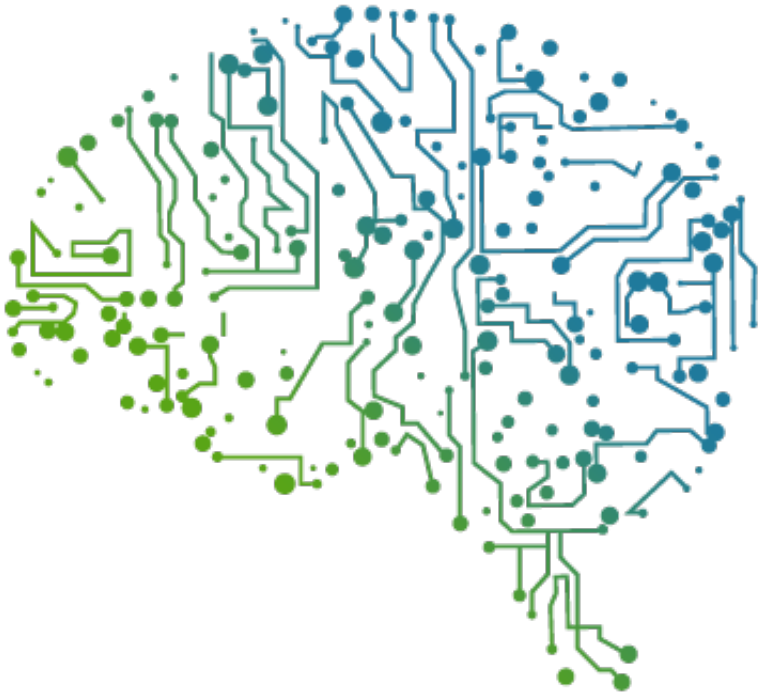
CFPT

Travail de diplôme



T-Rex Ceptional

Loris De Biasi

Classe :

T.IS-E2A

5 mai 2017

1 Introduction

Le *Machine learning*[1] est devenu de plus en plus important et de plus en plus utilisé ces dernières années. Un exemple concret serait les voitures automatiques dont les constructeurs n'arrêtent pas de vanter les mérites, Google adWords[2] qui génère plus de 60 milliards par année ou bien encore les moteurs de recommandations telles que celui d'Amazon ou de Spotify qui permettent d'avoir des recommandations selon ce que vous écoutez/regardez.

Mon projet aura pour but de me plonger dans ce principe pour ainsi apprendre son fonctionnement et comprendre plus en profondeur ses mécanismes. Pour cela, j'ai choisi de développer une *IA*[3] pour le jeu du T-Rex présent sur *Google Chrome* lorsque qu'aucune connexion à internet n'est disponible. Pour être plus précis, je ne vais pas créer une *IA*, mais plutôt créer un cerveau "vierge" qui, en ayant connaissance des règles du jeu, devra apprendre de lui-même.

Étant donné que je souhaite apprendre un maximum de choses, je ne me servirai d'aucune bibliothèque lors de ce projet.

1.1 Résumé

1.2 Abstract

1. "apprentissage automatique" en français
2. régie publicitaire de Google
3. "dispositifs imitant ou remplaçant l'humain dans certaines mises en œuvre de ses fonctions cognitives"[1]

Table des matières

2 Cahier des charges

REPUBLIQUE ET CANTON DE GENEVE
DEPARTEMENT DE L'INSTRUCTION PUBLIQUE, DE LA CULTURE ET DU SPORT
ENSEIGNEMENT SECONDAIRE II POSTOBLIGATOIRE
Centre de formation professionnelle technique
Ecole d'informatique

Énoncé travail de diplome 2016-2017

Données candidat	Données enseignant
Nom : De Biasi Prénom : Loris Téléphone : 078 670 81 88 E-mail : loris.debiasi@gmail.com	Nom : Wanner Prénom : Nicloas E-mail : nicolas.wanner@edu.ge.ch

Titre du projet

Intelligence artificielle apprenant à jouer à un jeu vidéo et ayant pour but d'aller le plus loin possible.

Objectifs du projet

Création d'une IA ayant pour objectif d'apprendre à jouer au T-rex runner de chrome.

- Aucune bibliothèque ne sera utilisée.
- Affichage du jeu avec l'IA en direct.
- L'IA ne connaît pas les règles du jeu.
- L'IA connaît les touches permettant d'effectuer une action, mais ne connaît pas l'action effectuée.
- Affichage des touches pressées par l'IA.

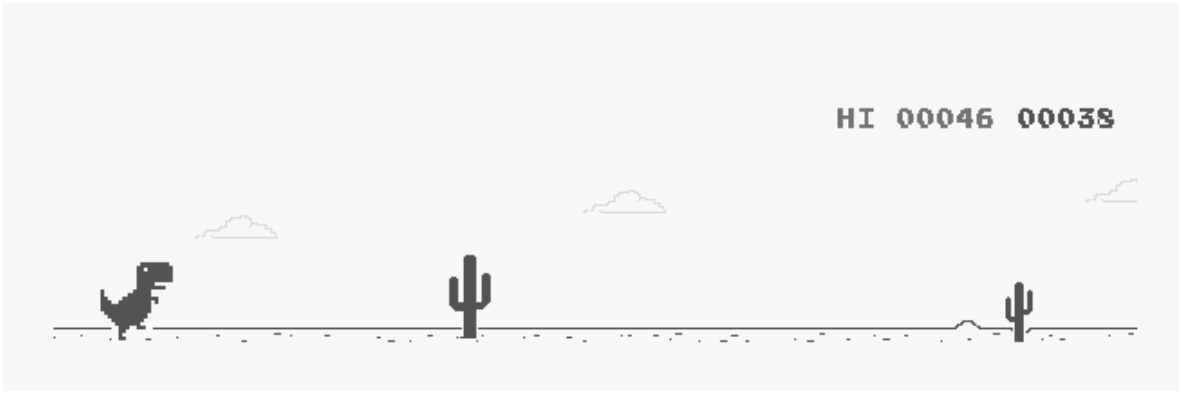


FIGURE 1 – Jeu du T-rex

Description détaillée

Le but est de créer une IA qui aura la capacité d'apprendre à jouer au jeu *T-rex runner* présent sur le navigateur *Google chrome* lorsqu'il n'y a aucune connexion à internet.

Le principe de ce jeu est simple, vous contrôlez un personnage (T-Rex) qui avance automatiquement. Devant vous apparaissent soit des cactus (obstacle au sol), soit des ptérodactyle (ennemi dans les airs). Les seules actions possibles sont soit de se baisser, soit de sauter. Le but est donc d'arriver le plus loin sans se faire toucher une seule fois.

L'IA utilisera deux principes, le "machine learning" et les algorithmes génétiques.

