Rapport Projet INF8225

Adrien Logut (1815142), Elliot Sisteron (1807165)

Introduction

Pour ce projet, on se propose de réaliser un projet destiné à mettre en pratique nos connaissances en apprentissage automatique. Afin de réaliser cela, nous allons développer une intelligence artificielle sur un jeu de course automobile. Ce jeu a précédemment été réalisé dans le cadre d'un travail pratique du cours LOG4715, sur Unity3D. Notre IA apprendra à évoluer dans le circuit, afin d'y éviter les obstacles et trouver un chemin optimal qui lui permettra d'arriver à la première place.

Table des matières

Introduction **1**

1 Installation du jeu et Tests **3**
- 1.1 Installation de Unity . . . 3
- 1.2 Ouverture du projet - Unity . . . 3
- 1.3 Ouverture du projet - Standalone . . . 4

2 Présentation du jeu **4**

3 Algorithmes utilisés - Première Partie **5**
- 3.1 Présentation rapide . . . 5
- 3.2 Définition des états et actions . . . 5
- 3.3 Q-learning . . . 5
- 3.4 Fonction de récompense . . . 5
- 3.5 Choix des actions . . . 6
- 3.6 Programmation de l'algorithme . . . 6
- 3.7 Explication du déroulement et fichiers intéressants . . . 6

4 Résulats - Première Partie **7**

5 Algorithmes utilisés - Deuxième Partie **7**
- 5.1 Présentation rapide . . . 7
- 5.2 Définition des états et actions . . . 7
- 5.3 Fonction d'approximation . . . 8
- 5.4 Implémentation . . . 9
- 5.5 Comment tester sous Unity ? . . . 9
- 5.6 Résultats . . . 9

5.7 Pistes d'amélioration 9

Conclusion **10**

1 Installation du jeu et Tests

Parce qu'on utilise des logiciels différents dans le cadre de ce TP, voici quelques instructions pour pouvoir tester le jeu et avoir accès aux parties de code qui sont intéressantes dans le cadre de la correction

1.1 Installation de Unity

Tout d'abord, il faut installer le logiciel Unity. On pourrait juste avoir le projet en standalone, mais avoir accès à Unity permet d'afficher des informations de debuggage et des ajouts dans la création de scène au fur et à mesure de la progression qui permettent de bien voir comment l'algorithme se comporte.

Vous pouvez télécharger Unity 4.7.1 sous Windows ici et sous Mac ici

Le logiciel possède 2 licences : une gratuite et une professionnelle. La gratuite suffit amplement (et c'est d'ailleurs celle la qu'on utilise).

Si vous ne souhaitez pas installer Unity (on le comprend, c'est assez lourd), vous trouverez dans l'archive des versions Windows et Mac du jeu. (Si vous avez besoin d'une version Linux, on peut compiler dessus aussi normalement). Vous ne verrez donc que l'algorithme apprendre sans plus d'informations (mais c'est déjà pas mal).

Le logiciel installe aussi MonoDevelop (son propre IDE pour coder en C#), si vous ne souhaitez pas cela et voulez ouvrir le code avec votre propre IDE vous pouvez désactiver son installation.

