Université de Montpellier / Faculté Des Sciences

Neurocontrôleurs

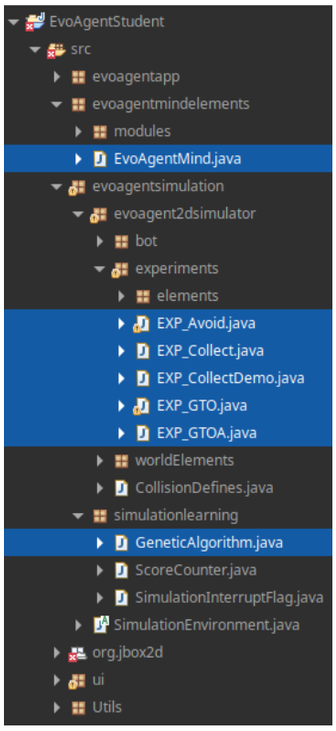
Algorithmes d'exploration — HMIN 233

Suro François

4 janvier 2021

Dans ce TP nous allons créer notre propre structure neuronale (un perceptron multi couches), adapter l'algorithme génétique de la séance précédente pour entraîner notre réseau neuronal, puis l'utiliser comme neurocontrôleurs sur un robot simulé afin d'obtenir des comportements comme ceux-ci : `https://www.lirmm.fr/~suro/videos/ComplexSkills.mp4`.

1 EvoAgentStudent

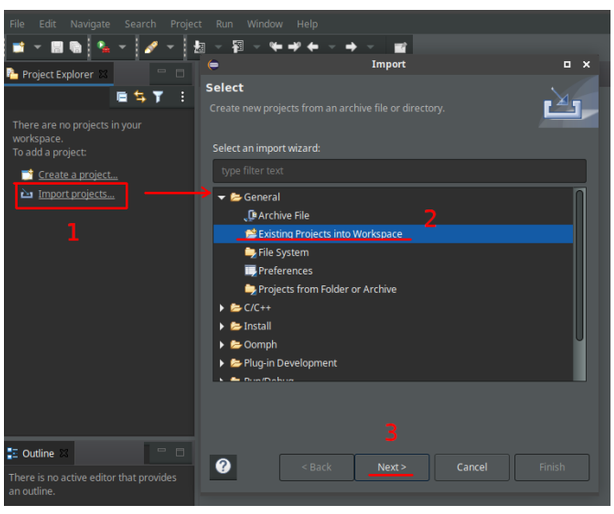


Vous allez utiliser le projet *EvoAgentStudent* qui est un petit outil qui s'occupera de la simulation, l'environnement physique et du calcul distribué de l'algorithme génétique.
À gauche vous pouvez voir l'arborescence du projet dans Eclipse. Les classes surlignées en bleu sont celles dans lesquelles vous interviendrez :

- `EvoAgentMind` : "L'esprit" de l'agent, dans lequel vous écrirez votre structure neuronale.
- `GeneticAlgorithm` : Dans la 2eme partie du TP vous porterez votre algorithme génétique de la séance précédente sur ce problème.
- `EXP_*` : Les classes expériences. Dans la dernière partie du TP vous mettrez en place les environnements et fonctions de récompense afin d'entraîner votre robot à accomplir des tâches. Les packages *experiments.elements* et *worldElements* contiennent des fonctions de récompenses et des éléments "physiques" de l'environnement respectivement, pour ceux qui voudrons créer leurs propres environnements.

En dehors de ces classes vous n'aurez en principe rien d'autre à modifier.

