Projet C/C++

Guillaume Gadeyne
Paul Lebaigue

22 janvier 2019

Une AI pour jouer au snake

Introduction

Le jeu "snake" est l'un des plus vieux jeux video, il semblerait qu'il soit apparu pour la première fois en 1976 dans le jeu d'arcade Blockade. Même si ce jeu semble très rudimentaire aujourd'hui, nous nous sommes interessé à sa conception dans un but pédagogique d'apprentissage du langage C++. L'objectif pour nous était tout d'abord de crée un jeu fonctionnel, puis, dans un deuxième temps, de créer une pseudo intelligence artificielle pour y jouer.

Dans cet optique, nous avons choisi de construire un réseau de neurones classique entièrement connecté. Comme contrainte, nous devions pouvoir récupérer des paramètres du jeux utiles afin d'en faire des paramètres d'entrée du réseau. Nous avons ensuite opté pour une solution moins classique pour l'entrainement de ce réseau, puisque nous avons préféré une optimisation par algorithme génétique.

1 Le jeu du Snake

Le jeu classique se joue dans une matrice rectangulaire, de taille 20x20 dans notre cas. Un serpent évolue dans cette matrice, avançant case par case. Une cible apparait aléatoirement sur la carte, lorsque le serpent passe sur la case de cette cible elle disparait pour augmenter la taille du serpent. Le serpent ne peut se déplacer que d'une case en une case, et lorsqu'il heurte autre chose que la cible (son propre corp ou un mur), la partie s'arrête.

L'objectif du jeu est donc d'obtenir un serpent le plus long possible, le score de la partie étant la longueur du serpent -1 (on commence la partie avec uniquement sa tête), ce qui représente le nombre de cibles récupérées. Dans le jeu original, le serpent se déplace naturellement vers l'avant et l'utilisateur peut choisir de le faire tourner à droite ou à gauche. Pour des raisons pratiques pour l'implémentation de l'IA, notre serpent n'avance pas de lui même et le joueur à donc trois choix : droite, gauche, et avant. Cela permet notamment de s'abstraire du temps et de séquencer le jeu par étapes, chaque mouvement constituant une étape.

2 Le réseau de neurones

2.1 Principe général

Un réseau de neurone peut être vu comme une fonction de transfert. Dans notre cas, cette fonction de transfert va prendre en entrée des caractéristiques du jeu et va renvoyer un choix de mouvement au serpent. Cette fonction de transfert est composée de neurones qui comportent chacun un certain nombre de paramètres. C'est l'optimisation de ces paramètres qui déterminera l'efficacité de cette fonction de transfert.

Pour ce réseau de neurones, nous avons choisi de prendre huit paramètres d'entrées. Deux d'entre eux sont associé à la position de la tête du serpent (X_{tete}, Y_{tete}), deux autres à la position de la cible (X_{cible}, Y_{cible}) et les quatre derniers sont associés aux distances des obstacles dans les quatre directions par rapport à la tête $(X^+_{obstacle}, X^-_{obstacle}, Y^+_{obstacle}, Y^-_{obstacle})$. On se ramène donc à une problème de classification, les directions que prend le serpents faisant office de classes.

2.2 Fonctionnement d'un neurone

Un neurone est lui aussi une fonction de transfert, prenant un certain nombre d'entrée et renvoyant une seule sortie. Cette sortie est la somme pondérée des entrée par les poids du neurone. Il y a donc autant de poids que d'entrées, et ce sont ces poids que nous allons faire varier pour que l'ensemble du réseau renvoi la bonne le bon choix de direction pour le serpent.

La sortie est donc une combinaison linéaire des entrées. Lorsque on considère un réseau de neurone à plusieurs couches, les neurones prennent en entrée les sorties des couches précédentes. Ainsi, sans modification supplémentaires, mettre plusieurs couche n'apporte rien de plus qu'une seule couche puisque cela reste des combinaisons linéaires des entrées du réseau. Afin de modéliser des comportements plus génériques, on applique une transformation non linéaire à la sortie d'un neurone. Cette transformation est appelée fonction de transfert et casse donc la linéarité pour faire un modèle plus généralisable.