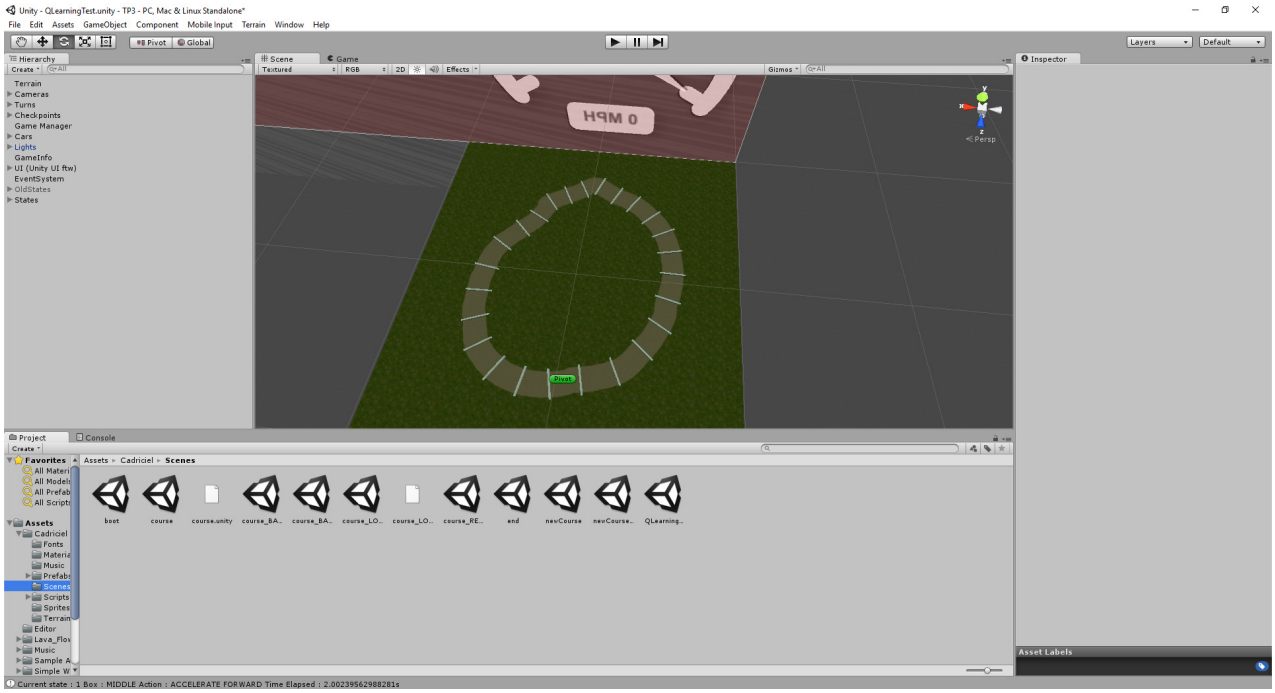
1.2 Ouverture du projet - Unity

Lorsque que Unity est installé, il faut déjà le lancer et activer une licence gratuite. Un compte doit être créé, c'est assez rapide à faire et on s'excuse d'avance des inconvénients que porte notre projet.

A partir de ça, il faut ouvrir le projet. Dans l'archive vous trouverez le dossier Projet qui faudra indiquer à Unity comme racine du projet. S'en suit un traitement qui se fait seulement lors du premier lancement. Unity génère ses fichiers nécessaires à son utilisation.

Vous vous retrouvez ensuite face à pas grand chose. Pour pouvoir voir le circuit d'apprentissage il faut faire *File -> Open Scene* puis trouver le fichier *./Assets/Cadriciel/Scenes/QLearningTest.unity*.

Vous vous retrouvez face à cet écran :



A partir d'ici c'est tout simple, appuyer sur Play (en haut au milieu) permet de lancer le jeu (et passe dans la vue Game, qui voit à travers la caméra que le joueur verrait s'il jouait vraiment).

On peut aussi pendant que le jeu est en route repasser à la vue Scene (ongle juste à côté de Game) pour voir toutes les infos apparaître et cliquer sur l'onglet "Console" pour voir les infos de débuggage.

On peut aussi naviguer dans la scène 3D pour mieux voir(Toujours dans l'onglet Scene). Les raccourcis sont :

— Clic droit pour faire tourner la caméra

— Clic Molette (Bouton de la molette enfoncé) pour faire bouger la camera en gardant une direction fixe

— Molette pour zoomer

Cela devrait suffire pour naviguer rapidement dans la scène.

1.3 Ouverture du projet - Standalone

Si vous ne voulez pas installer Unity, il suffit de lancer l'executable dans le dossier "Standalone" et vous pouvez admirer le résultat !