Le "machine learning" sera là pour créer un "cerveau" (réseau de neurones) à cette IA, lui permettant ainsi de définir des règles. Dans ce cas, il ressemblera à quelque chose similaire à cela :

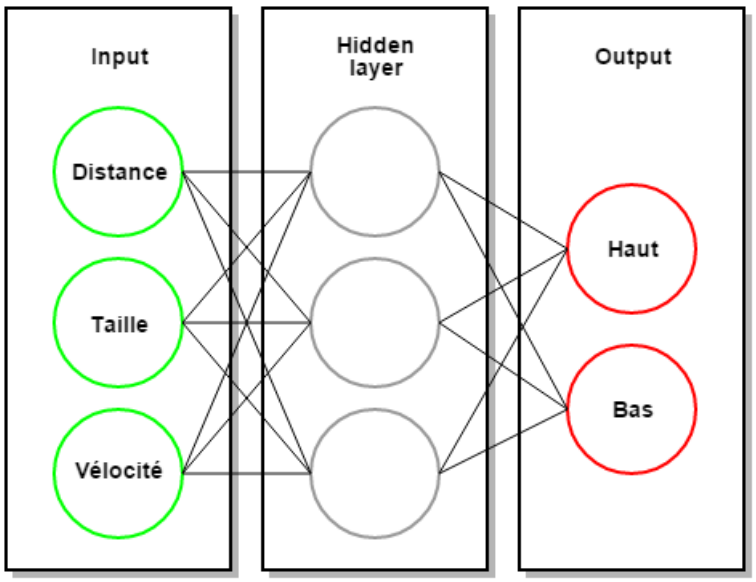


FIGURE 2 – Exemple du réseau de neurones du T-rex

Dans le rectangle de gauche se trouve la zone "input", permettant de définir quelles caracté-

ristiques du jeu influeront sur les actions du T-Rex. Dans ce cas, trois choses sont requises :

- **La distance** : permettant de connaître la distance séparant le T-Rex d'un obstacle
- **La taille de l'obstacle** : permettant de connaître la taille en largeur de l'obstacle le plus proche
- **La vélocité** : permettant de connaître la vitesse actuelle du T-Rex.

Lorsque le T-Rex court, ces trois caractéristiques sont requises, car si le T-Rex ne connaît pas la distance le séparant de l'obstacle, il ne saura jamais quand sauter. De même si le T-Rex ne connaît ni sa vélocité ni la taille de l'obstacle, il ne comprendra jamais qu'il faut un timing précis pour un obstacle plus long ou bien avec une vitesse plus élevée.

Dans un premier temps, ces informations seront récupérées directement en ayant accès au code, mais il est envisageable de le faire en examinant le jeu, sans avoir directement accès à celui-ci.

Dans le rectangle du milieu se trouve la zone "Hidden layer". Cette zone contient une seule chose, des neurones. Ces neurones sont en faits simplement des fonctions qui prennent en entré un nombre, puis le passe dans la fonction et le transmet, soit au neurone suivant, soit à la troisième zone ("output"). Ces fonctions sont appelées "fonction d'activation" et ils en existent énormément de type différente, permettant toute quelque chose de différent. Les plus connus étant : Sigmoig, ReLu, Tanh et Linear. Le nombre de neurone est-ce que l'on appelle un *hyperparameter*, cela signifie qu'ils ne peuvent pas être appris par le réseau de neurones et qu'il faut les renseigner à la main. Les neurones présents dans l'image du haut ne sont qu'un exemple, le réseau de neurones final ne ressemblera probablement pas à cela et contiendra certainement plus de neurones ainsi que plusieurs "Hidden layer". Plus un réseau est complexe, plus il permet d'avoir un résultat précis, mais plus il prend de temps, il faut donc choisir le bon ratio temps/résultat.

Avant de parler du troisième rectangle, il faut savoir que tous les traits entre les cercles représentent des "poids", il s'agit d'un nombre que l'on va utiliser avec le nombre du neurone précèdent pour la passer dans la fonction d'activation. Ces "poids" sont les seuls choses qui varient dans un réseau de neurones "standard".

Finalement, la troisième partie, appelé "Output", est simplement le résultat de tous les calcules précèdent. Selon le résultat obtenu sur le neurone de sortie, la touche sera pressée ou non. Par exemple, si le résultat de sortie est "0.6" et que nous avons défini que la touche s'active seulement si un nombre supérieur à "0.5" est trouvé, alors la touche sera activée. Il faut savoir que les résultats de sortie de chaque neurones sont "normalisés" en général entre 0 et 1, grâce aux valeurs minimales et maximales du réseau de neurones.

Parlons maintenant de l'algorithme génétique. Il permet de créer plein de réseau de neurones, appelé "Genome". Comme dis précédemment, la seule variable d'un réseau de neurones sont ses poids, nous allons donc créer plein de réseau de neurones avec des poids différents. Ces réseau de neurones sont ensuite testés sur le jeu, puis, en faisant de la sélection naturelle, nous allons jeter ceux ayant le moins bon score. Ensuite, nous allons faire se que l'on appelle de l'enjambement ("cross over" en anglais). Cela consiste à prendre des parties de chacun des réseaux de neurones restant pour les combiner en un seul. Par la suite, nous allons choisir des valeurs aléatoire auquel seront ajoutées des valeurs aléatoire (étape appelée "mutation"). Il ne reste plus qu'à répéter ce processus jusqu'à avoir le nombre de genome souhaité, créant ainsi une nouvelle génération. Tout ce processus (sélection, enjambement et mutation) est répété jusqu'à avoir le résultat escompté.

Inventaire des étapes du projet

Le total des heures correspond à 312 heures (39 x 8h00).

Début : Mercredi 5 avril 2017
Reddition intermédiaire (documentation + poster) : Vendredi 5 mai 2017
Reddition intermédiaire (résumé + abstract) : Vendredi 19 mai 2017
Reddition finale : Lundi 12 juin 2017

Inventaire du matériel

- PC + 2 écrans
- clavier USB
- souris USB

Inventaire des logiciels

- Système d'exploitation : *Windows 10* ou *Linux mint*
- Outils de développement : *Sublime text 3*
- Langage de programmation : *Javascript/HTML/CSS*

Délivrables

- 1 carnet de bord
- 1 exemplaire papier de la documentation technique
- 1 exemplaire papier du mode d'emploi
- 1 CD contenant les sources (projet logiciel)
- d'autres exemplaires papier de la documentation technique et du mode d'emploi en fonction des demandes des experts
- mis à part le carnet de bord, tous les documents seront également restitués sur le serveur *Moodle*

Signatures	
Date :	Date :
Candidat :	Enseignant :

3 Étude d'opportunité

L'utilisation du *Machine learning* dans les jeux vidéo est quelque chose de très peu utilisé et cela est dû à plusieurs raisons. Premièrement, une *IA* utilisant le *Machine learning* ne peut pas être contrôlée, elle effectuera les actions qu'elle aura apprises, la rendant ainsi imprévisible. De plus, que ce soit pour le développement ou lors de l'exécution, cela n'est pas rentable en termes de temps. Elles sont plus complexes à développer et plus lente à l'exécution, mais permettent d'obtenir des résultats qui seraient impossibles à coder "a la main".

Néanmoins, il existe tout de même des jeux vidéo utilisant le *Machine learning*. Il s'agit cependant, très souvent, de personne qui développe par-dessus des jeux existants. En voici quelques exemples.

3.1 Flappy learning

Ce projet est une reprise du jeu *Flappy bird* qui est un jeu vidéo sorti sur smartphone en 2013. Dans ce jeu, vous contrôlez un oiseau devant naviguer entre des tuyaux et le but est d'aller le plus loin possible sans les toucher. Cette *IA* a été réalisée en *Javascript* sans aucune bibliothèque et utilise les principes de *Neuroevolution* et d'*Algorithme génétique*. Tout son code est disponible sur *Github*[2]. Malheureusement, aucune documentation n'est présente, il n'y a que le code. La personne ayant créé cette *IA* se nomme apparemment Vincent BAZI et a un master en informatique.