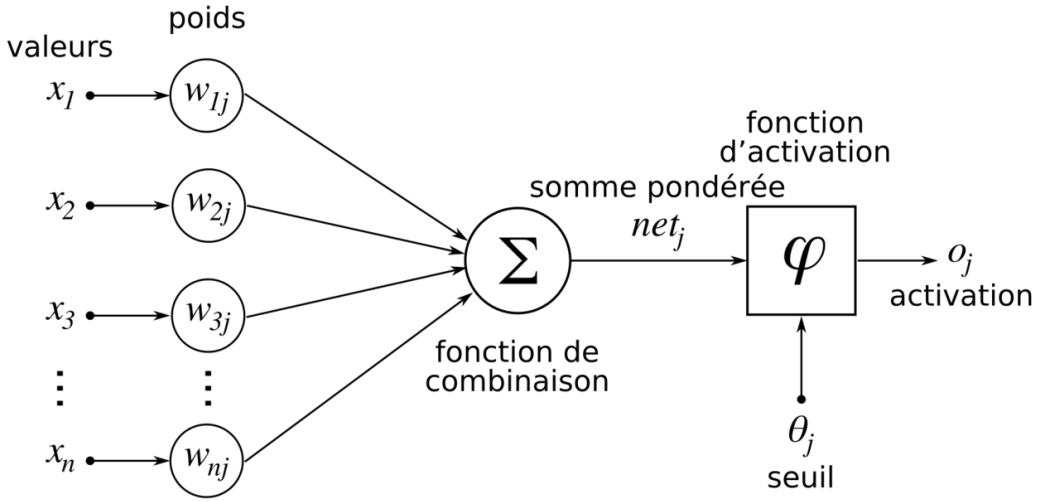

Figure 1 – Illustration du fonctionnement d'un neurone

3 L'optimisation des paramètres

3.1 Principe de l'algorithme

Afin de trouver des paramètres permettant au serpent de prendre de bonnes décisions, nous avons opté pour un algorithme d'optimisation génétique. On prend une population de taille N définie par avance, chaque individu de cette population représentant un jeu de paramètre d'un réseau. Les poids des paramètres de la première population sont initialisés au hasard.

Pour l'optimisation, on séléctionne 25% de cette population, ce seront les parents. Ces 25% sont choisit en bonne partie comme étant les meilleurs scores de la génération précédente, et en plus faible proportion aléatoirement (afin de garder de la richesse génétique). On croise ensuite les parents afin de créer des "enfants", la somme des parents séléctionnés et des enfants générés forme la nouvelle génération. Nous avons ici choisi de garder la taille de la population constante, ainsi on génère les 75% manquant à partir des parents, et donc un parent a en moyenne 6 enfants.

L'idée est que l'évolution de la population converge vers une famille de solution qui arrivera à guider le serpent en fonction des entrés, à partir du principe de séléction naturelle. Ces performances dépendent bien entendu du nombre de générations utilisées dans l'optimisation et de la taille choisie pour la population.

3.2 Le croisement entre deux parents

Afin de générer des enfants, on effectue le croisement entre deux parents. Pour chaque paramètre du réseau enfant on va choisir aléatoirement le paramètre de l'un des parents. On génère de cette façon un nouveau réseau qui dispose en espérance d'autant de paramètre de chacun des parents. C'est une technique appelé croisement uniforme, par opposition au croisement par point ou l'on découpe les gènes en morceaux et ou l'on séléctionne les morceaux sur l'un ou l'autre des parents. Une autre technique envisageable est de selectionner directement les neurones des parents pour former un enfant.

