Rapport projet NEAT

Introduction

Le but du projet est d'implémenter l'algorithme NEAT, et l'utiliser pour résoudre le problème du XOR.

L'algorithme NEAT offre une approche qui permet l'évolution de la topologie des réseaux neuronaux en permettant l'ajout de neurones, ou de connexions.

Ce rapport détaille la conception et la mise en œuvre du projet, en explicitant mes choix de conceptions, les paramètres utilisés, et en analysant les résultats obtenus tout au long des générations. Je vais expliquer étape par étape comment j'ai développé ce projet.

Architecture du projet

Le projet suit une architecture basique. La boucle d'évolution se retrouve dans la classe Main du programme, tandis que tous les algorithmes et processus spécifiques à NEAT se retrouvent dans différentes classes et helpers.

Node

La classe Node représente un nœud dans le réseau neuronal utilisé dans l'algorithme NEAT. Chaque nœud possède plusieurs attributs comme son identifiant, son type, sa couche, ainsi que les sommes d'entrées et de sorties associées.

Brain

Cette classe représente un réseau neuronal avec des méthodes pour l'initialiser, le faire fonctionner, le muter, et calculer ses performances. Elle stocke les informations comme le fitness, son ID, et l'ID de l'espèce à laquelle il appartient.

Connexion

Cette classe représente une connexion entre deux nœuds dans le réseau neuronal utilisé dans l'algorithme NEAT. Chaque connexion est caractérisée par plusieurs attributs comme son identifiant d'innovation, les identifiants des nœuds d'entrée et de sortie, le poids associé à la connexion, et un booléen indiquant si la connexion est active ou pas.

Le poids est une variable importante car c'est elle qui détermine la justesse de la connexion.

Specie

Représente une espèce avec ses membres spécifiques, un ID, et des informations sur son évolution au fil des générations.

Helpers

Différents Helpers sont utilisés afin de simplifier la lecture. Il existe un helper pour toutes les méthodes propres à l'utilisation des Brains, un pour les Species, un pour les Connections, un pour les paramètres tels que la fitness ou les offsprings, et enfin un TournamentManager, permettant d'utiliser la technique de sélection par tournoi. Tous ces termes seront expliqués dans le rapport.

La génération 0

La première étape de réalisation de ce projet a été d'implémenter une première génération, sa logique de création, ainsi que de créer les différentes classes nécessaires.

Ca n'est pas l'étape qui est techniquement la plus compliquée, mais elle permet de se familiariser avec quelques concepts importants.

Par exemple, la fonction runNetwork() de la classe Brain implémente la façon dont la valeur de sortie de chaque nœud va être calculée.

```
for (int layerCount = 2; layerCount <= outputLayer; layerCount++) {
    //hidden nodes
    if (layerCount != outputLayer) {
        for (Node hiddenNode : neatParameters.hiddenNodes) {
            if (hiddenNode.nodeLayer == layerCount) {
                hiddenNode.sumInput = 0;
                for (Connection connection : neatParameters.connections.stream().filter(co -> co.isEnabled).toList()) {
                    if (hiddenNode.nodeID == connection.outNodeID) {
                        Node connectionInputNode = findNodeById(connection.inNodeID);
                        assert connectionInputNode != null;
                        hiddenNode.sumInput += (connectionInputNode.sumOutput * connection.weight);
                    }
                }
                hiddenNode.sumOutput = 1 / (1 + exp(-4.9 * hiddenNode.sumInput));
            }
        }
    }
```

Couche par couche, pour chaque noeud :

- On calcule sa valeur d'entrée : c'est la somme des valeurs de sorties des nœuds pour lesquels il existe une connexion multipliée par le poids de cette connexion.
- On calcule sa valeur de sortie :

  hiddenNode.sumOutput = 1 / (1 + exp(-4.9 * hiddenNode.sumInput));

Au bout de la première génération, un certain nombre de réseaux de neurones (RN), nombre défini dans la variable populationSize, auront tenté de résoudre le XOR pour les 4 valeurs possibles de l'entrée. Ils y seront arrivés avec plus ou moins de succès.