FIGURE 1 – Flappy Learning

La différence entre ce projet et le mien, mis à part le jeu, est que le réseau de neurones utilisé est beaucoup plus petit. *FlappyLearning* utilise un réseau de 3 neurones : 1 input, 1 hidden et 1 ouput. Le fait d'avoir un réseau de neurones plus petit ne change pas le code des réseaux de neurones mais plutôt le temps que l'*IA* va prendre à trouver comment avancer. Il implémente également le fait de pouvoir exécuter plusieurs oiseaux en parallèle pour ainsi obtenir un résultat plus rapidement. C'est une chose qui n'est pas prévue dans mon cahier des charges, mais que je souhaite implémenter (pas forcément sous cette forme). Ensuite, les principes du jeu *Flappy bird* son plus simple que le jeu du *T-Rex*. Là où le jeu du *T-Rex* possède plusieurs éléments variables tels que la vélocité, la taille d'un obstacle, sa position sur l'axe Y ou encore la distance par rapport à celui-ci, *Flappy bird* ne possède que la distance par rapport à un objet ainsi que sa position sur l'axe Y. Rendant ainsi le réseau de neurones bien moins complexe.

3.2 MarI/O

Ce projet est celui qui m'a donné envie d'utiliser le principe du *Machine learning* dans un jeu vidéo. Il reprend également un jeu vidéo qui est cette fois-ci *Super Mario World*, jeu sorti en 1990. Ce projet est certainement le plus gros projet fait par quelqu'un d'indépendant. Il a été créer par une personne connue sur *Youtube* se nommant *SethBling*. Les seules informations disponibles à son sujet sont qu'il a travaillé chez microsoft en tant que développeur sur *Xbox* et *Bing*.

Premièrement, la plus grosse différence avec tous les autres projets dont je vais parler est que tout cela a été fait sans avoir accès au code. Tous les autres projets cités ici ont tous un accès direct au code et aux valeurs souhaité, alors que MarI/O est une *IA* écrite en LUA pour un jeu qui n'est pas *open source* et qui est prévu pour fonctionner sur *Super Nintendo*.

Pour ce faire, cette personne a donc utilisé un emulateur (Bizhawk[8]) qui permet d'ajouter des scripts fait en LUA. Hormis le fait qu'il a dû développer un réseau de neurones énorme dû au nombre de variables et de bouton, il a dû synthétiser l'interface pour ne récupérer que ce qui est utile pour l'*IA*. Une autre des grosses différences est qu'il utilise un réseau de neurones de type NEAT [4] qui en plus d'altérer les paramètres "Weight" comme tous les types de réseau de neurones, altère également la structure du réseau, ce qui en fait un réseau de neurones très spécial et complexe. Ce projet est colossal et aura pris 2 mois au total (apprentissage de NEAT + développement).

4. NeuroEvolution of Augmenting Topologies[9]

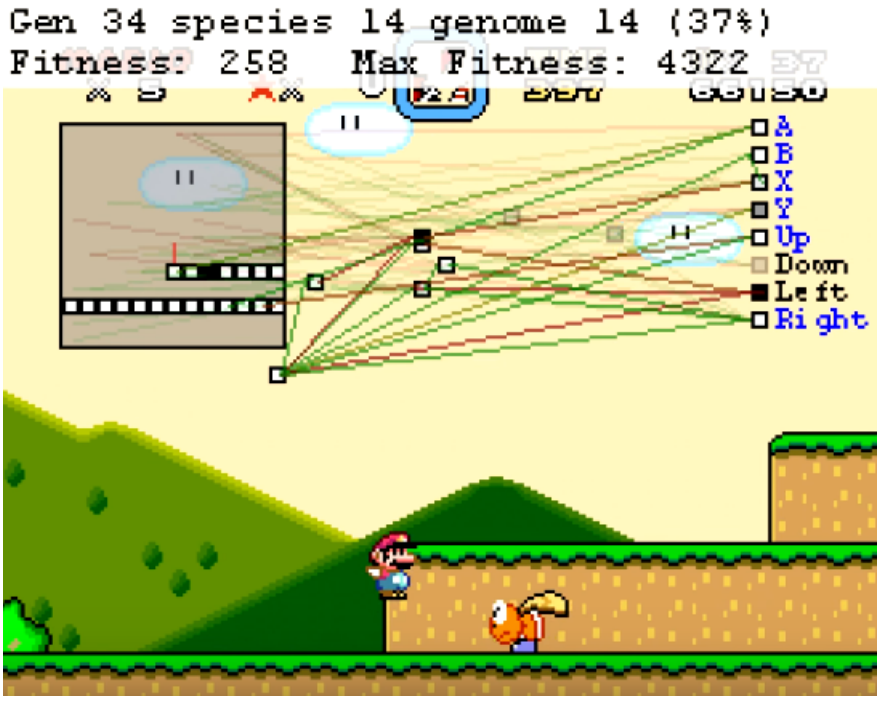


FIGURE 2 – MarI/O

Il est aussi important et intéressant de noter que ce projet aura pu être réutilisé (avec quelque léger changement) pour fonctionner sur le jeu *Super Mario Kart* qui est un jeu complètement différent dans lequel on contrôle un kart, ainsi que le jeu *Super Mario Bros* dans lequel un *glitch*[5] a été trouvé grâce à celui-ci. Dû au fait que l'*IA* ait accès à plus de chose qu'un joueur (elle a accès à ce qui se passe réellement, pas ce qui est affiché à l'écran), l'*IA* s'est rendu compte qu'il était possible de passer par un passage normalement bloqué par un ennemi, lui faisant ainsi gagner du temps.

5. "terme employé pour désigner un bogue dans un jeu vidéo, où un objet animé a un comportement erroné (par exemple : passage au travers des murs, « téléportation » inattendue), qui peut-être exploité pour finir un jeu le plus vite possible"[10]

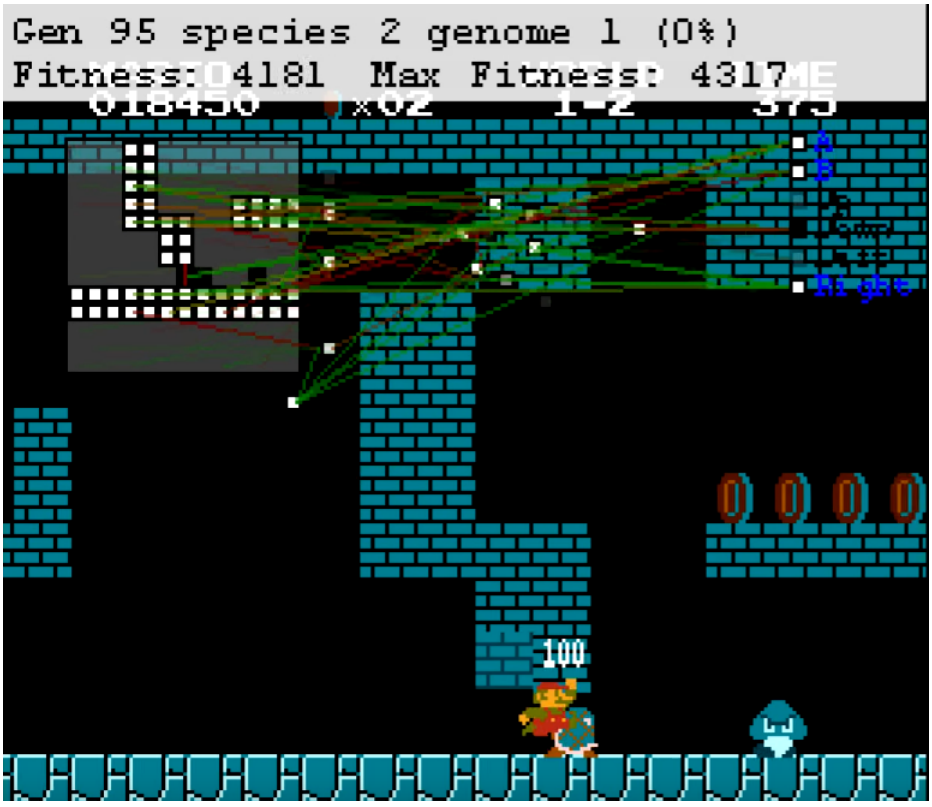


FIGURE 3 – Glitch trouvé dans *Super Mario Bros*

Malheureusement, encore une fois, aucune documentation n'est disponible, je ne pourrai donc pas en dire plus sur ce projet.

