SAE IA

Voici le lien github:
https://github.com/MathsYeux/SAE_IA

Le projet contient un package pour chaque Exercice (gps, mlp etc.)

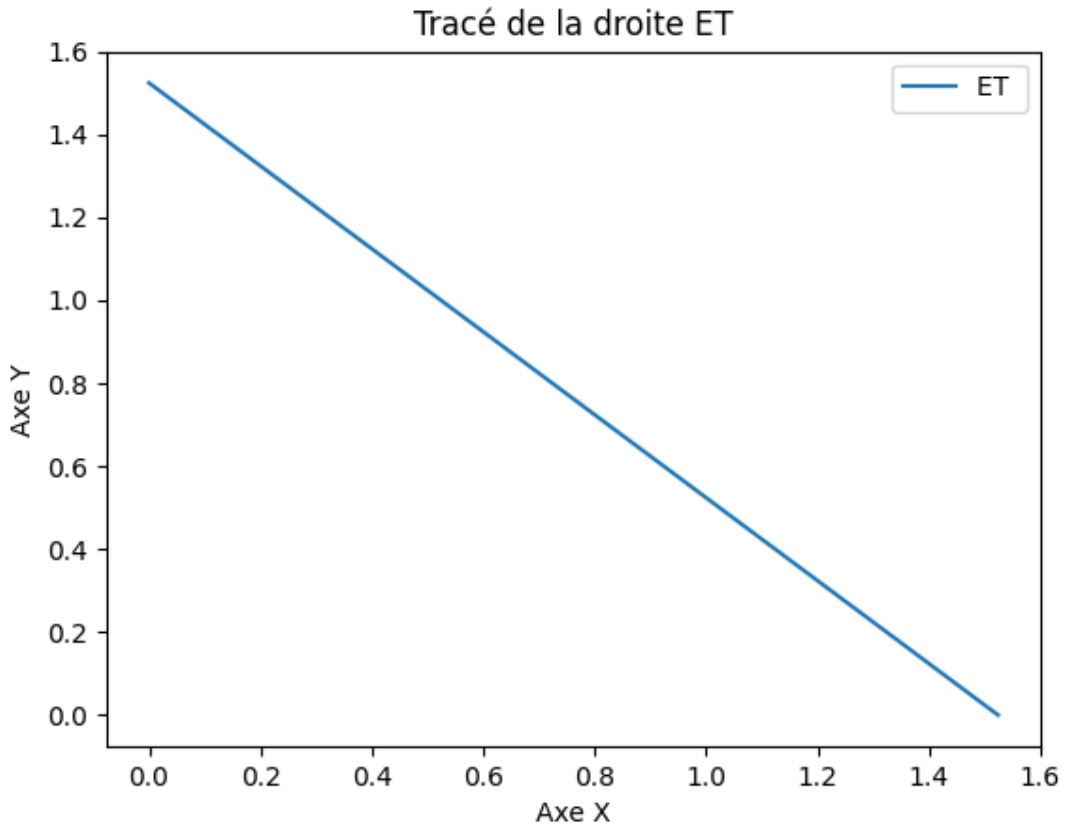
Tracer Table Et et Ou

Fonction de transfert : Sigmoïde

Nombre itération : 1 000 000

ET :

