ICS3209 - Advanced Game AI

Sugar Rush - Group Assignment

Valerija Holomjova, Francesca Silvio, Deborah Vella

ABSTRACT

This report describes the game produced as part of the Assignment for ICS3209 - Advanced Game AI. This assignment comprised of creating a game which includes a reenforcement learning component. This document will describe the implementation process followed to create the deliverables along with an analysis on the results obtained and a discussion on possible future improvements.

1. INTRODUCTION

The goal of our project is to create a game which employs reinforcement learning techniques and to experiment with and improve the performance of the implemented agents. The game created takes the form of a simple platformer runner game. In the runner game, the player will be playing individually and collecting points and dodging obstacles. The runner game includes two levels, one more challenging than the other through the manipulation of platform and obstacle positioning. In each level, the player can reach a checkpoint which triggers a mini game which the player has to win in order to proceed to the next level of the game. In the mini games, the user will be playing against a pretrained AI. There are a total of two mini games that the player can play; the karts racing game and the table tennis game.

An extra agent was introduced to play the runner game and compete against the AI that were created for the mini games. The Unity[1] game engine (version 2018.4.15) was used to build the main components of the game and the agents were created using Unity's Machine Learning Agents (ML-Agents) Toolkit[2] [1]. The main goals of our research are of the following:

- Creating a game that has a fun concept and smooth gameplay.
- Adding reinforcement learning features and experimenting with training to try improve the overall performance of our agents.
- Setting up ML-Agents and using it's features to create and train our own agent from scratch.

Reinforcement learning is a general framework of Machine Learning (ML) which consists of goal-oriented algorithms which are aimed to carry out suitable actions in a particular situation in order to maximize rewards. In contrast to supervised learning, a reinforcement agent learns from its experience rather then from an annotated dataset. Section 2 will discuss and compare various deep learning algorithms. This section will proceed by presenting the general design, mechanics and UI that was implemented for the game.

[1]https://unity.com/
[2]https://github.com/Unity-Technologies/ml-agents

1.1 Main Concept and Design

The main play style of the game is a platform runner game. Two different mini games were added to the game to create a more varied gameplay and challenge the target user's gaming ability. Additionally, elements in both mini games were altered to match the main theme of the game. A 'Candy' theme was chosen as a main theme for the game due to the vibrancy and contrast of the game elements, which could potentially increase the agents performance. It should be noted that the following assets were used for the game[3][4]. It should be mentioned that we initially tried to draw and create our own game art for the game as depicted in the image below, but decided that a more vectorized art style would yield better performance when training.

Figure 1: This image shows the assets that were initially created for the game.

Figure 2: This figure shows the assets which were used in the runner game.

[3]https://assetstore.unity.com/packages/2d/gui/icons/easter-gui-142479
[4]https://kenney.nl/assets/platformer-art-candy

1.2 Player Mechanics

The following section summarizes the player mechanics implemented for each of the game:

- In the runner game, the player is expected to reach the final checkpoint of the level by jumping on platforms, collecting points and dodging obstacles. The game automatically moves the player towards the right direction at a constant speed. In the meantime, the user has to press the space bar key when they want to jump. The distance the player moves and the amount of points they pick up determine the score of the game scene.
- In the table tennis game, the user is meant to press the "W, A and D" keys to manipulate the movement of their racket. The goal is to push the ball spawned at each round, using the racket, into the agents side until one of the players reaches a total of five points.
- In the kart racing game, the user can use the arrows on their keyboard to move their go-kart in the direction they desire. The goal of this game is to compete against the AI and reach the finish line first.

1.3 The Physical Game World

1.3.1 Runner Scene

The main environment consists of the background, platforms, points and obstacles. The background is given a parallax effect so that it would move to the appropriate position once the player gets out of its scope for a continuous effect. There are a total of 4 different platform sizes; small, medium, large and very large. The colliders for each platform were manually set and each platform was then manually placed into the game scene.

The points are in the form of cherries and positively influence the score of the player and reward of the trained agent of the game. At the end of each level, a checkpoint is used to transition the player from the runner scene to a mini game and resembles an easter egg. There are three types of obstacles in the game; the ice cream, the lollipop and the jelly. Each obstacle has a varying height and width, and is used to intensify the difficulty of the game. It should be noted that all the game entities were placed into the scene manually.