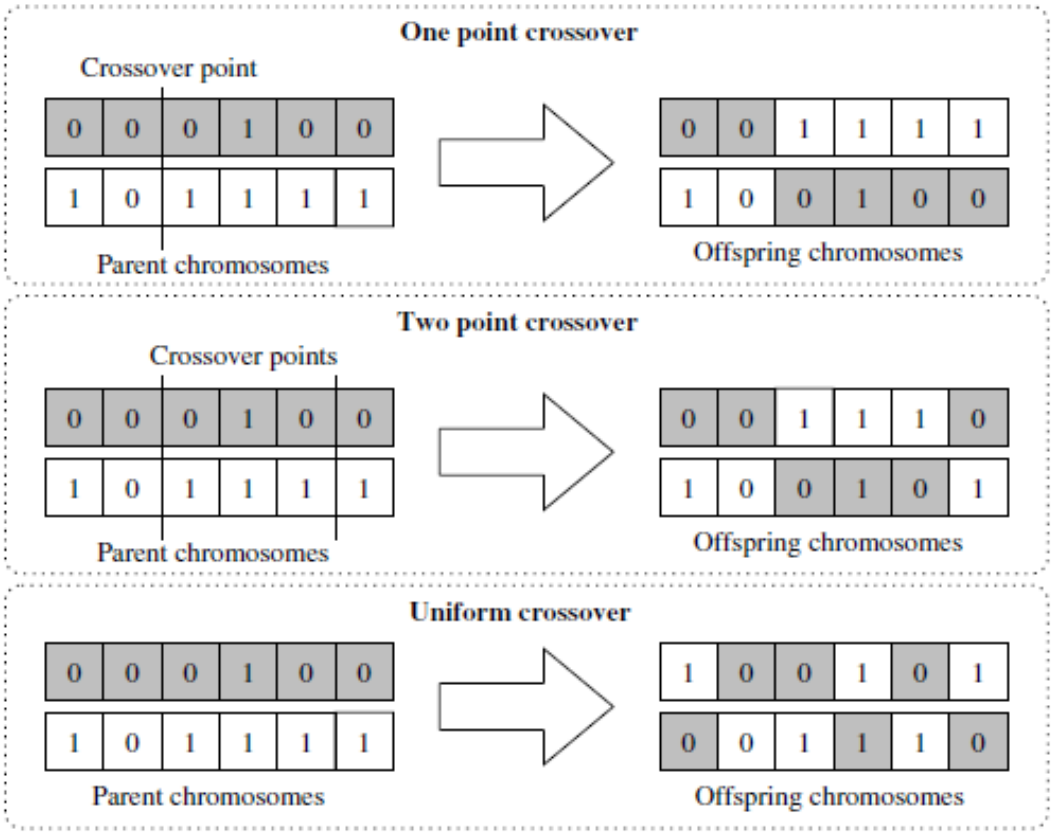

Figure 2 – Illustration des principes de croisement

4 Architecture du code

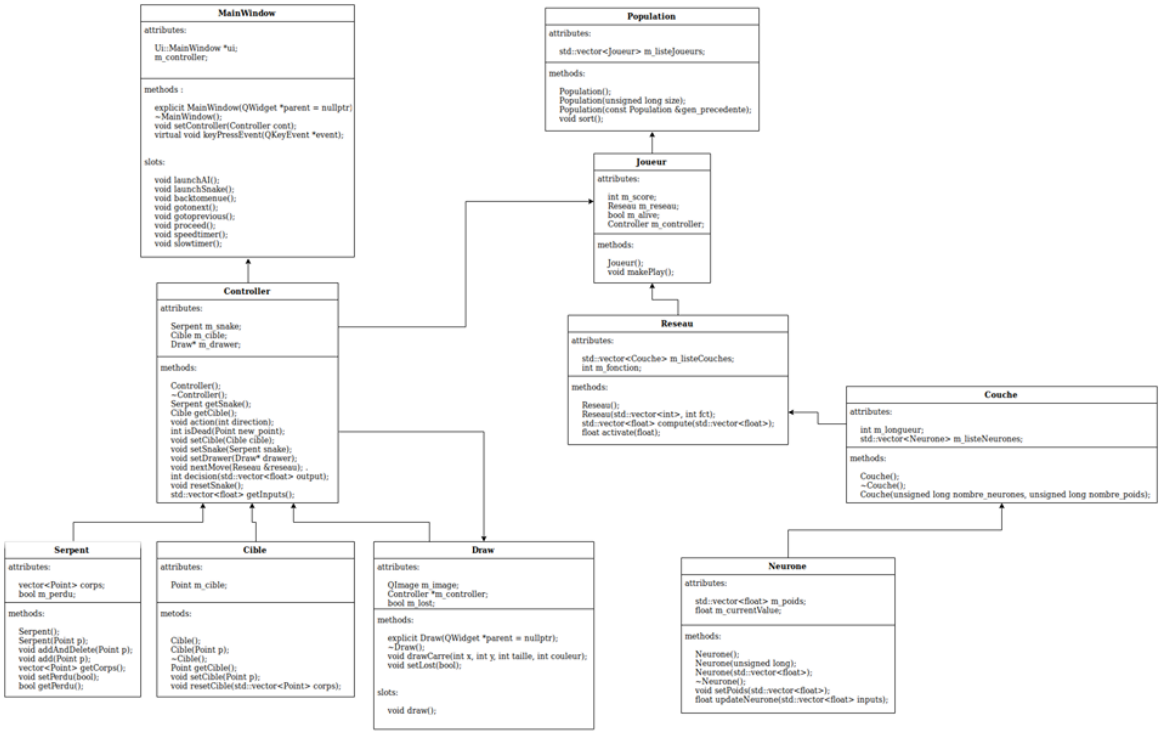

Figure 3 – Relations entre les différentes entités

A défaut d'avoir trouvé un éditeur d'UML correct, dans ce schéma $A \rightarrow B$ veut dire : A est un attribut de B.

