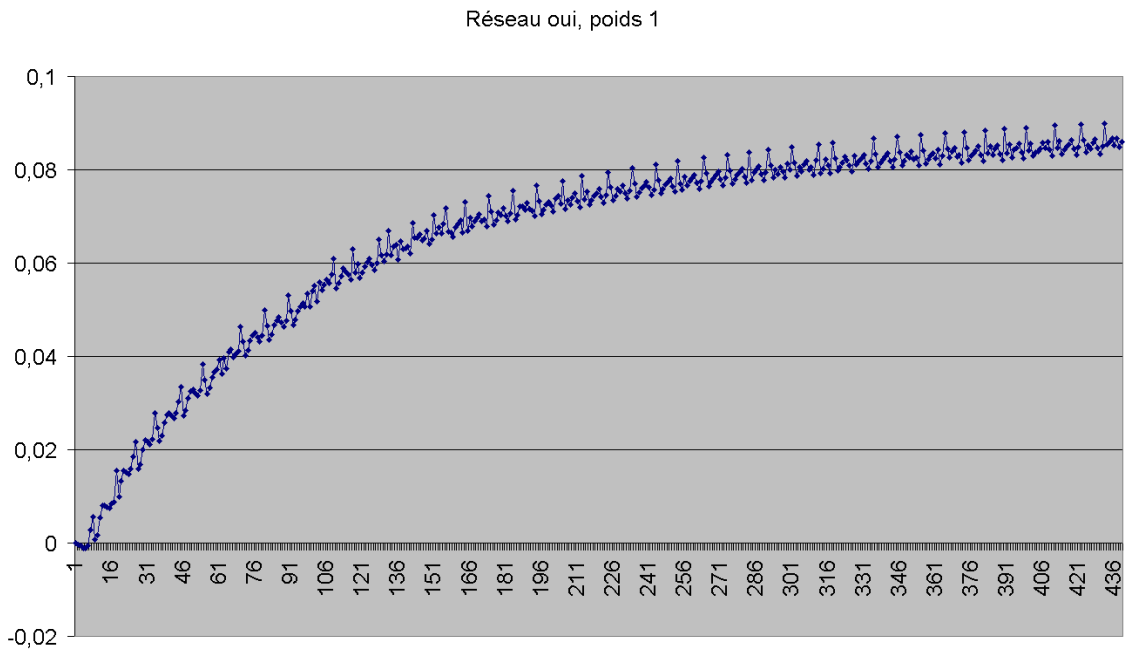

FIG. 12 – Suivi d'un poids du réseau à simple couche pour $\epsilon = 0.0001$ pendant l'apprentissage automatique