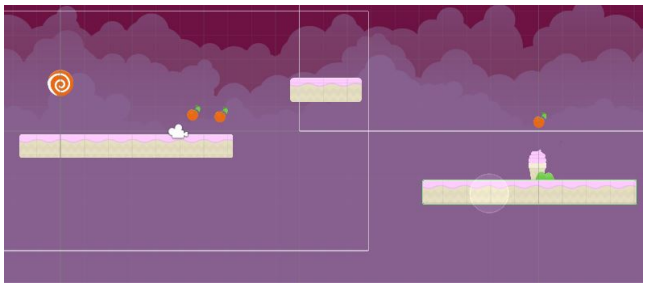

Figure 3: This image shows the Runner Scene being described in Section 1.3.1.

1.3.2 Table Tennis Scene

This scene is an adaptation of one of Unity's example scenes from the ML-agents library. The scene contains 2 rackets and a table tennis court. The colours of the scene and font of the score text were changed in order to match the theme of the game. Some changes were also made to the logic of the game. The original unity game awarded points to the player/agent if they throw the ball out of the court. This made it very easy for the agent to learn and the game would end up being very boring and easy. In order to increase the challenge, the game was changed to follow the rules of a normal table-tennis game - if a player throws the ball out of the court, the other player gets awarded a point. Due to the fact that the court size is quite small, this factor makes the game very difficult, both for agents and also human players. The original game also keeps on going indefinitely, until the game is actually stopped. The mini-game adaptation will stop after one of the players scores five points, making this player the winner of the mini-game.

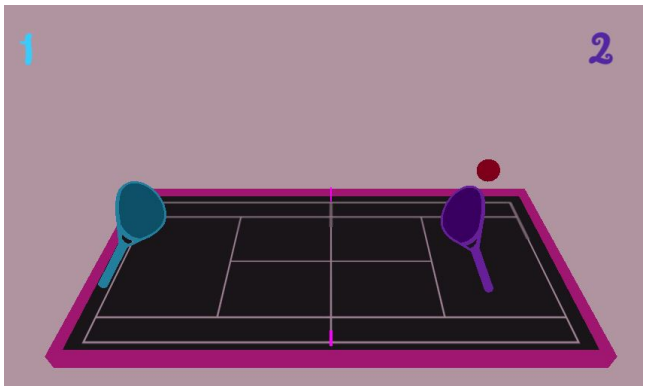

Figure 4: This figure shows the Table Tennis Scene being described in Section 1.3.2.

1.3.3 Kart Racing Scene

This mini-game is another adaptation of a game obtained from Unity [5]. This mini-game took more work than the table tennis mini-game as it is not a full example, but just a demo of a trained model. First, a player scene was created, which included a kart which could be controlled by a player and one which could be controlled by the AI agent. Next, some scripts had to be changed in order to train the AI agent. As previously mentioned, this was just a demo project and the only information found was on the demonstration. Different versions of ml-agents are not compatible with each other and no information on the project's ml-agents version could be found. Due to this, some ml-agents scripts needed to be changed in order for the game to work with the version we were using.

Some other changes made include making the game finish after one lap, with the first person crossing the finish line being the winner and not restarting the game whenever the player hit a wall.

1.4 UI Elements and Sound Effects

1.4.1 General

The User Interface elements were chosen to compliment the "Candy" theme, therefore, containing bright coloured buttons, panels covered in icing sugar and different backgrounds which play an important role in setting the ambience. An essential method to set the atmosphere of the game is by integrating sounds and background music within the environment. Once again, the sounds chosen mirror the

[5] https://ole.unity.com/mlkartproj

Figure 5: This figure shows the Kart Racing Scene being described in Section 1.3.3.

Figure 6: A screen capture of the main menu.

"Candy" theme by setting a joyful atmosphere, with each sound effect complimenting one particular type of action, such as jumping, collecting points, clicking buttons, completing or failing a level and more.