3.3 Forza Drivatar

Forza Drivatar est une *IA* utilisant le *Machine learning* qui a été créer par le studio de recherche de Microsoft au Cambridge dans le but d'être utilisé dans le jeu vidéo *Forza Motorsport*(jeu de course de voitures). Elle a été créée dans le but d'apprendre à se comporter comme un vrai joueur pour pouvoir le remplacer (une option est présente pour activer la conduite automatique) et éviter de rendre les *IA* trop prévisibles et similaires. Pour ce faire, elle récupère les informations de joueur humain et envoie ces informations dans une simulation.

Ce système fonctionnait, dans les 4 premiers opus, en local et donc calculé directement sur la console. Depuis *Forza Motorsport 5*, ce système est maintenant effectué depuis le *cloud* grâce au *Cloud Computing*[6], permettant ainsi d'augmenter grandement la puissance de calcul.

Ce projet est le seul que j'ai pu trouver qui a été développer "dans le jeu" et non pas en tant qu'ajout une fois celui-ci terminé. Malheureusement, aucune information sur le réseau de neurones n'a été donnée. Néanmoins, le réseau utilisé doit être semblable au

6. "exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants"[11]

réseau de neurones d'une voiture autonome et doit probablement être du *Reinforcement Learning*.

3.4 DeepMind et Starcraft

DeepMind est une entreprise ayant été racheté par Google qui est spécialisée dans l'*IA*. Son principale objectif est de développer une *IA* qui a pour objectif de résoudre même les problèmes les plus complexes sans pour autant lui dire comment faire. D'après eux, les jeux vidéo seraient le meilleur environnement pour faire cela, car elles leur permettraient de visualiser rapidement l'intelligence de l'*IA*.

DeepMind n'en est pas à ses premiers essais puisqu'ils sont les développeurs de *AlphaGo*. *AlphaGo* est la seule *IA* ayant réussi à battre un joueur professionnel (par ailleur, le champion du monde) de *Go*. Programmer un joueur de *Go* est quelque chose d'extrêmement complexe puisque, pour comparer à un jeu plus connu, le nombre de parties différentes possibles de *Go* est estimé à 10^{600}, là où les échecs en ont 10^{120}.

Pour revenir au sujet principal, *DeepMind* a choisi de développer une *IA* sur *StarCraft* (jeu de gestion de troupes), trouvant qu'il serait un bon environnement de développement dû au fait que les compétences requises pour jouer à *StarCraft* pourraient être utilisée pour effectuer des tâches dans la vrai vie.

Ce projet est le seul dont je parle qui utilise un réseau de neurones de type *Reinforcement Learning*(ou en tout cas le seul ou cela est affirmé). Ce type de réseau de neurones est utilisé lorsque l'on ne sait pas comment classifier les données, mais que nous savons quel résultat nous souhaitons. L'entraînement se fait avec des données fournies et requière un système de "récompense". Lorsque le résultat est correct on en informe l'algorithme en lui disant que ce qu'il a fait est correcte et lorsqu'il se trompe, on lui dit qu'il a mal fait quelque chose.

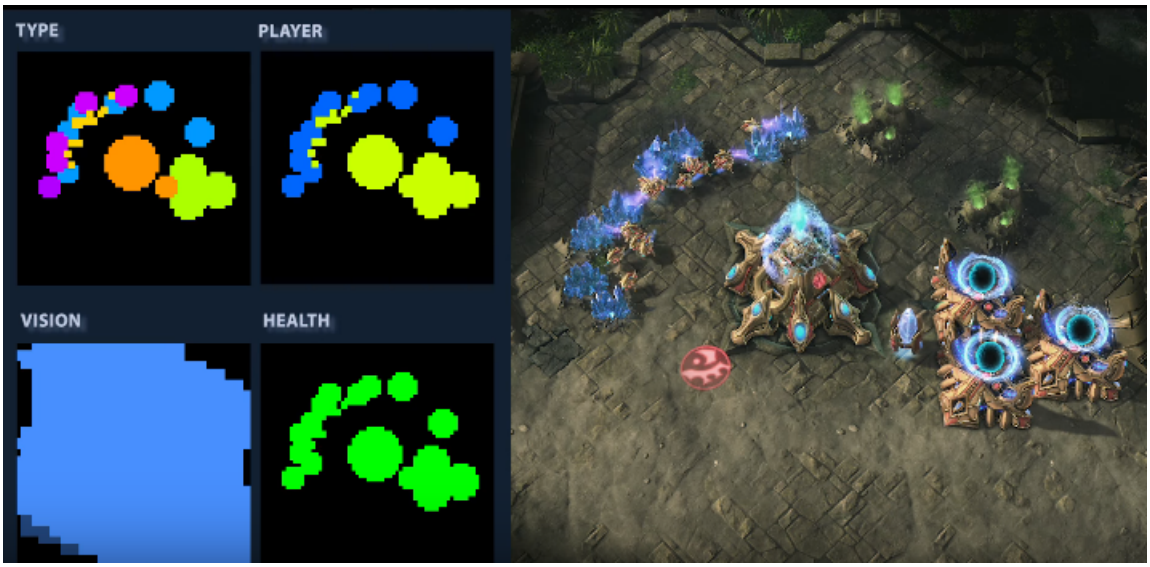


FIGURE 4 – À gauche, ce que voit l'*IA*. À droite, le jeu

4 Analyse fonctionnelle

4.1 Réseau de neurones

4.1.1 Modèle

Il faut commencer par, soit créer un modèle, soit en utiliser un existant. Le modèle définit la façon dont seront organisées nos informations et il impacte également les performances de l'application finale. Lors de ce travail, je vais donc créer un modèle moi-même. N'ayant jamais créée de modèle précédemment, je vais devoir me documenter et apprendre. Il faut savoir qu'il existe plusieurs types/catégories de modèle, les plus connus étant : *Supervised*, *Unsupervised* et *Reinforcement*.

— **Supervised** : Utilisé lorsque l'on sait comment classifier les données et que l'on souhaite juste les faire trier. Entraînées avec des données fournies et requièrent des données de test.

— **Unsupervised** : Utilisé lorsque l'on ne sait pas comment classifier les données. Requière des données de test, permettant au modèle de définir un "pattern" dans les données reçu.

— **Reinforcement** : Utilisé lorsque l'on ne sait pas comment classifier les données, mais que nous savons quel résultat nous souhaitons. Entraîné avec des données fournies et requière un système de "récompense". Lorsque le résultat est correct on en informe l'algorithme en lui disant que ce qu'il a fait est correcte et lorsqu'il se trompe, on lui dit qu'il a mal fait quelque chose.

Le modèle que je vais créer sera de type "Supervised". Ce n'est pas le modèle le plus optimisé pour ce genre de choses, mais utiliser un modèle *NEAT* sans avoir réalisé de réseau de neurones au préalable serait du suicide je pense, bien que je pensais l'utiliser au départ. J'utiliserai également les algorithmes génétiques dont je parlerais plus amplement plus tard. Voici un exemple d'un réseau de neurones pour le jeu du T-Rex (pas forcément celui qui sera utilisé).