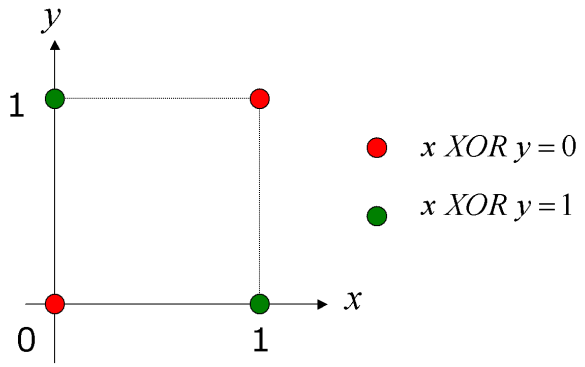

FIG. 13 – L'échantillon du XOR n'est pas linéairement séparable

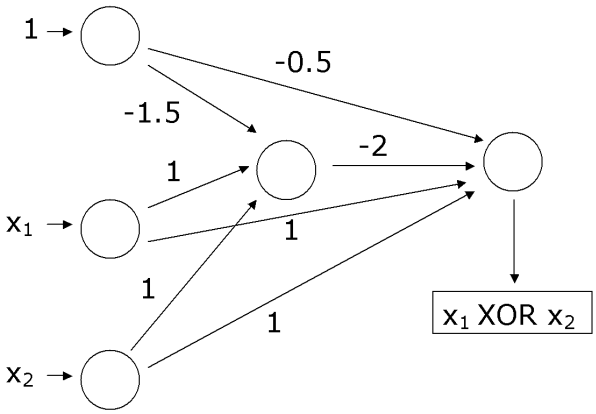

FIG. 14 – Un réseau à couche cachée calculant le XOR

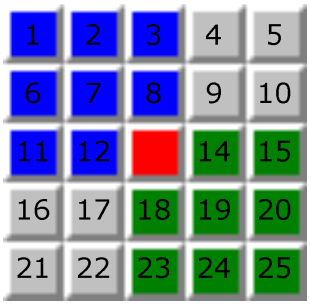

FIG. 15 – Deux des huit groupes de huit cases

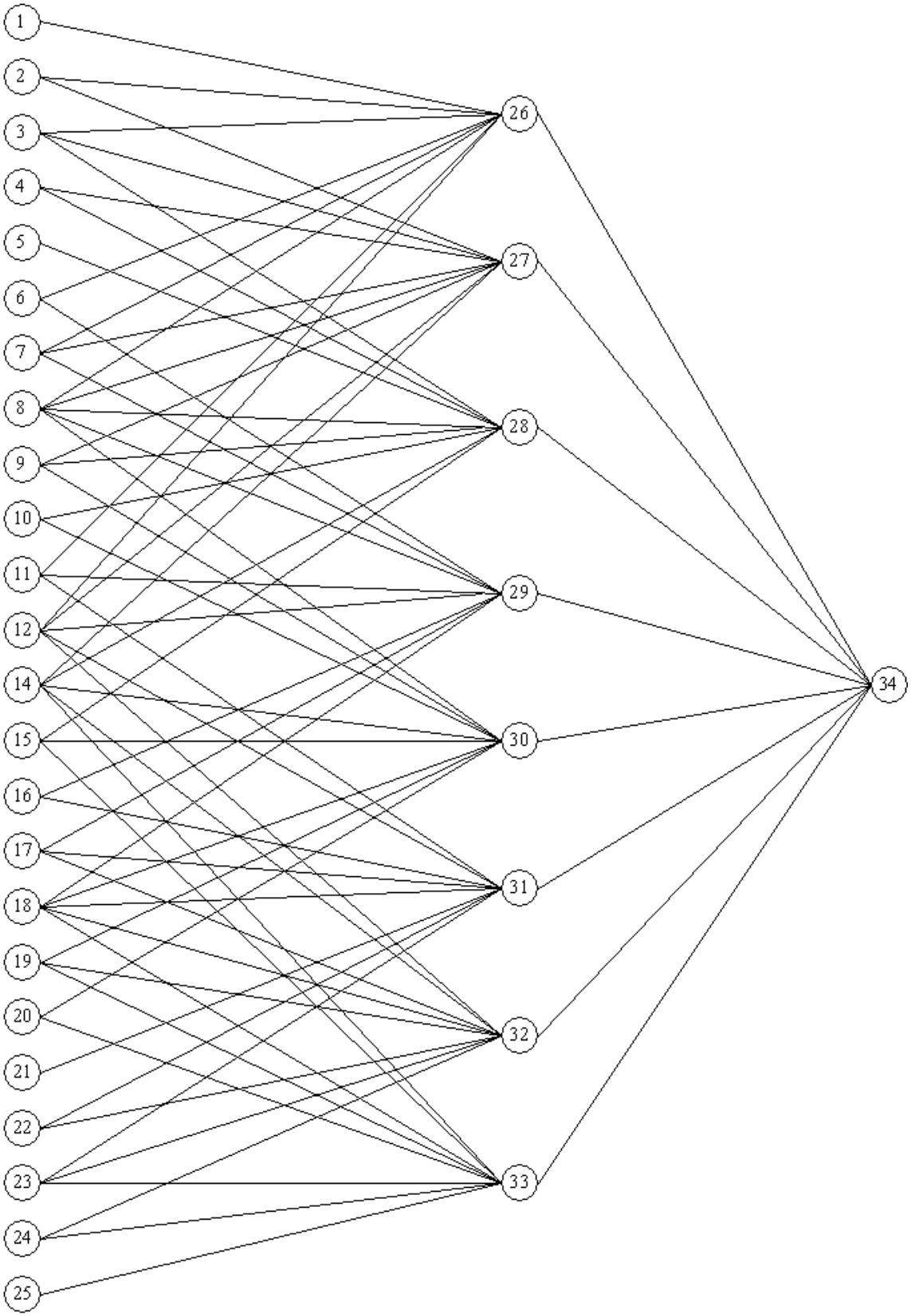