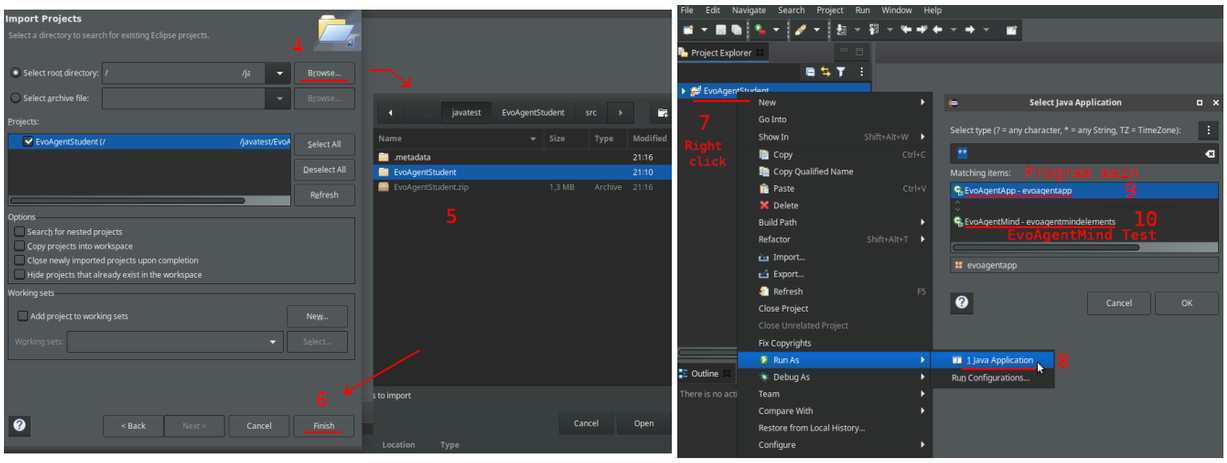
2 Importer le projet eclipse



Il est fortement recommandé d'utiliser Eclipse pour ce projet (`www.eclipse.org/downloads/`).

Lancez Eclipse et créez votre workspace (dossier de travail).

Après avoir décompressé le dossier `EvoAgentStudent.zip` dans votre workspace, suivez les étapes de ce guide pour importer le projet. L'étape 7 permet de lancer le projet : le choix 9 lance le programme principal, le choix 10 permet de tester votre perceptron sans lancer le projet.



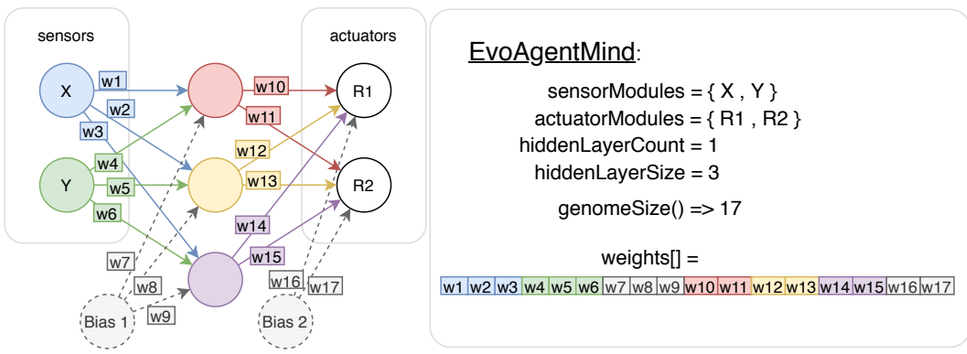
3 Le perceptron multi couches

La structure neuronale sera implémentée dans la classe *EvoAgentMind*, qui est intégré dans le projet.

Cette classe comporte une fonction main pour tester votre code sans lancer tout le projet. Vous pouvez la modifier à loisir.

La figure suivante présente le contenu d'un objet de type EvoAgentMind correspondant au schéma du réseau neuronal donné.