5 Exécution du programme

Après avoir ouvert le projet sous Qt-Creator, il faut le compiler puis l'éxécuter pour arriver sur le menu suivant :

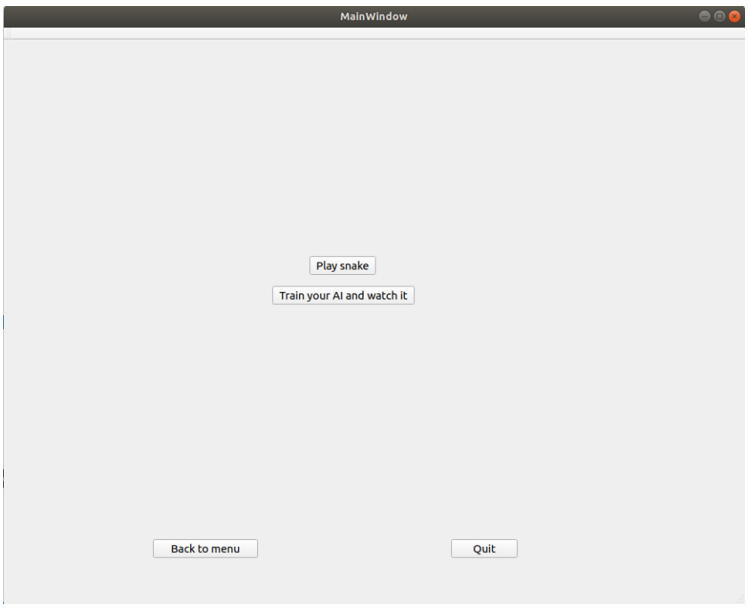

Figure 4 – Menu d'entrée du jeu

On peut alors choisir de jouer au snake ou de regarder l'AI s'améliorer de génération en générations. Il est notamment possible de faire varier la vitesse de jeu ou de changer de générations.

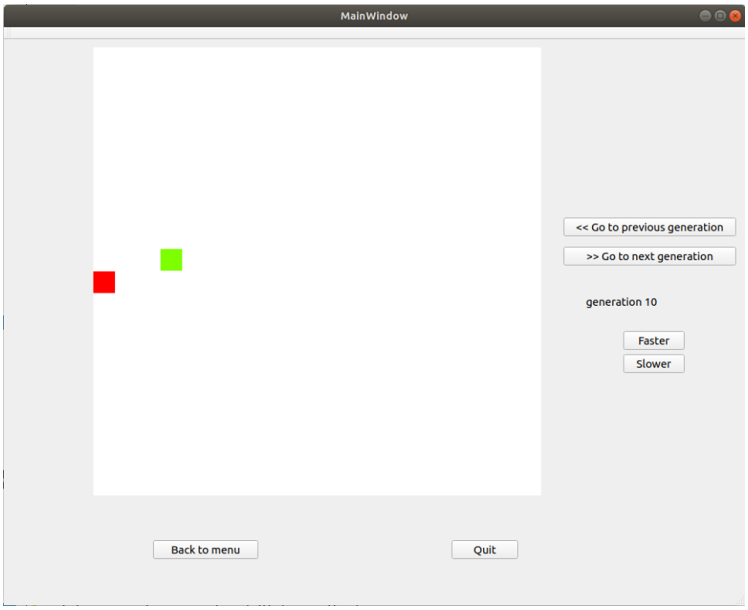

Figure 5 – Exemple d'une partie jouée par l'AI

6 Résultats

Après quelques générations la pseudo-AI touche la cible 1 ou 2 fois par partie. Cette valeur n'augmente pas avec le nombre de générations, ainsi tout laisse à penser que notre algorithme n'apprend pas réellement à jouer.

Nous avons testé l'algorithme génétiques pour plusieurs valeurs de populations, avec différents croisements, et avec différentes topologies de réseaux, malheureusement nous ne sommes pas parvenus à optimiser le réseau. Il est probable que cette optimisation soit faisable et que nous nous y soyons mal pris, mais il est également envisageable que nous ayons mal chosi les données que prend en entrée le réseau, et que cela ne marche pas correctement pour cette raison.

Nous avons gardé la possibilité de modifier la taille de la population, le nombre de génération dans MainWindow : :launchAI(), la fonction d'activation dans Joueur : :Joueur() ainsi que la topologie utilisée dans Joueur : :Joueur(). On peut espérer qu'il soit possible de jouer un peu plus longtemps sur ces paramètres pour obtenir un résultat plus convaincant. L'autre solution plus classique de ce genre de problème serait d'entrainer le réseau de neurone par descente de gradient grâce notamment à l'astuce de la rétropropagation, à laquelle nous ne nous sommes pas frottés ici.