1.4.2 Main Menu

The player is initially presented with a main menu screen, which provides three different choices:

- **Start with A.I**: When clicking this, the entire game including the mini-games are played by a well trained agent for each different level. In the mini-games, this agent plays against another agent which is not as well trained as the player agent. Therefore, when one chooses this option, the user does not have to intervene during the game.
- **Start without A.I**: This option allows the user to play the entire game himself. Every time a mini-game is reached, the player has to play against a pre-trained agent playing the role of the opponent.
- **Tutorial**: This should be chosen if the player does not have any knowledge of how the game works, so that he is guided through the mechanics of the game.

After choosing if the game should be played by the A.I. or the user, the scene transitions to another choice, this time providing the difficulty options of the game. This choice was not implemented as problems with changing the learning academy at run-time were encountered. The two choices are either easy or hard, then according to the difficulty chosen, the respective agent learning brains would be used. Hence, if the easy option is chosen, the opponent would make use of the least trained brain, while if the hard option is chosen, the opponent would make use of the more trained brain. It should be noted that, when the player is the A.I. itself, it makes use of the mostly trained brain, to be able to defeat the opponents and complete the entire game.

1.4.3 Tutorial

The tutorial component of a game is an essential part as it walks the user through the mechanics of the game, its goal, and what to avoid or collect. This way the player gains any knowledge required to eventually play the game.The tutorials also conform with the main theme, displaying vibrant hues but making sure that each scene is still easy on the eye, by using fonts and sizes which prevent any eye straining, and using colours which compliment each other. Furthermore, the tutorials provide both visual and textual information, so that the user is given a clearer picture. The tutorials are straight to the point, as they do not contain any extra unnecessary information and neither any lack of information. Five different tutorial scenes were created in all, allowing the user to flow through them. Three of them explain the runner game, which are:

- **Player Controls**: Explains the different jumps one can use, and how to use them.
- **Obstacles**: Explains and shows which game elements should be avoided to complete the game
- **Checkpoints**: Shows what a checkpoint looks like and what happens when it is reached.

The other two are one for each of the two mini-games. These are displayed after the completion of a particular level, and before the mini-game starts. This way the user does not have to remember the controls from before the entire game starts, but are given only when needed.

- **Kart game tutorial**: Explains which buttons from the keyboard are used and what each one does. It also, explains the goal of the game and what happens when completed.
- **Table tennis tutorial**: This also explains the controls that should be used for playing and the target of the game.

1.4.4 Runner Scene

The following scene includes UI to notify the player of the current reward they have obtained, which is displayed in the top left corner of the scene. In other words it represents the current score obtained by the player or agent. The reward is updated positively if the player performs positive actions such as collecting points or jumping on different platforms. Alternatively, it updates negatively for negative actions such as falling into obstacles. For more information on how the rewards were optimized please refer to Section 4.1.

1.4.5 Table Tennis Scene

The UI elements in this scene are the scores of each player. These are updated during the game, starting both from zero,

Figure 7: This figure shows the first tutorial screen for the runner game.

Figure 8: This figure shows the obstacle tutorial screen for the runner game.

Figure 9: This figure shows the table tennis tutorial screen which will be displayed before playing the mini-game.

and adding one to the score of the player who wins a round. The first player that reaches five, wins the mini-game. If the player reaches five first, then the user is presented with a level completed screen. On the other hand, when the opponent wins, the user is presented with a level failed screen.

Figure 10: This figure shows the kart racing tutorial screen which will be displayed before playing the mini-game.

Figure 11: This figure shows the screen displayed when the player loses the tennis mini-game.

1.4.6 Kart Racing Scene

This scene starts with a small countdown, so that the game is played fairly having the karts leave at the same time preventing any initial advantage one over the other. Just like the table tennis scene, the user is notified with a screen whether he won the level or lost the level.

2. BACKGROUND

Reinforcement learning (RL) describes the problem of an agent which must learn how to complete a task in a dynamic environment through trial-and-error interactions [2]. This section will discuss the history of developments in RL and also describes different RL models used to train intelligent agents in games.

2.1 History of Reinforcement Learning

The roots of reinforcement learning are purely psychological. The concept of reinforcement learning in psychology dates back to 1911, where studies on learning by trial and error and the psychology of animal learning were conducted

Figure 12: This figure shows the screen displayed when the player wins the kart racing mini-game, hence successfully completing the game.