Pour rappel, les valeurs d'entrée possibles et la sortie correspondante attendue sont :

1. {0, 0} qui a pour sortie 0
2. {0, 1} qui a pour sortie 1
3. {1, 0} qui a pour sortie 1
4. {1, 1} qui a pour sortie 0

Pour chaque sortie, on retient l'erreur que le RN a fait. Elle correspond à la valeur absolue de la différence entre la sortie attendue et la sortie obtenue.

Une fois que les sorties ont été calculées pour les 4 entrées, la **fitness** du RN va prendre la valeur :

Fitness = 4 – (somme des erreurs)

Cette fitness sera le score du RN. Le but est de se rapprocher au maximum de 4. Lorsque toutes les fitness ont été calculées, la première génération a fini son travail.

Spéciation

Principe

Après qu'une génération ait fini de tester tous ses RN, il est temps de passer à la spéciation. La spéciation consiste à diviser la population en différentes espèces. Chaque espèce aura une fitness moyenne, et surtout une fitness ajustée moyenne, critère qui permettra de déterminer combien de RN de la même espèce on veut au maximum dans la génération suivante.

La répartition en espèces se fait de cette manière :

Pour la première génération :

- Un RN sans espèce est sélectionné au hasard comme étant le chef d'une espèce
- Chaque autre membre sans espèce de la population est comparé à ce chef grâce à une valeur appelée la différence de compatibilité (DC)
- Si la DC est inférieure à un seuil, le membre est associé à l'espèce
- Après avoir parcouru tous les membres, s'il en reste qui n'ont pas encore d'espèce, on recommence le processus

A la fin, chaque membre sera associé à une espèce. Parfois, une espèce ne contiendra qu'un membre.

Différence de compatibilité

Le critère de différence de compatibilité est calculé en fonction des différences de topologie et des différences de poids de connexions. Il se fait en 4 parties :

La première consiste à vérifier les connexions supplémentaires qu'un RN a par rapport à un autre. Une connexion supplémentaire correspond à une connexion dont l'innovationID est supérieur au plus haut de la liste de l'autre RN.

```java
public static double getExcessConnections(Brain brainLeader, Brain brainCompared) {
    double result = 0;
    int highestInnovationIDLeader = getHighestInnovationID(brainLeader.neatParameters.connections);
    int highestInnovationIDCompared = getHighestInnovationID(brainCompared.neatParameters.connections);
    // If leader has higher innovationIDs than compared brain, count how many.
    if (highestInnovationIDLeader > highestInnovationIDCompared) {
        for (Connection leaderConnection : brainLeader.neatParameters.connections.stream().filter(co -> co.isEnabled).toList()) {
            if (leaderConnection.innovationID > highestInnovationIDCompared) {
                ++result;
            }
        }
    }
    // If compared brain has higher innovationIDs than leader brain, count how many.
    else if (highestInnovationIDCompared > highestInnovationIDLeader) {
        for (Connection comparedConnection : brainCompared.neatParameters.connections.stream().filter(co -> co.isEnabled).toList()) {
            if (comparedConnection.innovationID > highestInnovationIDLeader) {
                ++result;
            }
        }
    }
    // Note : if they have the same highest innovation ID, result will be 0 which is correct
    return result;
}
```

La seconde étape consiste à obtenir le total de connexions disjointes, c'est-à-dire le nombre de connexions pour lesquelles l'innovationID est plus petit que l'innovationID maximal de la plus petite liste de connexions et qui ne se retrouve pas dans l'autre liste.

```java
public static double getDisjointConnections(Brain brainLeader, Brain brainCompared) {
    double result = 0;
    List<Connection> leaderConnections = brainLeader.neatParameters.connections.stream().filter(co -> co.isEnabled).toList();
    List<Connection> comparedConnections = brainCompared.neatParameters.connections.stream().filter(co -> co.isEnabled).toList();
    // get the highest innovation ID in the list that has the smallest highest one.
    int highestInnovationIDInSmallestList = Math.min(getHighestInnovationID(leaderConnections), getHighestInnovationID(comparedCo

    // Now, scan through both lists to get the amount of connections with Innovation IDs that are in only one of the two lists
    // First we check how many connections are in the leader list and not in the compared one.
    // get a list of all compared innovation IDs
    Set<Integer> innovationIDsSet = ConnectionsHelper.getInnovationIDSet(comparedConnections);
    for (Connection leaderConnection : leaderConnections.stream().filter(co -> co.isEnabled).toList()) {
        if (leaderConnection.innovationID < highestInnovationIDInSmallestList) {
            if (!innovationIDsSet.contains(leaderConnection.innovationID)) {
                ++result;
            }
        }
    }
    // Then we do the opposite
    innovationIDsSet = ConnectionsHelper.getInnovationIDSet(leaderConnections);
    for (Connection comparedConnection : comparedConnections.stream().filter(co -> co.isEnabled).toList()) {
        if (comparedConnection.innovationID < highestInnovationIDInSmallestList) {
            if (!innovationIDsSet.contains(comparedConnection.innovationID)) {
                ++result;
            }
        }
    }
    // After that we can return the result
    return result;
}
```