W1 : 7.20992890317847
W2 : 7.209929427242944
Biais : 11.103034954732578

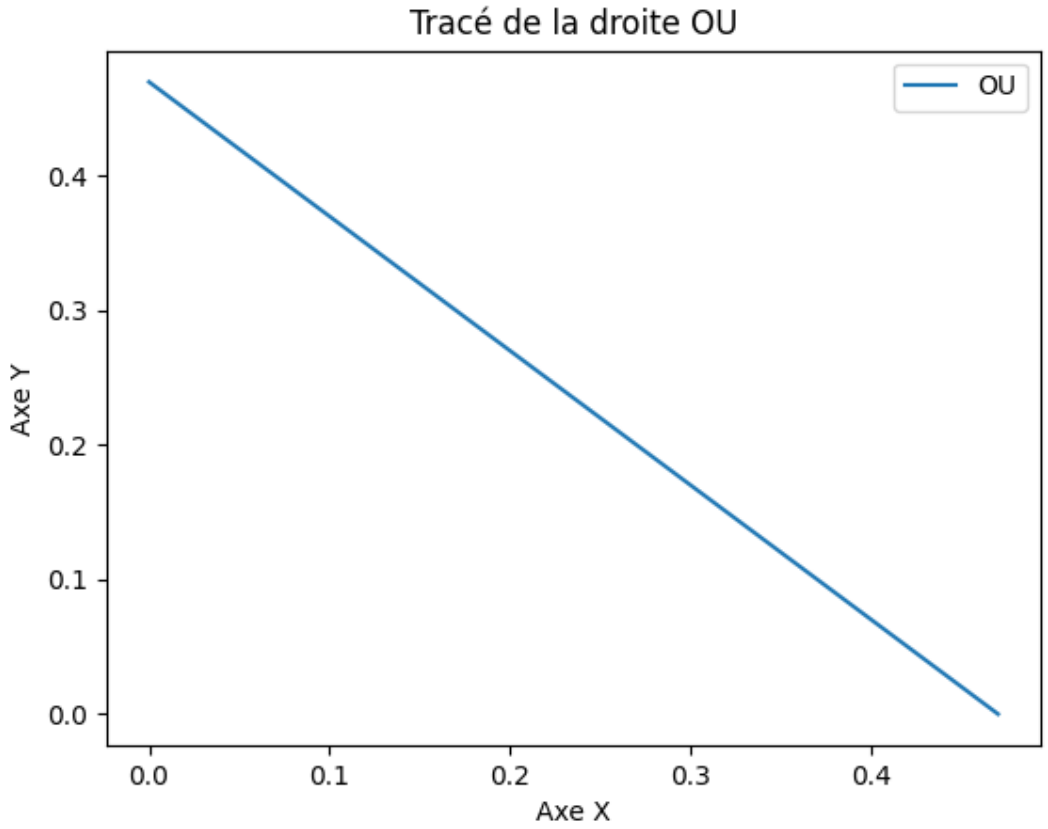


OU:

W1 : 13.09281121653773
W2 : 13.092809237593643

Biais : 6.20756101318874

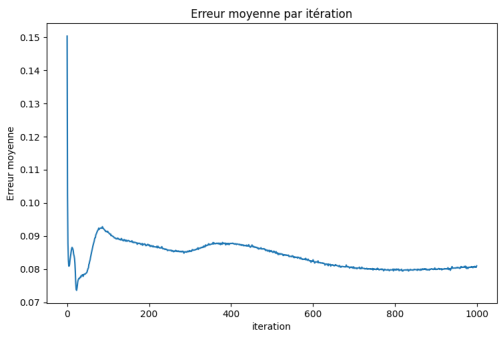
Fonction de transfert: Tanh



Test avec des nombres de neurones et de couches différentes :
On a utilisé Sigmoid avec un learning rate de 0.01
A noter que dans notre cas 1 itération, c'est lorsqu'on passe une fois la base de données de 60 000 images

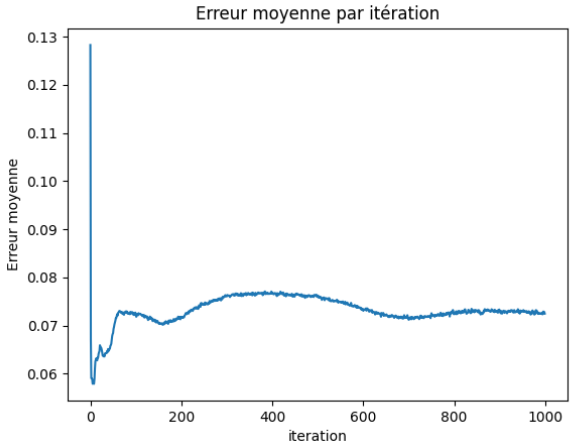
Avec 1 couches cachée:

nb = 10



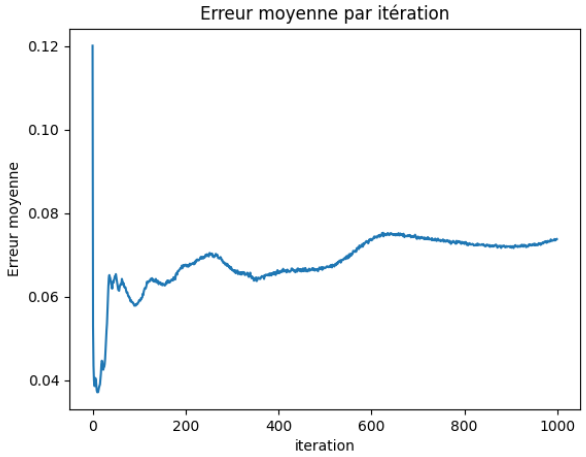
On tend vers les 0.08 d'erreur moyenne, et ce à partir de 300 itérations.