- sensorModules : une liste qui contient des objets de type SensorModule, qui vous fournissent une valeur normalisée des capteurs de l'agent. EX : pour accéder à la valeur du premier capteur de la liste : `sensorModules.get(0).getValue()`.
- actuatorModules : une liste qui contient des objets de type ActuatorModule, qui transmettent les commandes motrices à l'agent. EX : pour écrire la commande du premier actionneur de la liste : `actuatorModules.get(0).setMotorValue(0.5)`. Attention : la valeur fournie doit être normalisée (entre 0 et 1).
- hiddenLayerCount : le nombre de couches cachées (ici une couche).
- hiddenLayerSize : le nombre de neurones dans chaque couche cachée, sans compter le neurone de biais si vous souhaiter l'implémenter (ici 3 neurones).
- : weights[] : un tableau de doubles représentant les poids du réseau. Sa taille sera définie par la fonction genomeSize(), en fonction de tous les paramètres précédents. Ici vous pouvez voir une façon d'organiser les poids dans la liste, peu importe votre choix d'organisation il faut qu'il soit le même tout le temps.



Complétez les fonctions suivantes :

1. `public double applyTransfer(double v)` : Applique une fonction de transfert sur la valeur passée en paramètre. On pourra utiliser la tangente hyperbolique (Math.tanh(x)).
2. `public int genomeSize()` : Fonction qui renvoie la taille du génome en fonction du nombre de neurones dans les différentes couches ainsi que du nombre de couches cachées. Pensez à compter les neurones de biais si vous en utilisez. La taille du génome correspond à la taille du tableau des poids, et donc au nombre de connexions dans le réseau.
3. `public void doStep()` : Calculez la valeur de sortie pour les actionneurs (actuators) en fonction des valeurs d'entrés des capteurs (sensors) et du tableau des poids. Attention la valeur de sortie doit être comprise entre 0 et 1.
4. `public double normalize(double v)` : Facultatif. Une fonction pour normaliser vos valeurs de sortie si besoin.

4 L'algorithme génétique

Adaptez l'algorithme génétique de la séance précédente à ce nouveau problème. Vous compléterez la classe *GeneticAlgorithm* qui contient un certain nombre de fonctions pour l'intégration au projet.

Vous pouvez voir 3 attributs particuliers :

— `private int populationSize = 200;` : le nombre d'individus par génération.

— `private int maxGeneration = 100;` : le nombre de générations à calculer avant l'arrêt automatique de l'algorithme.

— `private int repetitions = 1;` : combien de fois l'évaluation d'un individu doit être répété (le score est la somme des évaluations).

Les fonctions associées à ces attributs vous permettront de contrôler l'apprentissage du robot. Par exemple : les taches dont l'environnement contient beaucoup d'éléments aléatoires sont mieux évalués avec plusieurs répétitions ... au prix d'un temps de calcul plus long.

Recopiez les éléments de l'algorithme génétique de la séance précédente, mais gardez à l'esprit les différences suivantes :

— Le **génome** est maintenant un tableau de poids (type double).

— Par conséquent, les opérations de **croisement** et de **mutation** sont affectés.

— **L'initialisation** devrait être plus simple ...

— **Le score** doit être maximisé. Il est fourni sous forme de nombre flotant (double), la fonction compareTo renvoie un int ...

Complétez les fonctions suivantes :