[3]. In [4], Thorndike states a 'Law of Effect', which discusses the effect which reinforcement has on the tendency to select actions. This law describes trial and error learning by splitting it into two parts. The first part involve selectional learning, where a choice is made by comparing the consequences of different options. The second part pertains to associative learning, where selections are made based on association to situations. In 1954, this law was used as a basis for the computational implementation of trial and error learning [5].

Another concept which played an important role in the history history of RL is the problem of optimal control. This problem describes the issue of designing a controller for a dynamical system which optimizes the objective function over a period of time. This problem is solved by dynamic programming methods which started being developed in the 1950s [6].

In the 60s, different reinforcement learning methods started being applied to simple games such as tic-tac-toe, henceforth paving the way for greater developments. With progress in the fields of computer science, artificial intelligence, statistics and psychology, reinforcement learning systems continued evolving throughout the years, leading to the creation of the reinforcement learning techniques used today. Some of these technique are described in the following subsections.

2.2 Reinforcement Learning Concepts

This subsection discusses basic concepts which are used in the RL techniques, which will be described in the following sections.

Standard RL models include an agent connected to an environment via perception and action. The agent receives an input and an indication of the environment's current state in order to choose an action to output. This action changes the environment's state and a reinforcement signal is in turn sent back to the agent. The agent should aim to pick actions which will return a higher return signal. This is done by using reinforcement learning algorithms and systematic trial and error. A reward function describes the objective feedback from the environment in order to create the reinforcement signal. These rewards are usually scalar variables associated to some states or state-action pairs. The transition function describes the probability of going from your current state to a particular state with a particular action.

Figure 13 depicts the flow between these different concepts in a diagram [2] [3].

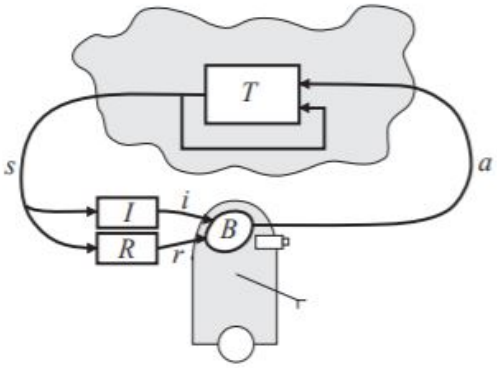

Figure 13: Standard RL model, where i is the input, s is the current state, a is the output action, r is the reinforcement signal, B is the agent's behaviour, I is the input function, T is the transition function and R is the reward function [3].

2.3 Markov Decision Process

The Markov Decision Process (MDP) is a discrete stochastic version of the optimal control problem which handles problems with delayed reinforcement. The MDP consists of a set of states S, a set of actions A, a reward function R, which represents the instantaneous reward as a function of the current state and action and a state transition function which gives the probability of the next state as a function of the current state and action. A Markov state follows the Markov property ($p(s', r|s, a) = Pr(R_{t+1} = R, S_{t+1} = s'|S_t, A_t)$) in order to summarize the relevant past information compactly [3] [2].

2.4 Monte Carlo

Monte Carlo methods discard the assumption of having complete knowledge of the world but learn only from experience, consisting of actions, rewards and sample sequences of states. The algorithm goes through a number of episodes, assuming that no matter what actions are taken, each episode terminates at one point or another. At the end of every episode, the states, actions and rewards are acquired, to eventually compute the average of the V(s) and Q(s). Monte Carlo deals with multiple number of states, and each time an action is performed on one particular state, its corresponding return depends on the actions which will be performed in later states throughout the episode at that point in time. The value functions are learnt from the Markov Decision Process returns. This addresses the problem of nonstationarity, which stems out from the fact that every action undergoes learning [7].