2 Présentation du jeu

Tout d'abord un rapide point sur la technologie utilisée pour ce projet et son contexte. Ce projet a été fait sous Unity3D en version 4.x, en programmant en C#. Il s'agit d'une simulation de course de voitures, réalisé dans le cadre du cours LOG4715. Il a été repris dans le cadre de ce TP comme base déjà testée pour mettre en place une IA de voiture qui va apprendre à suivre un circuit de manière la plus optimale possible.

3 Algorithmes utilisés - Première Partie

3.1 Présentation rapide

La base de notre travail repose sur l'algortithme Q-learning qui permet d'avoir un apprentissage par renforcement. En effet, notre IA de voiture va essayer d'explorer les actions qui sont possibles pour elle, et tenter d'atteindre la ligne d'arrivée tout en restant sur la piste.

3.2 Définition des états et actions

L'algorithme se base sur un ensemble d'états noté $\mathcal{S}$ et un ensemble d'actions noté $\mathcal{A}$. Ici notre ensemble d'états est visible sur la scène. Un état correspond à une boîte grise. Plus précisément, cela correspond à 3 états. Chaque boîte est divisée en 3 états, sa partie gauche, droite et centrale. La voiture va donc passer dans un nouvel état lorsqu'elle va rentrer dans une nouvelle boîte.

On a donc $Card(\mathcal{S}) = 3 * N$, N étant le nombre de boîtes dans la scène.

Pour ce qui est des actions, une voiture peut accélérer, freiner ou se laisser aller. Ainsi que tourner à droite, à gauche ou aller tout droit. Dans le cadre de notre TP, la simulation prend en entrée 2 valeurs h et v avec $h \in [-1, 1]$ et $v \in [-1, 1]$. h correspond à l'inclinaison du volant. -1 pour braquer à gauche et 1 pour braquer à droite. De même v défini l'appui sur la pédale d'accélération ou de frein. -1 pour piler (à fond sur le frein) et 1 pour pied au plancher.

Dans la première simulation, on a enlever la possibilité de freiner (Freiner c'est pour les faibles). On a discrétisé à 3 actions possibles pour accélération $v \in \{0, 0.5, 1\}$ et 5 actions possibles pour le volant $h \in \{-1, -0.5, 0, 0.5, 1\}$. On a donc $Card(\mathcal{A}) = 3 * 5 = 15$

3.3 Q-learning

L'algorithme de Q-learning se base sur une fonction Q définie ainsi : $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

Et l'algorithme se base sur cette formule :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}}(Q_t(s_{t+1}, a)) - Q_t(s_t, a_t))$$

qui met à jour la valeur pour Q au fur et à mesure de ses explorations où :

— α : Facteur d'apprentissage

— γ : Facteur d'actualisation

— $\max_{a \in \mathcal{A}}(Q_t(s_{t+1}, a))$: Valeur optimale espérée

— R_{t+1} : Récompense (positive ou négative)

L'algorithme va donc passer d'états en états et choisir des actions puis voir quelle récompense il obtient.

3.4 Fonction de récompense

Un élément clé de l'algorithme, qui fait qu'il va bien appendre ou non est la fonction de récompense. Dans la première simulation elle dépend de la distance entre la voiture et le centre de la boîte lorsqu'elle rentre dans la boîte ainsi que l'angle que fait la voiture avec la normale à la boîte (sens du circuit en somme). Plus l'angle est elevé, plus la valeur retournée est faible.

L'agent a aussi une récompense positive s'il est au centre, nulle s'il est à la limite avec la boîte gauche ou droite et négative au delà. Cela permet à l'agent d'essayer de viser la boîte du milieu.

De plus si la voiture met plus d'un temps constant défini (2s dans la première simulation) ou rentre dans la boîte avec un angle supérieur à 90° (à l'envers), ou encore qu'elle saute une boîte, la fonction renvoie une forte valeur négative pour punir l'agent.

Enfin si l'agent atteint la boîte finale, on lui octroie une forte récompense positive.