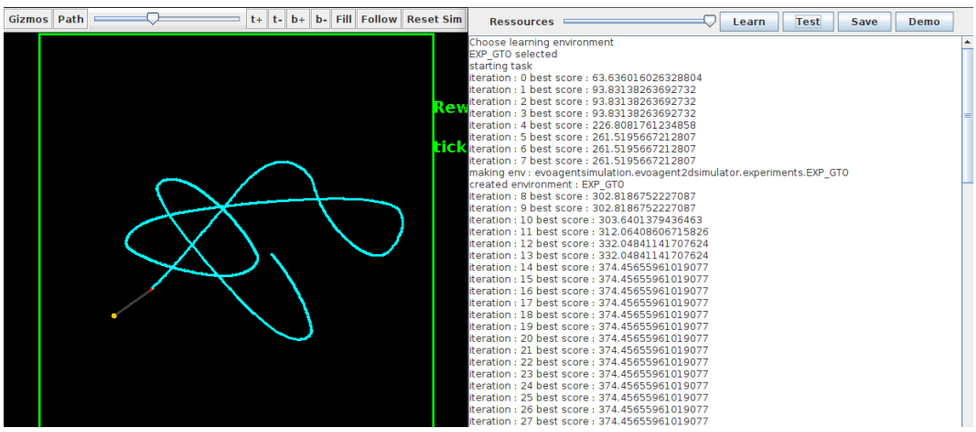
— Classe interne *Individual*

1. `public void random()` : Génère un génome aléatoire pour l'individu.
2. `public Individual crossbreed(Individual i2)` : Fonction de croisement. Retourne un nouvel individu.
3. `public void mutation(double probabilite)` : Applique une mutation sur l'individu. Vous pouvez changer la signature de la fonction comme il vous plaît.
4. `public int compareTo(Individual compare)` : De l'interface *comparable*.

— Classe *GeneticAlgorithm*

1. `public void initialise()` : Initialise l'algorithme génétique (par exemple avec une population aléatoire).
2. `public void breedNew()` : Génère une nouvelle population à partir de la population actuelle. La nouvelle population remplace celle de la liste `ArrayList<Individual> population`.

5 Apprentissage de comportements



Vous devriez maintenant pouvoir lancer l'apprentissage de comportements.

Le contrôleur que vous avez programmé s'intègre au projet, les objets SensorModule vont fournir les informations des capteurs sous forme d'un réel compris entre 0 et 1 (représentant par exemple l'orientation vers un objectif : 0.25 pour 90° à gauche, 0.5 devant et 0.75 pour 90° à droite). Les objets ActuatorModule reçoivent la commande sous forme d'un réel compris entre 0 et 1 (par exemple pour une roue : 0 marche arrière, 1 marche avant, 0.5 stop). Il ne reste plus qu'à apprendre la bonne association entre le signal des capteurs et celui des actionneurs.

Si vous lancez le main principal (*evoagentapp.EvoAgentApp*) vous obtiendrez l'interface ci-dessus.

— Learn : ouvre la liste des environnements disponible et lance l'apprentissage sur la sélection.
— Test : permet d'observer le comportement du meilleur individu de l'apprentissage en cours.
— Save : sauvegarde le meilleur individu de l'apprentissage en cours.
— Demo : charge un contrôleur à partir d'un fichier de sauvegarde et lance un environnement en observation.
— Ressources : permet de régler le nombre de processeurs utilisés par l'algorithme d'apprentissage (baissez le slider si vous voulez continuer d'utiliser votre ordinateur pendant l'apprentissage :)).

5.1 Go To Object

La première tâche définie dans `EXP_GTO` est entièrement configurée, et devrait permettre a votre agent d'apprendre un comportement qui le dirige vers un objectif (un objet). Vous pouvez lancer cet apprentissage, si votre programme ne plante pas, profitez en pour regarder comment cet environnement est configuré. Il y a 3 fonctions principales à configurer :

1. `public EvoAgentMind makeMind()` : Instancie un contrôleur. Ajoutez les capteurs et actionneurs nécessaires, définissez les propriétés du réseau.
2. `public GeneticAlgorithm makeGeneticAlgorihm(int genomeS)` : Configurez votre algorithme génétique, par exemple la taille de la population , ou toute autre option que vous aurez ajouté a votre algorithme.
3. `public void initialisation()` : Définie toutes les propriétés de l'environnement. Pour les taches suivantes vous définirez la seconde partie de cette fonction : quelles fonctions de contrôle et de récompense utiliser. Ici 5 fonctions sont utilisées :
   — `CF_NextOnTimeout(getBot(),this, 15000)` : Fonction de contrôle (CF) termine l'évaluation au bout de 15000 ticks.
   — `RW_SensorOverThreshold(getBot(), 15, getBot().sensors.get("SENSOBJ"), 0.5)` : Donne une récompense de 15 si le capteur d'objet dépasse le seuil 0.5 (un objet est détecté).
   — `RW_ClosingOnTarget(getBot(), 0.08, to)` : Donne une récompense variable (positive ou négative) si l'agent s'approche de l'objet.
   — `new RW_ForwardMotion(getBot(), 0.002)` : Donne une récompense variable (positive ou négative) si l'agent se déplace vers l'avant.
   — `new RW_Speed(getBot(), 0.00005)` : Donne une récompense variable (positive ou négative) en fonction de la vitesse de l'agent.