2.5 Q-Learning

Q-Learning was initially proposed by Watkins [8] in 1989. Q-leaning is a type of model-free based learning, which essentially means that the agent maximises the policy based only the possible actions it can perform and does not attempt to model the environment, thus, it is able to generalise

to various environments. The basic process of Q-Learning is the following: At each state the agent finds itself in, it computes the immediate reward or penalty each action has together with the value of the state the action transitions to. As a result, if all possible actions in all the states are evaluated frequently, it will eventually learn which are the best actions in the long-term [9]. Furthermore, Q-Learning is an off-policy algorithm meaning that it computes the optimal action-value function, q_*, independent of the policy used. However, the policy is still a requirement as it determines which state-action pairs are visited and updated, but all pairs need to be updated to eventually result in convergence. It has been proven that the action-value function, Q, can converge with probability 1 to q_*. The function of one-step Q-learning is as follows [7]:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Where, $Q(S_t, A_t)$ is the action A in state S at time t, R_{t+1} is the reward, S_{t+1} represents the next state, α is the learning rate and γ is the discount factor which lies within the range of [0,1).

Q-Learning gave rise to various other reinforcement learning algorithms such as Delayed Q-learning, Fitted Q-iteration and Phased Q-learning. These algorithms are very similar to the original Q-learning algorithm, but focus more on attempting to increase the convergence rate given that Q-learning is a rather slow algorithm [10].

2.6 Deep Q-Networks

Deep Q-network was firstly introduced by Mnih et al. [11]. A deep Q-network (DQN) is able to learn policies from high-dimensional inputs by using reinforcement learning. In fact, the concept of DQN is to combine deep neural networks with a reinforcement technique. DQN specifically makes use of deep convolutional neural networks (CNN) which implement convolutional filters that given an image, go through an entire image for the network to eventually start recognizing or detecting parts of the image, as seen in Figure 14. Just like in Q-learning, DQN cater for tasks where the agent has to interact with its environment through actions and rewards, the objective being to maximise the cumulative future reward. The main function of DQN comprises of a CNN which estimates the optimal action-value function as follows:

$$Q^*(s, a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a,]$$

where r_t is the maximum sum of rewards, discounted by γ at time t. Having policy $\pi = P(a|s)$, which follows an observation s, and an action a.

Furthermore, with DQN the value function is parameterized by $Q(s, a; \theta_i)$, where θ_i are the weights of the Q-network at iteration i. An essential part of DQN is inspired by biological mechanism, which they called experience replay, whose role is to randomize over data and store the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ at every time-step t in training data $D_t = \{e_1, ..., e_t\}$. Just like any Neural Network, DQN needs its own loss function in this case to apply Q-learning updates over mini-batches:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \cup(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2]$$

The algorithm of deep Q-learning is outlined in Figure 15, which make use of components discussed in this section, mainly the experience replay. DQN can obtain very good results but is still not sufficient enough in certain scenarios.

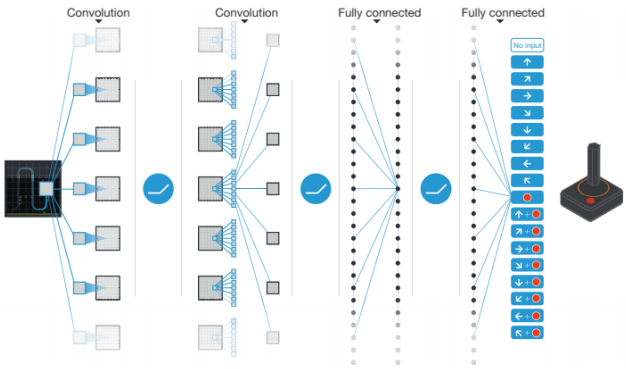

Figure 14: This diagram represents the convolutional neural network component of deep Q-network. It is made up of two convolutional layers and two fully connected layers with a single output for each valid action. [11]

An agent using deep Q-network can only remember the previous four states which provide visual information of each state and bases the next action decision on them. As a result, DQN will not perform as well in games which require a longer memory of previous states. Moreover, DQN can be computationally expensive during training, as it can take up to twelve or fourteen days to completely train the network [12].

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 15: This is the algorithm of deep Q-learning taken from the original paper [11].