3.5 Choix des actions

A chaque nouvel état, l'agent doit choisir une action, qu'il tiendra tant qu'il n'aura pas atteint un autre état. Cette action est choisie simplement, en prenant dans un état donné s_t, l'action a qui maximise $Q_t(s_t, a)$.

Dans le cas où la voiture saute une boite, ou prends trop de temps pour aller à l'autre boite, l'agent est renvoyé au départ et est pénalisé (comme dit précédemment).

Cette politique est appelée *On-Policy* car la politique de choix des actions pour l'apprentissage est la même que celle qu'on essaye d'appprendre. On peut l'améliorer pour plus d'exploration à une politique nommée ϵ-*policy* qui est la même mais avec un choix aléatoire d'actions de probabilité ϵ. Elle n'a cependant pas été retenue ici car cette exploration ne convient pas à la modélisation de notre projet. En effet, un choix d'action aléatoire à chaque état bloque l'avancée de la voiture. S'il prend une action aléatoire 1 fois sur 10 par exemple, elle a plus de chance de crasher la voiture et la faire recommencer que de trouver une meilleure voie. Au final la voiture n'apprends en pratique que sur les premiers états et ne parvient rarement à aller plus loin que les 10 premiers états avec cette politique.

3.6 Programmation de l'algorithme

L'algorithme a été codé en C#, qui est le langage de scripting utilisé par Unity (pas le seul, Javascript peut aussi être utilisé entre autres). Le défi majeur a été de savoir comment stocker les valeurs de notre fonction Q et surtout pouvoir garder les résultats entre plusieurs sessions de jeu, pour éviter à l'agent de devoir tout réapprendre.

Pour stocker les valeurs, on utilise un tableau bi-dimensionnel de doubles de taille $Card(\mathcal{S}) \times Card(\mathcal{A})$. On retrouve les états en lignes et les actions en colonne.

Pour enregistrer et charger les valeurs, C# met à disposition dans ses librairies standards une classe appelée *BinaryFormatter* qui permet de faire une *serialization* dans un fichier. Tout simplement, on envoie notre matrice de valeurs Q et il va s'occuper de l'écrire dans un fichier binaire. On peut aussi tout simplement lire le fichier binaire et le cast en notre matrice de valeur pour charger le fichier. C'est très simple et intuitif. Pas de gros parseurs à coder !

3.7 Explication du déroulement et fichiers intéressants

Toute l'IA est contenue dans 3 scripts, situés dans *./Assets/Cadriciel/Scripts/IA*. Le script Q-learning importe de manière générique l'algorithme Q-learning. Il peut être réutilisé avec n'importe quelle IA, car sa fonction de mise à jour reçoit toutes les valeurs nécessaires).

Le gros du projet est donc dans LearningAgent, qui est une classe qui dérive de MonoBehaviour, la classe Mère d'Unity. Dans les grandes lignes, cette classe mère permet de redéfinir des fonctions qui vont s'exécuter à différents moments de l'application. Par exemple la fonction *Start* est appelée qu'une seule fois au chargement de la scène, alors que les fonctions *FixedUpdate*et *Update* sont appelées à chaque frame.

C'est cette classe qui s'occupe de générer la fonction de récompense et s'occupe de générer les valeurs h et v dépendamment de l'action courante, qui vont être envoyés à la fonction *car.Move* qui s'occupe de la simulation.

Enfin la classe StateScript est le script qui est "accroché" à chaque boite dans le jeu et qui possède une fonction *OnTriggerEnter* qui est simplement appelée quand quelque chose entre dans la boite. On vérifie donc bien qu'on effectue une action que si c'est bien la voiture qui rentre dans la boite (et pas un autre élément du jeu) et on calcule les différentes valeurs (distances et angles) qui sont ensuite envoyé à l'agent pour mettre à jour ses états.

Cette classe possède aussi la fonction *OnDrawGizmos* qui va s'occuper de tracer les différentes figures géométriques dans notre scène pour bien représenter les différentes valeurs calculées (point d'impact, angle etc...)