La troisième étape consiste à calculer, pour chaque connexion commune aux 2 listes, la valeur absolue de la différence moyenne de poids.

Une fois qu'on connait ces 3 facteurs et qu'on connait également la taille de la plus grosse liste de connexions, on peut appliquer cette formuler :

```java
return (c1 * (excessConnections / highestConnectionsAmount))
        + (c2 * (disjointConnections / highestConnectionsAmount))
        + (c3 * (weightDifference / highestConnectionsAmount));
```

En sachant que c1, c2 et c3 sont des poids définis arbitrairement visant à minimiser ou maximiser l'impact de l'une des 3 parties de la formule.

Calcul des offsprings

Jusqu'à présent, la génération s'est essayée au XOR, puis a été répartie en différentes espèces. Maintenant, on veut savoir combien de membres de cette espèce on peut accueillir au maximum dans la génération suivante.

Ce choix se fait en fonction d'un critère important : la fitness ajustée.

Pour chaque RN, la fitness ajustée se calcule en divisant sa fitness par le nombre d'individus de la même espèce que lui. On peut donc calculer la fitness ajustée moyenne d'une génération.

Afin de calculer le nombre d'offsprings d'une espèce, qui correspond au nombre d'enfant maximal accueillables dans la génération suivante, on divise sa fitness ajustée moyenne par la fitness ajustée moyenne globale de la génération, et on multiplie cela par le nombre de membres actuel de l'espèce.

Crossover

Quand les membres ont été divisés en espèce, et que l'offspring de chaque espèce a été calculé, on peut considérer l'étape de spéciation comme terminée.

La suivante est l'étape de crossover, qui consiste à créer les membres de la génération suivante en se basant sur les membres de la génération actuelle. Chaque « enfant » aura des caractéristiques de ses 2 « parents ».

Pour chaque espèce, on va d'abord sélectionner des parents. Cette sélection peut se faire de différentes manières, comme avec une sélection par roulette par exemple, qui laisse une chance équilibrée à chaque membre d'être choisi. J'ai personnellement décidé de fonctionner avec une sélection par tournoi, pour avoir plus de chances de choisir des parents qui ont bien performé vis-à-vis de leur génération.

La sélection par tournoi va d'abord choisir aléatoirement un nombre prédéfini de participants, et ensuite les faire concourir entre eux pour ne garder que celui dont la fitness ajustée est la plus haute. Ca sera le premier parent. On réitère l'opération pour le deuxième.

Une fois cette sélection terminée, chaque espèce va créer un nombre d'enfants correspondant à son offspring, calculé plus tôt. Parmi les 2 parents, on clone celui qui a la meilleure fitness ajustée. Ensuite, pour chacune des connexions du clone, son poids va être modifié pour prendre la valeur du poids de la connexion correspondante d'un des 2 parents au hasard. Après cela, le clone est ajouté à la génération suivante comme enfant des 2 parents.

Mutation

Nous avons donc désormais une base pour la première génération : une série d'enfants qui ont subi le phénomène de crossover. La dernière étape de l'évolution est la mutation.

Chaque enfant va avoir une chance de muter :

Changement de poids

La première mutation qui peut arriver est le changement de poids. Si un RN est sélectionné pour muter de cette manière, pour chaque connexion, elle aura 90% de chances de voir son poids augmenter ou diminuer de 20%. Sinon, son poids sera juste réinitialisé de manière aléatoire.