FIG. 16 – Câblage du réseau à couche cachée

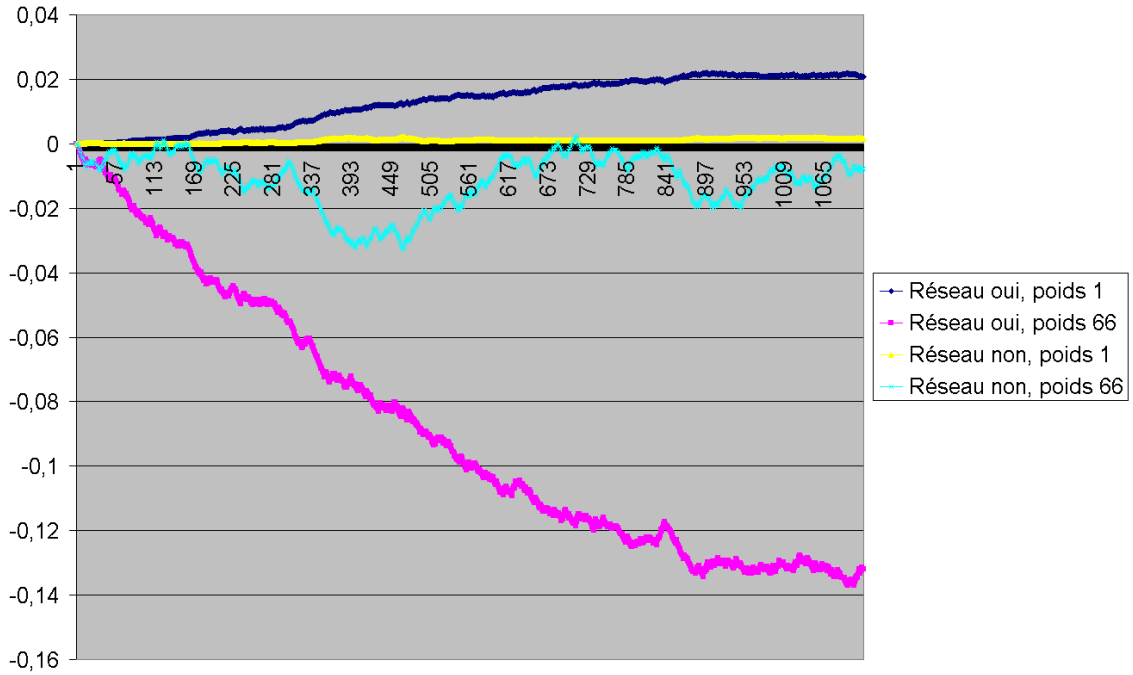

FIG. 17 – Suivi de quelques poids des réseaux à couche cachée pour $\epsilon = 0.01$ pendant l'apprentissage manuel

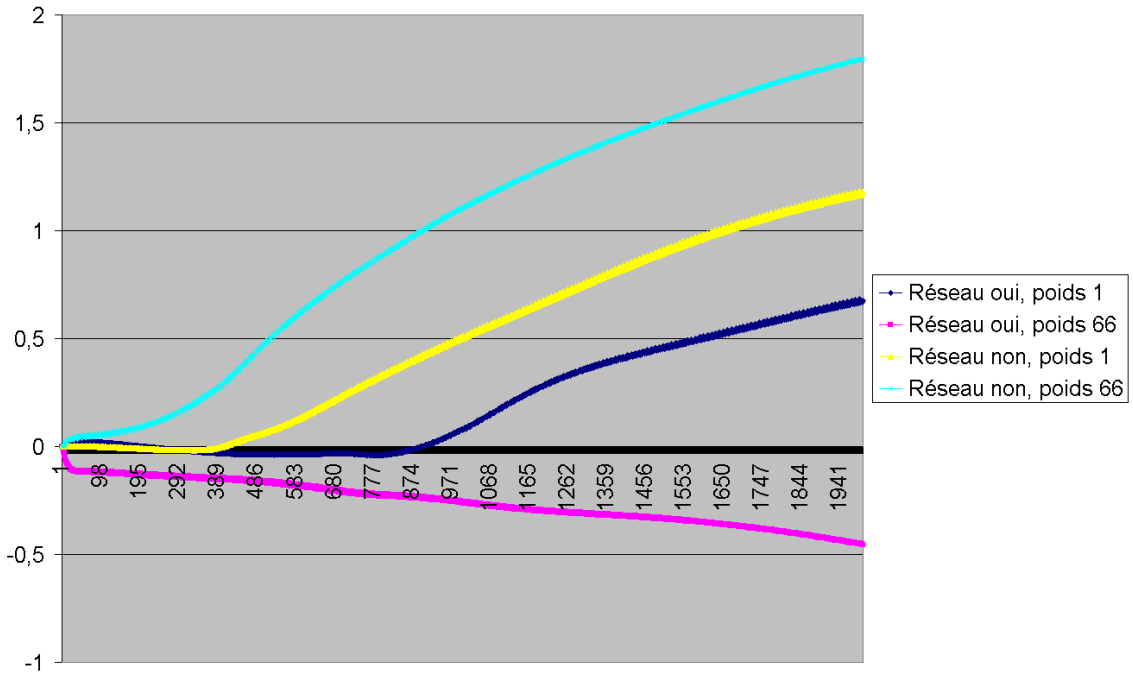

FIG. 18 – Suivi de quelques poids des réseaux à couche cachée pour $\epsilon = 0.01$ pendant l'apprentissage automatique