4 Résulats - Première Partie

Lors de la première simulation, l'agent a mis environ 10 min pour arriver au bout de la course, de manière non optimale (en conduisant plutôt comme une personne totalement saoule) mais y arrive tout de même.

En laissant l'algorithme tourner, on voit bien que la voiture apprend de ses erreurs.

5 Algorithmes utilisés - Deuxième Partie

5.1 Présentation rapide

Après réflexion, il convient que la première modélisation n'est pas la meilleure. En effet discrétiser les états avec juste les boites n'est pas satisfaisant. On voudrait avoir plus de fléxibilité, que notre voiture sache se débrouiller en n'importe quel point du circuit.

Il faut donc revoir la modélisation, et sans doute adapter notre méthode.

5.2 Définition des états et actions

Avec la nouvelle modélisation, on garde le même ensemble d'actions, il semble que cette discrétisation soit plutôt correcte.

Notre ensemble d'états va lui être modifié, et passer à un ensemble continu, où les valeurs ne sont plus discrétisées mais continues. Un etat sera donc représenté par :

— La vitesse de l'agent
— La distance de l'agent au centre de la piste
— L'angle entre l'agent et la direction du circuit
— L'angle entre direction du circuit et sa direction quelques metrès plus loin

La vitesse est importante pour définir un état, en effet on ne va pas réagir de la même manière si on est quasiment à l'arrêt ou que l'on est lancé à pleine vitesse. De même de la distance de l'agent à la piste. Si on est au bord de la route ou au centre de la piste c'est différent. Pour ce qui est de l'angle entre l'agent et la piste, c'est pour détecter les virages à court terme ou remettre la voiture dans le droit chemin, tandis que l'angle entre 2 points du circuits sert à détecter les virages (et surtout ceux en tête d'épingle).

Une hypothèse peut être faite que ces caractéristiques suffiraient à faire évoluer la voiture dans n'importe quel circuit avec assez d'entrainement. En effet lorsqu'on y réfléchit bien, c'est en règle générale toutes ces infos et seulement elles qui nous font prendre des décisions, nous humains, lors d'un jeu de course. On verra si cette hypothèse se validera.

5.3 Fonction d'approximation

Une première idée sachant que nos états sont définis par des variables continues seraient de discrétiser chaque valeur en plusieurs intervales, permettant d'utiliser l'algorithme de Q-Learning. Mais en choisissant un intervale de 5° pour les angles, 1 pour la distance (distance $\in [0, 7]$, valeurs empiriques) et 0.1 pour la vitesse ($v \in [0, 1]$), on aurait $\frac{360}{5}^2 * 7 * 10 = 362880$, ce qui commence à faire beaucoup d'états. Si on combine ça à nos 15 actions, on a une matrice de 362880*15 doubles ce qui nous donne une matrice de 43Mo.

On voit bien que ce n'est plus raisonable. Autant d'états ne pourront jamais être tous visités plusieurs fois pour avoir une valeur correcte en un temps raisonnable. Il faut donc trouver une autre technique.

Ainsi au lieu d'avoir la fonction $Q(s, a)$ représentée par une matrice, on va utiliser une approximation de fonction $Q(s, a|\theta)$. Et pour l'implémenter, on utilise un réseau de neurones.

L'implémentation du réseau est la suivante :

— 1 couche d'entrée, 1 couche cachée et 1 couche de sortie.
— 4 entrées, correspondant à nos 4 variables d'état, celles si sont normalisées (entre -1 et 1).
— 30 neurones dans la couche cachée, avec la fonction d'activation tanh.
— Card($\mathcal{A}$) neurones en sortie, correspondant à la valeur approximée de Q pour chaque action pour un état donné.
— La fonction d'activation en sortie est linéaire, ce qui permet d'avoir des valeurs de Q sur $\mathbb{R}$ tout entier.
— Les poids sont initialisés de manière random, et entre -0.1 et 0.1
— Les biais sont mis à 0