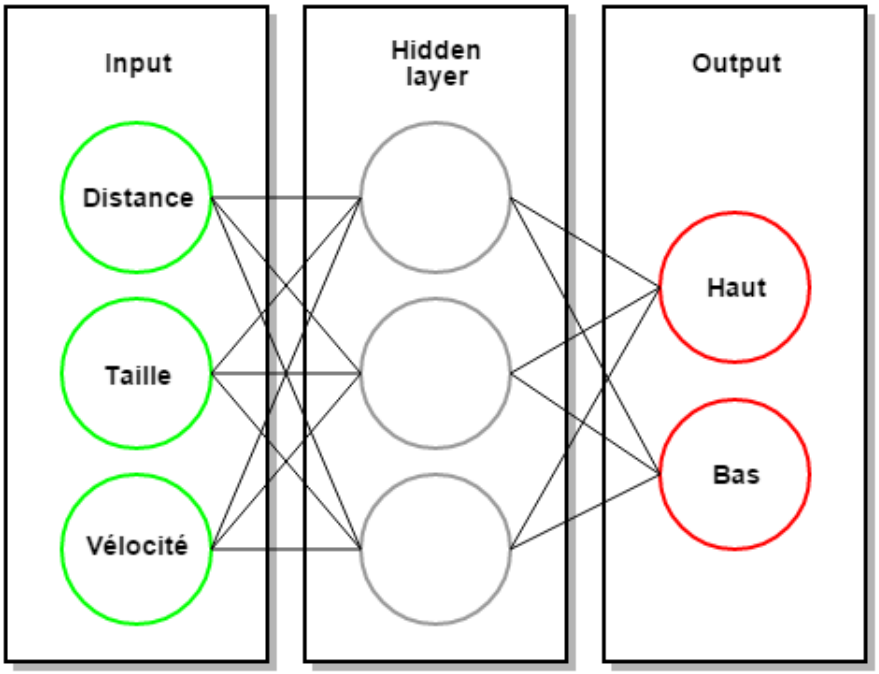


FIGURE 5 – Exemple du réseau de neurones du T-Rex

Un réseau de neurones se divise en trois parties :

- Input
- Hidden layer
- Output

4.1.2 Input

La partie "Input" représente toutes les caractéristiques qui influeront sur les actions du *T-Rex*. Dans ce jeu, plusieurs choses peuvent influencer le déclenchement du saut.

- **La distance :** permettant de connaître la distance séparant le T-Rex d'un obstacle
- **La taille de l'obstacle :** permettant de connaître la taille en largeur de l'obstacle le plus proche
- **La vélocité :** permettant de connaître la vitesse actuelle du T-Rex.
- **La position Y :** permettant de connaître la position sur l'axe Y de l'obstacle le plus proche.

Tous ces éléments sont importants au déclenchement du saut. Si le *T-Rex* ne connaît pas la distance le séparant de l'obstacle, il ne saura jamais quand sauté. De même si le *T-Rex* ne connaît ni sa vélocité ni la taille de l'obstacle, il ne comprendra jamais qu'il lui faut un timing précis pour un obstacle plus long ou bien avec une vitesse plus élevée.

Étant donné que l'*IA* est développé par-dessus le jeu, elle a accès à toutes les valeurs citées précédemment. Il est néanmoins envisageable dans le futur de la créer en dehors de celui-ci et de récupérer ces informations uniquement en analysant l'interface graphique.

4.1.3 Hidden layer

Passons maintenant à la zone du milieu, la partie "Hidden layer". Cette partie ne contient qu'une seule chose, des neurones. Il est important de noter qu'il peut y avoir plusieurs "Hidden layer" mis côte à côte, cela complexifie le réseau de neurones et permet d'obtenir un résultat plus précis, mais cela prendra également plus de temps à l'exécution, il faut donc trouver le bon rapport complexité/temps. Ces neurones sont en fait simplement des fonctions que l'on appelle "fonction d'activation". Ils prennent en entrée deux nombres, qui sont le résultat du neurone précèdent ainsi qu'un poids. Ils transmettent ensuite le résultat aux neurones suivants et ainsi de suite jusqu'à arriver à la zone "output". Les poids sont représentés sur le schéma par des traits, il s'agit en fait simplement de nombres générés aléatoirement se trouvant dans un intervalle défini. chaque trait est un nombre différent.

4.1.4 Fonction d'activation

Pour revenir sur les fonctions d'activations, ce sont des fonctions défini qui permettent de faire toute sorte de chose, les plus connus étant : Sigmoid, ReLu, Tanh et Linear. Il faut également savoir que, dans un réseau de neurones "standard", les seules valeurs qui peuvent varier sont les poids, ce sont les valeurs que le réseau de neurones va modifier jusqu'à obtenir le résultat escompter. Les valeurs que le réseau de neurones ne peut pas apprendre de lui-même sont appelées "Hyperparameter", ce type de paramètre doit être renseigné a la main (exemple : le nombre de neurones).

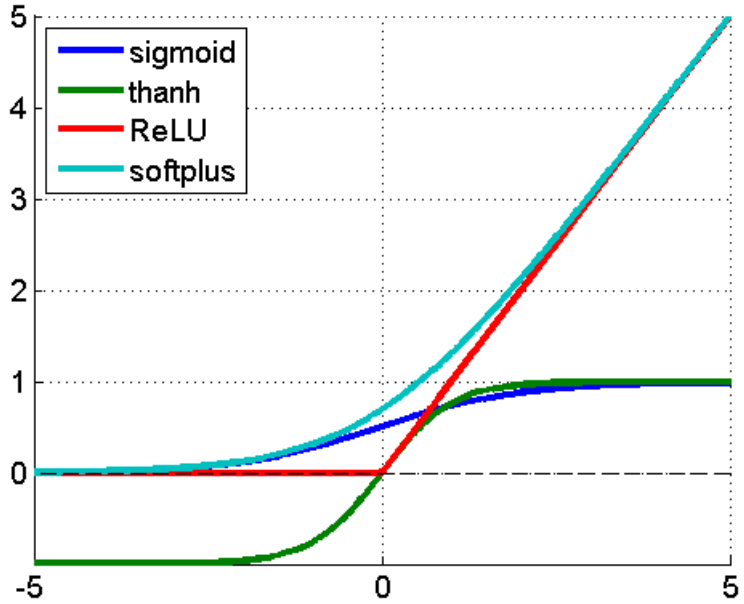


FIGURE 6 – Différente fonction d'activation

4.1.5 Output

Il ne reste maintenant plus que la troisième partie, la zone "Output". Cette zone est, comme son nom l'indique, la zone de sortie. Selon le résultat obtenu sur le neurone de sortie, une touche sera pressée ou non (dans le cas du *T-Rex*). Pour cela, il y aura simplement une fonction qui vérifiera le résultat obtenu, si le résultat est plus grand que 0.6 (par exemple), alors la touche est activée, sinon on ne fait rien. Il faut savoir que le résultat de sortie de chaque neurone est normalisé en général entre 0 et 1 (grâce aux valeurs maximum et minimum du réseau de neurones).

4.1.6 Entraînement

Dans le cas du jeu du *T-Rex* je ne pourrais pas utiliser les méthodes d'entraînement d'un réseau de type "supervised". Dans le jeu du *T-Rex* j'ai des données d'entrées (distance, etc..), mais pour faire du supervised il me faut également les valeurs de sortie, ce que je n'ai pas. Je ne peux pas lui dire "si tu as 10 pour la distance, 2 pour la vélocité, 9 pour la taille et 4 pour la position en Y, alors saute". Le but est qu'il apprenne de lui-même et en direct, pas que je lui dise "lis ces valeurs et trouve un pattern" pour ensuite le lancer sur le jeu pour qu'il aille un gros score.