Essayez de modifier les valeurs des différentes fonctions de récompense et observez l'effet sur l'apprentissage. Si vous êtes à l'aise, tentez de modifier la taille de l'environnent (variable `WORLD_SIZE`) et observez l'effet indirect sur l'apprentissage.

5.2 Avoid

Vous allez maintenant configurer l'environnent `EXP_Avoid` qui entraîne l'agent à éviter des obstacles.

1. Remplissez la fonction *makeMind*, le nombre et la taille des couches cachées, sélectionnez les capteurs et actionneurs qui vous semblent pertinents.

   Les capteurs de votre agent (botType8) sont les suivants :
   — S1 ... S10 : Capteurs d'obstacles disposés en cercle autour de l'agent (sens horaire).
   — DOF : Indique si l'agent est en mouvement.
   — DISTOBJ : Distance à l'objet cible.
   — RADOBJ : Orientation de l'objet cible.
   — SENSOBJ : Es-ce que l'objet cible est à portée pour être saisi.
   — DISTDZ : Distance à la zone de dépôt.
   — RADDZ : Orientation la zone de dépôt.
   — SENSDZ : Présence dans la zone de dépôt.
   — RANDOM : Donne une valeur aléatoire.

   Les actionneurs :
   — MotL : Roue gauche
   — MotR : Roue droite
   — EMAG : Électro-aiment (pour saisir l'objet).
2. Configurez votre algorithme génétique dans la fonction *makeGeneticAlgorihm*

3. Dans *initialisation*, ajoutez les fonctions de contrôle et de récompense qui vous semblent judicieuses. Vous trouverez la liste des fonctions existantes dans le package *elements*, vous pourrez aussi définir les vôtres en vous basant sur celles proposés.
4. Facultatif : pour ceux qui aiment bidouiller, allez regarder dans la classe parente *SimulationEnvironment* les fonctions *preStepOps*, *postStepOps* ... que vous pouvez surcharger dans votre classe environnent pour faire des opérations particulières à chaque tick.

5.3 Le reste

D'autres environnements sont proposés :
— `EXP_GTOA` : (go to object and avoid) combine les taches d'évitement et d'aller vers l'objet.
— `Collect` : aller chercher l'objet et le ramener dans une zone de dépôt.
Ici le temps d'apprentissage dépasse le temps de la séance de TP :)
Si vous souhaitez aller plus loin, vous pouvez vous baser sur ces modèles pour développer vos propres environnements.

6 Approche par vecteurs ?

Les commandes motrices de l'agent son données sous forme d'un vecteur : un pourcentage de la puissance du moteur droit et du moteur gauche.

Ne serait-il pas possible d'utiliser les techniques de combinaisons de vecteurs pour réaliser des taches complexes ?

Par exemple : combiner les comportements de `EXP_GTO` et `EXP_Avoid` pour obtenir le comportement de `EXP_GTOA`.

Vous pouvez tenter de modifier votre contrôleur pour charger 2 sous contrôleurs et attribuer un poids au vecteurs de sortie (par exemple 0.5 et 0.5).

Pensez-vous pouvoir concevoir un contrôleur capable d'apprendre les poids à attribuer aux vecteurs de sortie des sous-contrôleurs ?

Si vous y arrivez, bravo, vous avez le droit d'utiliser le projet EvoAgent pour les grands :)
`https://gite.lirmm.fr/suro/evoagents2020`
`https://www.lirmm.fr/~suro/articles/Thesis_suro.pdf`