nb = 20



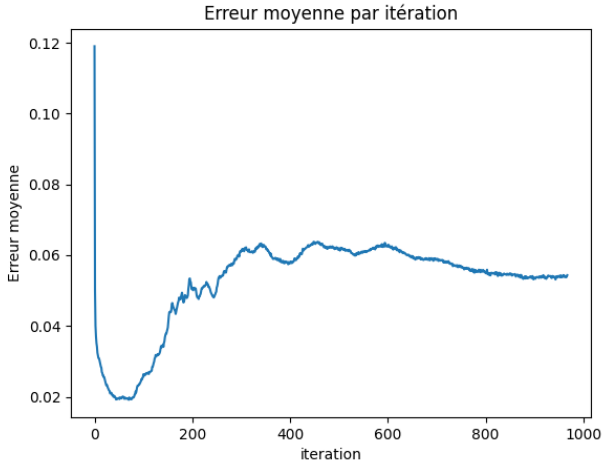
On tend vers les 0.07 d'erreur moyenne, et ce à partir de 200 itérations.

nb = 50



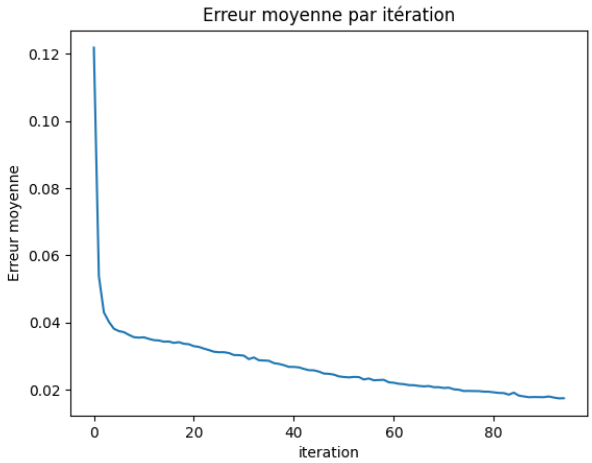
On tend vers les 0.08 d'erreur moyenne, et ce à partir de 200 itérations.

nb = 100



On tend vers les 0.06 d'erreur moyenne, et ce à partir de 200 itérations.

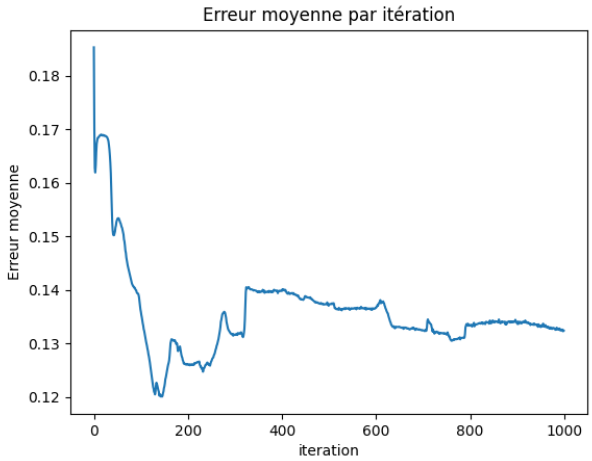
nb = 1000



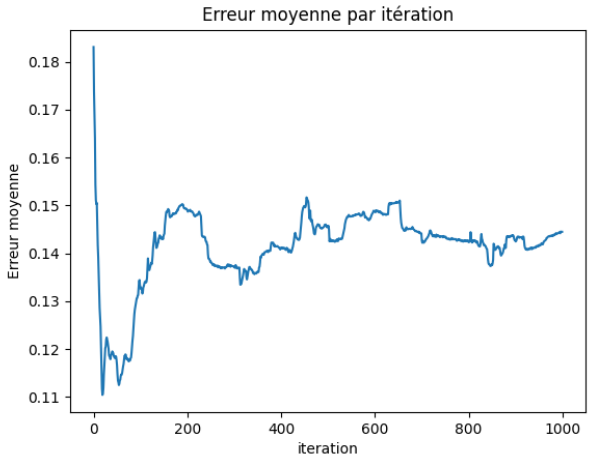
On tend vers les 0.02 d'erreur moyenne, et ce à partir de 80 itérations.