À cause de cela je suis obligé de faire autrement, je vais donc mettre en place un algorithme génétique qui servira uniquement à l'entraînement de ce réseau.

4.2 Algorithme génétique

Les algorithmes génétiques me permettront de générer plusieurs réseaux de neurones qui sont appelés "Génome". En général, la seule variable d'un réseau de neurones sont ses poids, nous allons donc créer plein de réseau de neurones avec des poids différents dans le but de les faire se "reproduire" pour obtenir des "enfants" plus "forts".

Ces réseaux de neurones sont ensuite testés sur le jeu, puis, en faisant de la sélection naturelle, nous allons jeter ceux ayant le moins bon score. Ensuite, nous allons faire ce que l'on appelle de l'enjambement ("cross over" en anglais). Cela consiste à prendre des parties de chacun des réseaux de neurones restant pour les combiner en un seul. Par la suite, nous allons sélectionner aléatoirement des parties d'un génome auquel seront ajoutées des valeurs aléatoires (étape appelée "mutation"). Il ne reste plus qu'à répéter ce processus jusqu'à avoir le nombre de génome souhaité, créant ainsi une nouvelle génération. Tout ce processus (sélection, enjambement et mutation) est répété jusqu'à avoir le résultat escompté.

4.2.1 Sélection

La sélection est la première étape dans la mise en place d'un algorithme génétique. Elle consiste, comme son nom l'indique, à sélectionner les meilleurs réseau de neurones. Pour cela il va falloir les évaluer. Dans mon cas, la valeur d'évaluation d'un réseau de neurones est le nombre d'obstacles que l'*IA* a réussis à passer. Cette valeur est appelé "fitness". il

existe plusieurs noms différents selon le type de résultat que l'on souhaite avoir. Si nous avons une *IA* qui a pour objectif d'obtenir un score le plus petit possible (réduire un maximum les dégâts / minimiser quelque chose) alors on utilisera le terme "cost". Si, a l'inverse, le but est d'obtenir le plus gros score possible, alors le terme "fitness" sera utilisé. Le principe de la sélection est simple, selon le nombre de génomes par génération, le nombre de parents sélectionnés change. Plus il y a de génomes, plus il y aura de parents sélectionnés. Cela permet d'augmenter la diversité et ainsi d'explorer plus de solutions. Une fois la sélection terminée, on envoie tous les parents pour qu'ils se reproduisent.

4.2.2 Enjambement

Cette méthode consiste à créer des enfants basés sur deux parents qui ont été sélectionnés. Les enfants sont créés en mélangeant les gènes de chacun des deux parents. Son but est de produire des enfants différents les uns des autres pour ainsi explorer plus de possibilités.

Il existe plusieurs méthodes d'enjambement, je parlerais ici de la méthode en un point (one point crossover) qui est une des plus simples à comprendre et à expliquer. Pour cela, on commence par tirer un point aléatoirement dans un des deux parents et on divise en deux les deux parents à partir de ce point. Il ne reste plus qu'à prendre la 1er partie du premier parent et de la mettre avec la 2eme partie du deuxième parent ainsi que la 1er partie du deuxième parent avec la 2eme partie du premier parent. Grâce à ça, on obtient deux enfants qui sont différent de leurs parents. Les autres méthodes consistent simplement à prendre plus de point et diviser les parents en plus de partie.

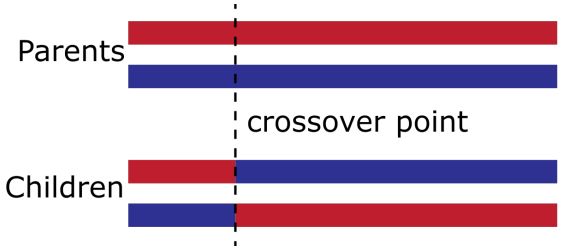


FIGURE 7 – méthode d'enjambement "crossover" en un point

4.2.3 Mutation

La dernière étape d'un algorithme génétique consiste à faire muter certaines parties des gènes des enfants. Son but est d'empêcher d'être bloqué dans des optima locaux. C'est une étape très importante puisque si deux parents sont identiques, les enfants créés le seront également. La mutation permet donc de sortir de cette boucle infinie.

Si on prend une courbe qui a plusieurs piques et que notre but est de trouver quel est le point culminant, en utilisant des algorithmes génétiques ou d'autres méthodes, il est possible que l'algorithme trouve un point qu'il considère comme étant le plus élevé et va donc s'arrêter de chercher alors qu'il existe un point plus loin qui est plus élevé, c'est ce que l'on appelle un optimum local. Le but de la mutation va donc être de dire à chaque enfant qu'il faut regarder plus loin en les modifiant légèrement pour peut-être trouver un optima global.

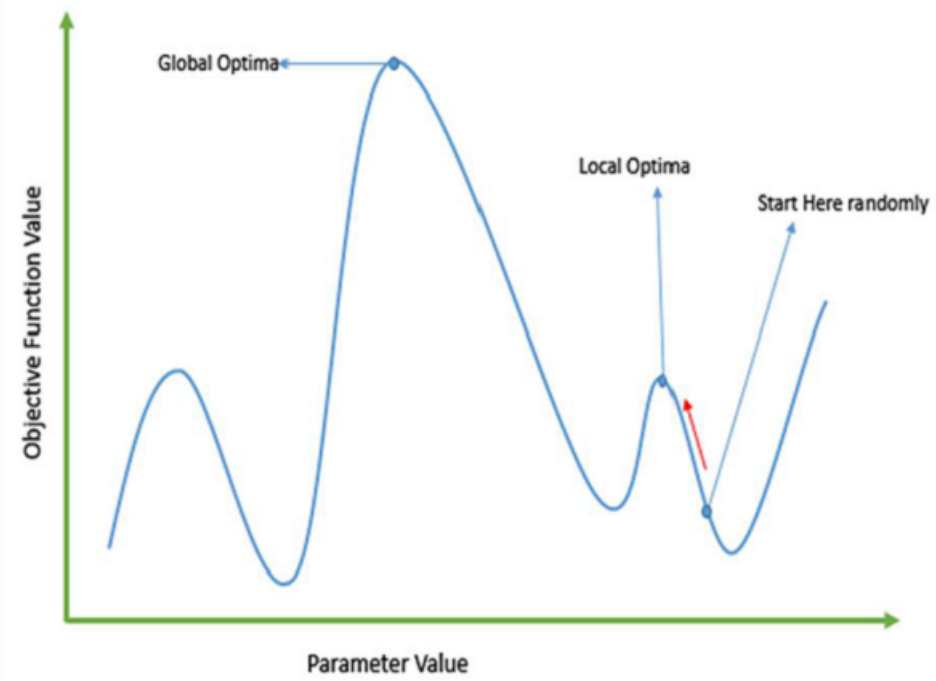


FIGURE 8 – exemple d'optimum global et d'optimum local

Il existe plusieurs méthodes de mutation et l'utilisation de chacune dépend du besoin ou des valeurs dans les gènes des parents (binaire, nombre, valeur dans une portée donnée, etc..). Personnellement j'ai choisi de faire de la mutation avec un taux. Les gènes de chaque enfant sont parcouru et chaque partie des gènes à X% de chance d'être modifié. Pour ce qui est de la modification des gènes, une valeur entre -1 et 1 est tirée aléatoirement et ajoutée à l'ancienne valeur du gène.