Cette mutation a 80% de chances d'arriver dans mon implémentation.

Ajout d'une connexion

Si un RN est sélectionné pour ce type de mutation, 2 de ses noeuds vont être choisis au hasard pour être un inNode et un outNode. S'ils respectent une série de critères de validation, une nouvelle connexion va être créée entre eux.

Les critères de validation sont :

- La couche de inNode doit être strictement inférieure à outNode
- Les 2 sélectionnés doivent être différents
- Il ne doit pas y avoir de connexion déjà existante entre ces 2 nœuds.
    - Si à cette étape, on se rend compte qu'il y a déjà une connexion, et que celle-ci est désactivée, elle aura 25% de chances d'être réactivée.

Afin de ne pas perdre trop de temps, les nodes pourront être sélectionnés au hasard au maximum 20 fois, après quoi je considère qu'elle a raté sa chance, et aucune connexion ne sera ajoutée.

Dans mon implémentation, chaque RN a 5% de chances de subir cette mutation.

Ajout d'un nœud

Si un RN est sélectionné pour ce type de mutation, 1 nouveau noeud va apparaitre dans sa topologie. Pour ce faire, différentes étapes :

- Sélectionner une connexion au hasard et la désactiver
- Créer un nouveau nœud
- Créer une première connexion, qui va de l'entrée de la connexion désactivée à la l'entrée du nouveau nœud
- Créer une seconde connexion, qui va de la sortie du nouveau nœud à la sortie de la connexion désactivée
- Puis, il faut recalculer les couches de chaque nœud, car le nouveau nœud dans la topologie a pu modifier les couches et la répartition des nœuds dans celles-ci.

Pour cette dernière étape, j'ai décidé d'implémenter l'algorithme de Depth-First Search. Cet algorithme récursif va me permettre de recalculer avec justesse la couche à laquelle chaque nœud se trouve, en calculant la taille du chemin maximal pour rejoindre un nœud de couche 1. Les nœuds de couche 1 sont les entrées, ceux-ci ne bougent jamais. La taille du chemin correspond donc au nombre de sauts maximum qu'il faut faire pour atteindre la première couche. Il reste à incrémenter ce résultat de 1 pour obtenir la couche où placer le nœud.

Résultats

Une fois la mutation terminée, l'évolution est entièrement complète. On peut donc maintenant faire s'enchainer les générations, et voir comment les résultats évoluent. Avant d'afficher les meilleurs résultats obtenus, je veux juste lister les paramètres modifiables qui impactent le plus l'évolution.

1. La taille de la population
2. Les bornes entre lesquelles le poids d'une connexion peut varier
3. Le seuil permettant de décider si oui ou non un RN appartient à une espèce
4. La valeur du facteur servant dans la sigmoïde pour le calcul de la valeur de sortie d'un node
5. La topologie de départ, avec la présence ou non d'un nœud de biais. Un nœud de biais est un nœud pour lequel la valeur d'entrée vaut toujours 1.

Maintenant place aux résultats. Ils ont extrêmement varié au cours de mes différents tests et modifications de mon programme. Dans ses meilleures versions, la convergence se faisait après une cinquantaine de générations. Mon meilleur résultat a été une convergence en seulement 24 générations, mais je n'ai pas pu la reproduire de manière constante. Pour obtenir de bons résultats, j'utilise une population de 50, des poids de connexions variant entre -1 et 1, un seuil de spéciation à 1.0 et ma sigmoïde possède un facteur de 4.9. Au départ, j'ajoute un nœud de biais à la topologie, qui restera présent dans chaque RN.

```
------------------------------------------------------------
In generation 24, the best brain is brain 3 from specie 27
It has a fitness = 3.9655207256482368, and an adjusted fitness = 3.9655207256482368
```

Le programme dans sa version actuel est toujours absolument fonctionnel, bien que j'ai assez honte de dire que je finis ce rapport juste avant la deadline, et que dans la version actuelle du programme, la convergence ne se fait que très lentement. Je sais qu'en jouant avec les paramètres et en repassant à d'anciens commits, je pourrais retrouver une version qui converge à tous les coups.