On observe donc que l'erreur moyenne ne descend pas beaucoup en fonction du nombre de neurones dans la couche cachée, il faut changer d'ordre de grandeur pour avoir des résultats significatifs

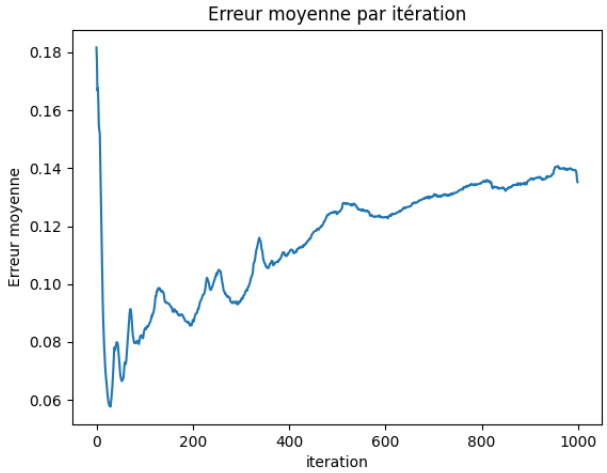
Avec 2 couches cachées:
nb = {5,5}



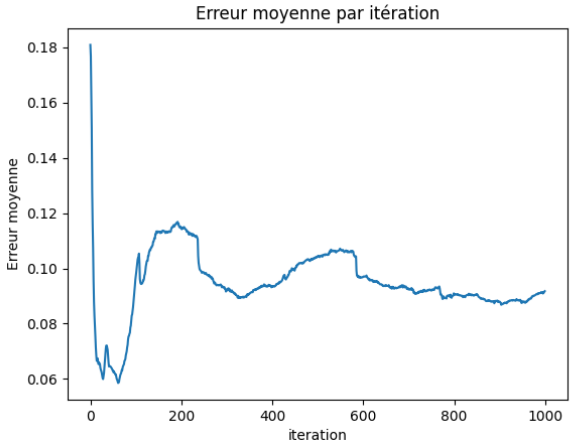
nb = {10,10}



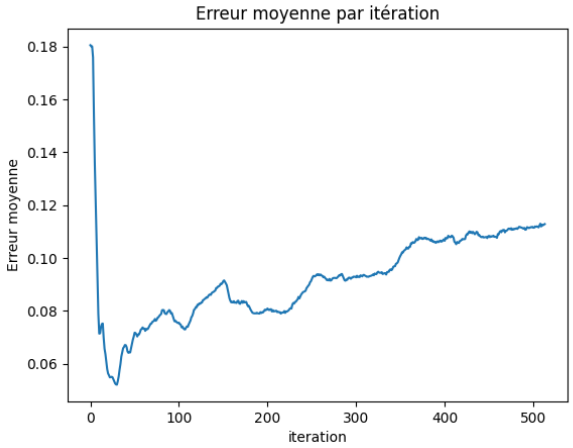
nb = {25,25}



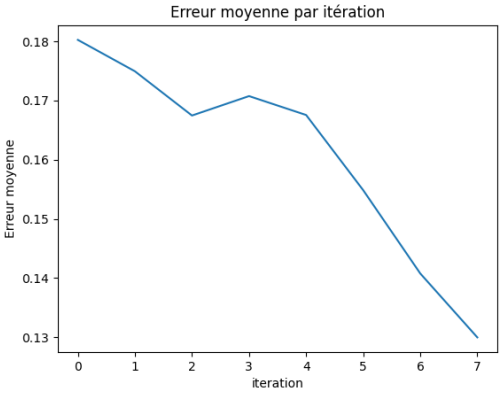
nb = {50,50}



nb = {100,100}



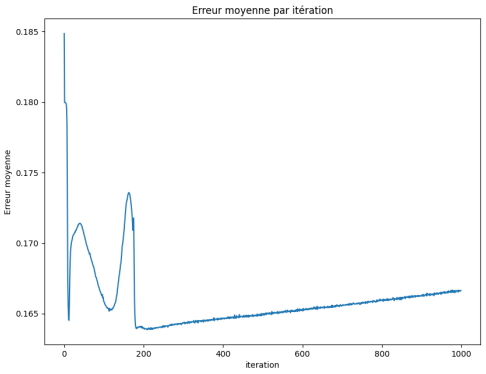
nb = {1000,1000}



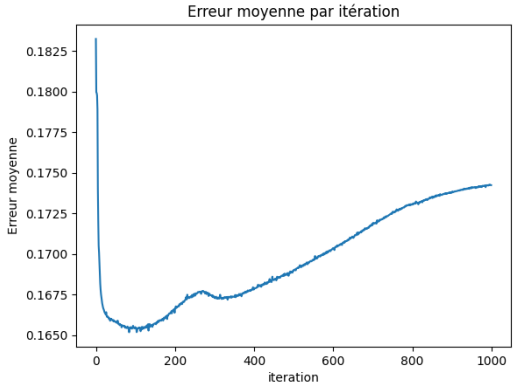
On constate qu'avec 2 couches cachées l'erreur a tendance à être plus élevé qu'avec une seul, comme c'est plus coûteux on va donc éviter cela.

Avec 3 couches cachées:

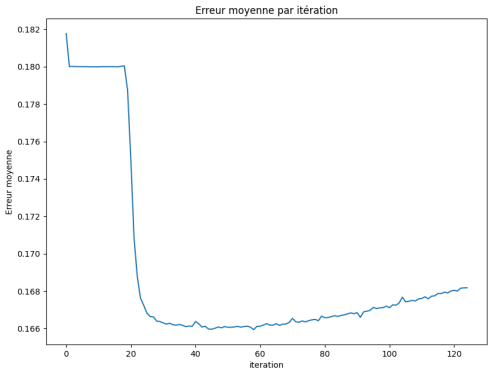
nb = {5,5,5}



nb = {10,10,10}



nb = {25,25,25}



Idem que pour la différence entre 1 et 2 couches cachées, passer à 3 a augmenté à la fois le coût des itérations et l'erreur moyenne.

En conclusion on peut obtenir des résultats plus fiables avec un réseau de neurones (ici avec Sigmoïde) qu'avec un algorithme KNN, en revanche cela nécessite beaucoup d'entraînement, ce qui nécessite du temps et une base de connaissances conséquente.

Défi 1 :

(Voir le projet joint)