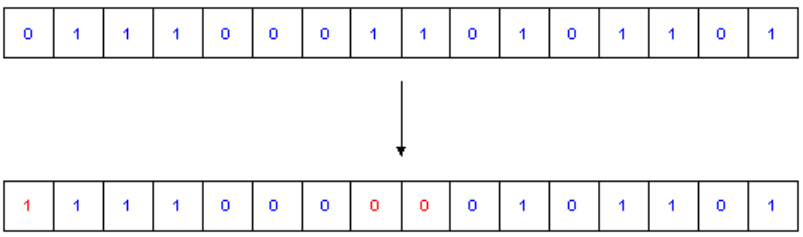


FIGURE 9 – méthode de mutation

4.3 Maquette de l'interface

Cette application contiendra une seule page. Sur celle-ci seront affichées :

- le jeu
- une partie "pressed key" qui contient les touches que l'*IA* peut presser et qui seront illuminées lorsqu'elles le sont.
- une partie "informations" qui contiendra la distance entre l'obstacle le plus proche et le dinosaure, la taille de l'obstacle, la vélocité du dinosaure ainsi que l'index de la génération actuelle.

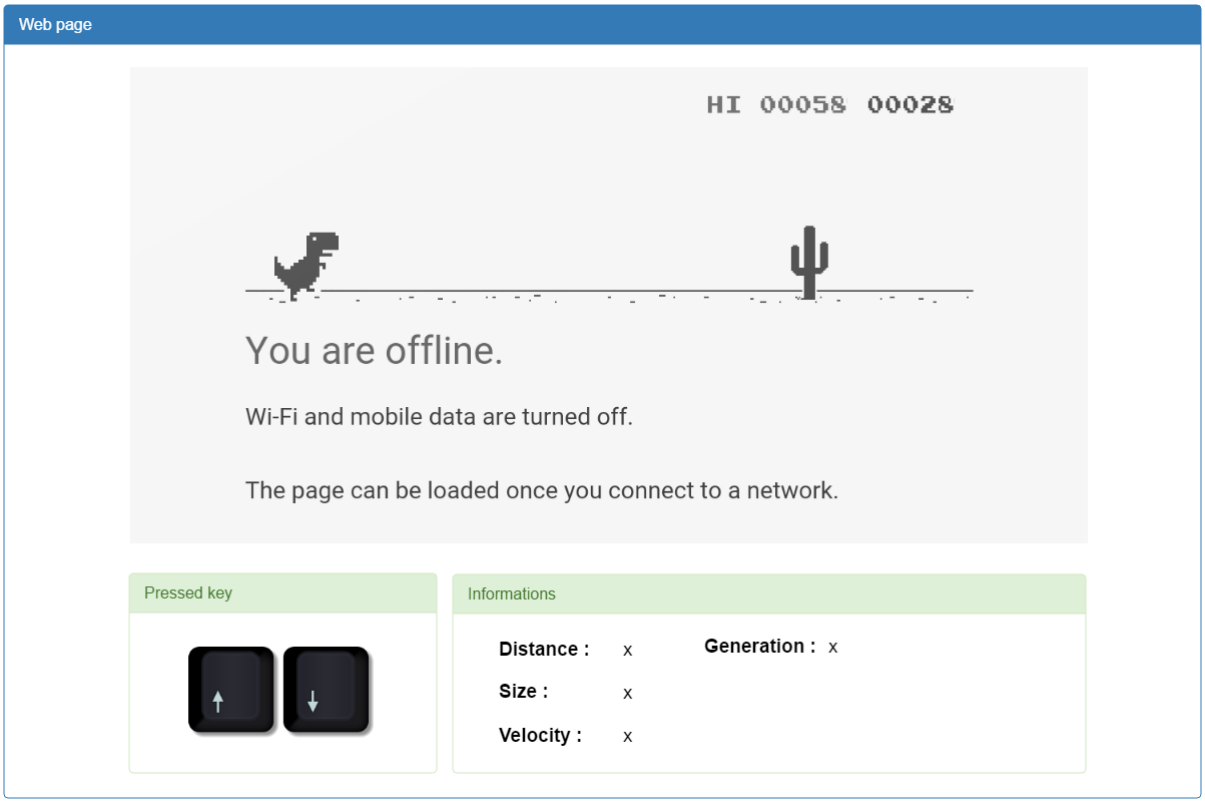


FIGURE 10 – Interface de l'application

4.4 Carte de navigation du site

Voici la carte de navigation du site. Comme montré ci-dessous, chaque fenêtre n'est accessible que depuis le "home" (page principale).

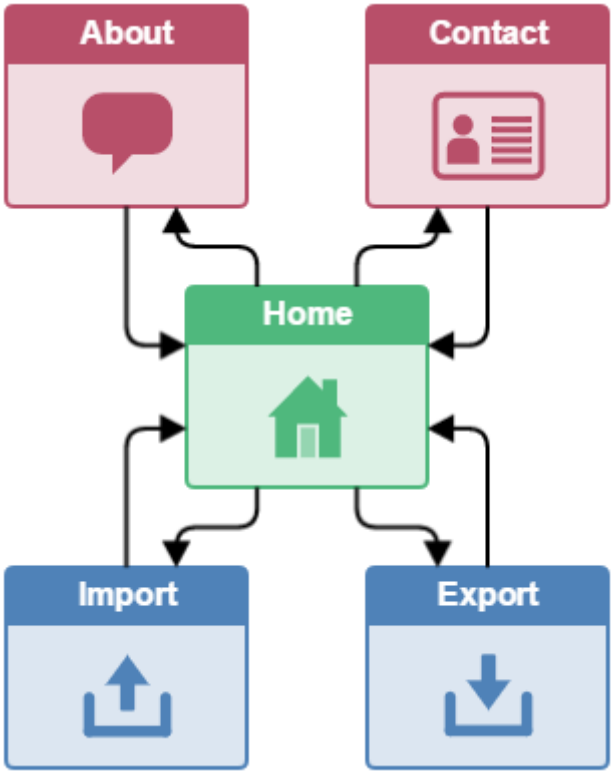


FIGURE 11 – Carte de navigation

5 Analyse organique

5.1 Diagramme de classe

5.2 Generation

5.2.1 constructor

5.2.2 run

5.2.3 nextGen

5.2.4 selection

5.2.5 crossover

5.2.6 singlePointCrossover

5.2.7 twoPointCrossover

5.2.8 uniformCrossover

5.2.9 mutation

5.2.10 mutationWithoutRate

5.2.11 mutationWithRate

5.3 Genome

5.3.1 constructor

5.3.2 feedForward

5.3.3 getOutput

5.4 NeuralNetwork

5.4.1 constructor

5.4.2 feedForward

5.4.3 getWeights

5.4.4 setWeights

5.4.5 backPropagation

5.4.6 getResults

5.4.7 getOutput

5.5 Layer

5.5.1 constructor

5.5.2 getWeights

5.5.3 setWeights

5.6 Neuron

Classe permettant d'instancier des neurones.

5.6.1 constructor

5.6.2 getWeights

5.6.3 setWeights

5.6.4 feedForward

5.6.5 calculateOutputGradients

5.6.6 calculateHiddenGradients

5.6.7 sumDow

5.6.8 updateInputWeight

5.7 Connection

Classe permettant de gérer les poids pour chaque connexion de chaque neurone.

5.7.1 constructor

Lance la méthode "randomWeight" pour ainsi créer un poids.

5.7.2 randomWeight

Permets de générer un nombre aléatoire entre 2 valeurs données en entrée. Les deux valeurs sont incluses.