2.7 Proximal Policy Optimization

Proximal Policy Optimization (PPO) was introduced by the OpenAI team in 2017[13] and is one of the deep learning algorithms used by the Unity ML-Agents toolkit. This was proposed as an improvement on standard policy gradient methods, which function by taking the stochastic gradient ascent algorithm on the estimator of the policy gradient, which is obtained by differentiating the policy loss (objective function)[13]. As a result, the agent will be encouraged to choose actions that yield high rewards and avoid bad actions. Consequently, if the step size of the gradient ascent was too high, there would be too much variability in

the training. Proximal Policy Optimization (PPO) solves this issue by introducing a new objective function called the "Clipped Surrogate Objective". This objective function of PPO is also based on the methodology behind trust region policy optimization (TRPO)[14], which maximizes an objective function (the "surrogate" objective) to a constraint based on the size of the policy update [13]. The Clipped Surrogate objective improves stability by limiting the policy update at each training step. The formula below shows the main proposed objective.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(rt(\theta)\hat{A}_t, clip(rt(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

In the function above, ϵ is a hyperparameter that helps define the clip range, and is set to a value of 0.2 in the paper[13]. $\hat{A}_t$ represent the estimator of the advantage function at timestep t and $r(\theta)$ is the ratio between the new policy and old policy. The function has two probability ratios; one clipped between the range of [1 - ϵ, 1 + ϵ] (the second term in the min function) and one non-clipped (the first term in min function), and takes the minimum of both objectives. Figure 16 depicts the algorithm of PPO with the clipped objective function.

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}) \right] \right]$$

end for

Figure 16: The PPO algorithm with clipped objective[7].

It should be noted that [13] mentions another alternative to the clipped surrogate objective using adaptive KL penalty which consists of changing the constraint to a penalty in the objective function. However, Schulman et al. mentions that this alternative performs worst when compared to the Clipped Surrogate Objective method. The Clipped Surrogate Objective method can be further improved by adding an entropy bonus to ensure sufficient exploration, to obtain the following objective shown in in the equation below which represents the final Clipped Surrogate Objective Loss function[13].

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

In the equation above, the first term represents the Clipped Surrogate Objective equation mentioned previously, and c_1 and c_2 are coefficients. L_t^{VF} is a squared-error loss represented by $(V_\theta(s_t) - V_t^{targ})^2$).

2.8 Soft Actor-Critic

Just like Q-learning, Soft Actor-Critic is a model-free learning method. It is a stable deep reinforcement learning algorithm used in large continuous domains, which manipulates the standard maximum reward RL target with an entropy maximization term [15]. Soft Actor-Critic alternates between optimizing the Q-function and the policy by applying gradient descent, with the aim of deriving an approximation to soft policy. Soft policy is an algorithm which is used to learn optimal maximum entropy policy by alternating between policy evaluation and policy improvement. SAC has the state value function $V_\psi(s_t)$, the soft Q-function $Q_\theta(s_t, a_t)$ and the policy $\pi_\phi(a_t|s_t)$. Each of these network has its own parameters which are ψ, θ and ϕ respectively. This means that the value function can be modeled by neural networks and the policy can be modeled by Gaussian.

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

Figure 17: This is the algorithm of Soft Actor-Critic[16]

Figure 17 illustrates the algorithm that the SAC algorithm follows. It makes use of two parameterized Q-functions which are trained individually to result in an optimization of $J_Q(\theta_i)$:

$$J_Q(\theta) = \mathbb{E}_{s_t, at \sim D}[\frac{1}{2}(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t))^2]$$

The minimum of the Q-function values is then used to compute the value gradient, $\hat{\nabla}_\psi Jv(\psi)$ and the policy gradient $\hat{\nabla}_\phi J_\pi(\phi)$. It was found that when using two Q-functions, the training time was significantly decreased. After their experiments, the results showed that SAC outperformed all previous state-of-the-art model-free deep Reinforcement Learning models, one of them being the PPO algorithm. The results also imply that robustness and stability can be improved by entropy maximisng RL models [16].

3. METHODOLOGY

3.1 Unity and ML-Agents

In [1], Juliani et al. describe four key dimensions of complexities in AI systems. These are sensory, physical, cognitive and social complexities. Between the Unity platform's two components (its game engine and editor), these complexities are all catered for, thus making it the perfect platform to develop AI systems. The ML-agents tool-kit allows developers to create learning environments using the Unity Editor and interact with them using a Python API. The tool-kit contains a number of functions which aid developers in creating their own agents and environments, some example environments, such as the Tennis example used for one of the mini-games and also reinforcement learning training algorithms which can be used with the Unity Platform. These algorithms include Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC), both described in the previous section.