Le projet a pour but de faire des requêtes API, une première pour récupérer la liste des communes de France, il va ensuite garder uniquement les 100 les plus peuplées, les transformer en objet Ville puis construire un graphe de villes avec comme poids soit les distances, soit le temps de trajet. Il parcourt ensuite le graphe avec l'algorithme A* et est censé renvoyer le trajet le moins coûteux, malgré beaucoup de temps passé à essayer de le faire fonctionner correctement, le programme renvoie toujours le trajet direct entre le départ et l'arrivée, malgré le fait qu'il parcourt bien le graphe.

Défi 3 - Amélioration de Minimax :

- Estimer le nombre d'états du puissance-4 et le facteur de branchement:

On peut avoir 3 états par case possible {'X','O','Empty'}. Il y a donc 3^42 états. Toutefois, il y a moins de 3^42 états, parce qu'il y a des états qui ne sont pas possible dû aux règles du jeu.

Si facteur de branchement désigne le nombre de possibilités, alors on en a au maximum 7 car on a 7 colonnes.

- Le tester sur le puissance-4 …

Le programme est très long pour effectuer un coup.

- Identifier pour le puissance-4 quelles sont les structures qui influent sur les chances de gagner.

Sur un état, on regarde pour le joueur chaque jeton lui appartenant, et on regarde le nombre d'alignement de 4 jetons. On aura un score arbitraire qui sera le nombre d'alignement de 4 jetons possibles.

- Modifier Alpha-Beta pour fixer une profondeur maximum. Une fois la profondeur atteinte la valeur retournée sera celle de la fonction d'évaluation

Voir la classe AlphaBetaMaxDepth.java. Dans la méthode evaluateGameState, on implémente l'heuristique précédemment proposée.