5.8 ActivationFunction

Classe abstraite permettant d'avoir un modèle pour les différentes fonctions d'activation. Elle contient un constructor qui empêche son instanciation ainsi que deux méthodes. La méthode "normal" qui est simplement la fonction et la méthode "derivative" qui est la fonction dérivée. Le but est qu'en changeant la fonction d'activation lors de la création de l'objet "Generation", tout ce fasse automatiquement.

5.9 sigmoid

La classe "sigmoid" est une classe permettant d'utiliser la méthode d'activation du même nom.

5.9.1 normal

Elle prend en paramètre un nombre et permet d'utiliser la fonction "normalement". Dans ce cas là, voila le calcule effectué : $\frac{1}{1+E^{-x}}$, x représentant le paramètre de la méthode et E étant une constante mathématique valant environ 2.71828.

5.9.2 derivative

Elle prend en paramètre un nombre et permet d'utiliser la fonction dérivée. Dans ce cas là, voila le calcule effectué : $E^{\frac{-x}{(1+E^{-x})^2}}$, x représentant le paramètre de la méthode et E étant une constante mathématique valant environ 2.71828.

5.10 tanh

La classe "tanh" est une classe permettant d'utiliser la méthode d'activation du même nom.

5.10.1 normal

Elle prend en paramètre un nombre et permet d'utiliser la fonction "normalement". Dans ce cas là, il existe une méthode directement implémenté dans *Javascript* qui se nomme "tanh" et s'occupe de faire la fonction (tanh $\left(\frac{E^x-E^{-x}}{E^x+E^{-x}}\right)$), x représentant le paramètre de la méthode et E étant une constante mathématique valant environ 2.71828.

5.10.2 derivative

Elle prend en paramètre un nombre et permet d'utiliser la fonction dérivée. Dans ce cas là, voila le calcule effectué : $1 - x^2$, x représentant le paramètre de la méthode.

6 Planification prévisionnelle

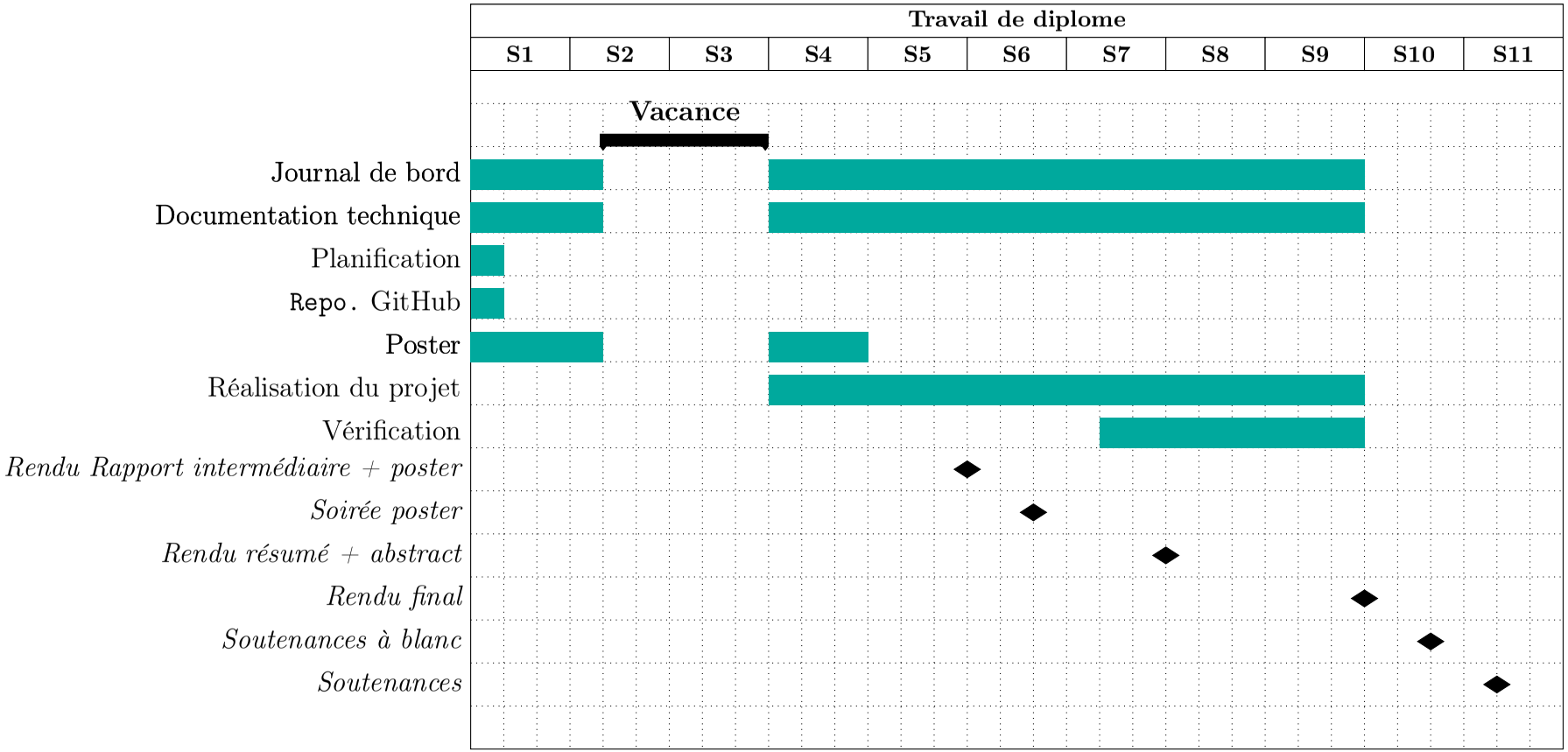


FIGURE 12 – Diagramme de Gantt

7 Tests

8 Conclusion

9 Dictionnaire

10 Références

[1] Wikipedia. *Intelligence artificielle.*
URL : https://fr.wikipedia.org/wiki/Intelligence_artificielle.

[2] Xviniette. *Flappy learning.*
URL : https://github.com/xviniette/FlappyLearning.

[3] Oriol Vinyals. *Starcraft IA.*
URL : https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment/.

[4] SethBling. *MarI/O.*
URL : https://www.youtube.com/watch?v=qv6UVOQ0F44.

[5] Mobilegeeks. *Forza Drivatar.*
URL : https://www.youtube.com/watch?v=twI0RSVwnR0.

[6] Xbox Wire Staff. *Forza Drivatar.*
URL : https://news.xbox.com/2014/09/30/games-forza-horizon-2-drivatars/.

[7] Kyle Orland. *Forza Drivatar.*
URL : https://arstechnica.com/gaming/2013/10/how-forza-5-and-the-xbox-one-use-the-cloud-to-drive-machine-learning-ai/.

[8] TASVideos. *BizHawk.*
URL : https://github.com/TASVideos/BizHawk.

[9] Wikipedia. *NEAT.*
URL : https://en.wikipedia.org/wiki/Neuroevolution_of_augmenting_topologies.

[10] Wikipedia. *Définition glitch.*
URL : https://fr.wikipedia.org/wiki/Glitch.

[11] Wikipedia. *Cloud computing.*
URL : https://fr.wikipedia.org/wiki/Cloud_computing.

[12] Wikipedia. *Alpha Go.*
URL : https://fr.wikipedia.org/wiki/AlphaGo.

[13] Wikipedia. *Reinforcement Learning.*
URL : https://fr.wikipedia.org/wiki/Apprentissage_par_renforcement.

[14] Burak Kanber. *Machine Learning : Introduction to Genetic Algorithms.*
URL : https://www.burakkanber.com/blog/machine-learning-genetic-algorithms-part-1-javascript/.

11 Table des figures