Comme d'habitude, une déscente de gradient est mise en place pour mettre à jour notre réseau et apprendre. La fonction de perte à minimiser est celle associée au Q-learning :

$$L_i(\theta_i) = (y_i - Q(s, a|\theta_i))^2$$

avec $y_i = r + \gamma \max_{a'}(Q(s', a'|\theta_{i-1}))$

La mise à jour se fait après chaque observation (une action, un nouvel état, observation de la récompense, mise à jour du réseau), ce qui correspond à une descente de gradient stochastique. Sur le vecteur de sortie, on va faire apprendre notre réseau sur le même vecteur de sortie donné

par la forward propagation mais avec la valeur correspondant au neurone de l'action prise et observée qu'on change pour minimiser $L_i(\theta_i)$

5.4 Implémentation

Unity limite toujours au C# (ou Javascript...) il fallait donc implémenter un réseau de neurones en C#. On a fait le choix d'utiliser un Framework qui s'appelle Aforge.Net qui est un Framework très complet sur plusieurs domaines de l'IA dont le Maching Learning. Il a falu cependant jouer fortement avec les paramètres pour avoir un réseau comme on voulait (les valeurs par défaut n'était vraiment pas bonnes). Il a fallu aussi coder nos fonctions d'activations (linéaire et tanh). Heureusement, le framework met à disposition une interface très simple, implémentée par nos classes fonctions d'activations pour pouvoir fonctionner avec le réseau et le framework.

Tout l'apprentissage est concentré dans le fichier *LearningAgentBis.cs*

5.5 Comment tester sous Unity ?

Etant donné que les 2 IA sont fournies, il est possible de switcher d'IA pour tester les 2 (Q-learning et Fonction d'approximation/NN). Ainsi :

- Pour le Q-Learning, il faut sur le GameObject "Agent" dans "Cars" cocher le script *LearningAgent* et décocher le script *LearningAgentBis* et *TextureDetector*. De même, il faut sélectionner tous les GameObjects "State" dans "States" et cocher *StateScript* et décocher *StateScriptBis*
- Pour le NN, il faut faire l'inverse !

5.6 Résultats

Les résultats sont cependant décevants pour l'espoir qu'on y avait porté. Il semble que les paramètres soient très durs à choisir et que les états en eux-mêmes ne sont soit pas assez représentatifs soit ne varient pas assez numériquement entre 2 états qui intrinsèquement sont bien différents pour le joueur humain.

L'agent se retrouve vite coincé avec une action qu'il exécute quasiment tout le temps et ne fait qu'apprendre sur cette action. Tant bien un coup d'aléatoire dedans fait qu'une petite variation apparaît, il reste dans sa routine.

5.7 Pistes d'amélioration

Il est possible de réfléchir encore plus sur les états, en rajoutant de nouvelles caractéristiques (positions ?), en jouant sur le learning rate, les récompenses, le momentum etc...

Le fait de devoir dépendre du moteur physique de Unity rend les expériences très longues, et non propices à tester en masse des paramètres. Décortiquer le code ou avoir une approximation de la simulation, qu'on puisse calculer nous même nous permettrait d'augumenter drastiquement la vitesse des calculs. Le jeu tourne à 1200fps, donc le moteur physique est capable de faire tous les calculs en $\frac{1}{1200}$s maximum au lieu de $\frac{1}{60}$s, le gain serait grand (fois 20).

Conclusion

Avec ce TP, nous avons pu mettre en pratique nos nouvelles connaissances en apprentissage automatique, notamment en implémentant l'algorithme du Q-learning et de l'approximation de fonction.

Cela montre que les réseaux de neurones sont très puissants mais ne sont pas non plus une solution magique facile. Il faut vraiment réfléchir au problème, prendre de bonnes valeurs de paramètres et étudier l'évolution au cours du temps pour affiner les réglages. En tout cas ça aura été avec grand plaisir qu'on ait pu aller plus loin sur notre TP de jeux video et améliorer encore le code qu'on avait la session précédente !