3.2 Training Agents

To train our agents using the ML-Agents library, an Anaconda environment had to be created and setup as depicted in the following link[8]. It should be noted that ML-Agents Version 0.10.0[9] was used to train the agents for each of the games. Additionally, we tried to use tensorflow_gpu and the NVIDIA CUDA® Deep Neural Network library (cuDNN)[10] instead of regular tensorflow to train the models and speed up training time, however, we didn't notice a large change in training time. The performance of agents was improved in training mainly through trial and error methods, consisting of changing reward values in the player agent scripts and manipulating certain player variables such as movement speed then adjusting such values further after observing how the agent reacts to such changes. For more information on how reward values were manipulated, please refer to Section 4.1. Tensorboard[11] was used to observe the performance of the player as it provided useful graphs such as the number of steps the agents takes against their cumulative reward.

3.3 Creating an Agent

In addition to building the base game, the agent and training environment created for the runner game was implemented using the ML-Agent toolkit for Unity. First, an agent script was created for the main player containing the actions the agent can perform, the environment observations, and the rewards. There were two possible actions the agent could perform whilst playing - jumping and no action. The player was given a vector observation space size of 39, which includes the agents position, a boolean which changes depending on whether the player has reached a checkpoint, and a set of raycast observations. A total of five different raycast angles were given to the player (20°, 60°, 90°, 120°, 280°) and were set to only detect objects of certain tags ("KillBox", "Points", "Platform", "Checkpoint", "Obstacles"). Whenever the agent collides with one of the objects a positive or negative reward is added accordingly. Rewards were also adjusted depending on which actions the agent took, such as jumping on different platforms. For more information on how these rewards were added and decided please refer to Section 4.1. It should be noted that once the agent hit an obstacle or fell off a platform, a negative score was given and the training area was reset. Additionally, if he reached the checkpoint, the 'Done()' function will be called to notify the agent that he has completed his objective.

Next, an area script was created to reload the environment objects when necessary, for example, for situations where the player hits an obstacle and restarts from the beginning. The area script moves the player back to his original starting position and resets all the points that the player has collected. Finally, an academy script which represents the training environment for agents, was created for the possibility of training multiple agents. Unfortunately, we did not manage to fully implement this feature to a working standard with multiple agents because we encountered issues with the player. However, we managed to make it work with a single agent and in future versions of the runner game, we wish to further experiment with this feature. It should be noted that there were not many resources readily available online to refer to for implementing a runner game, so creating our own agent from scratch posed a big challenge.

[8]https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation.md

[9]https://github.com/Unity-Technologies/ml-agents/tree/0.10.1

[10]https://developer.nvidia.com/cudnn

[11]https://www.tensorflow.org/tensorboard

4. RESULTS

In order to train the agents and obtain satisfactory results, a lot of time and effort had to be put into adjusting and optimizing parameters. Different game parameters were tried and also different configurations. The agent's performance was compared using graphs and testing out the agents within the environment. Initially, the PPO and SAC algorithms were taken into consideration to train our models. These two learning algorithms were chosen since they are supported by ml-agents. After some testing, it was decided that training would be implemented using the PPO algorithm. This decision was taken due to the fact that training using SAC requires more model updates, making it better for heavier and slower environments. Training graphs can be viewed in real-time via *tensorboard* and these usually give an indication of whether an agent is learning or not. However, performance could not always be compared using these graphs as in order to optimize the agent's performance, different rewards had to be adjusted often. The ml-agents *learn* function produces an *NN* file which can then be attached to an agent's brain and the in-game performance could be observed. In certain cases, multiple agents were inserted into the game scene, each using different brains in order to identify which performs best. Different training configurations were also set in ML-Agent's *YAML* file.

4.1 Runner Game Results

At the start of training, the agent was only rewarded after interacting with objects around the scene. Specifically, it was positively rewarded for collecting points and reaching checkpoints then, negatively rewarded for hitting obstacles and platforms. These values were adjusted several times according to trial and error. Training hyper-parameters were configured specifically for this game as shown in Figure **??**. The configuration was finalized by using the training hyperparameters of similar games from the ML-Agents examples and some testing by trial and error was also done. The most notable change from the default parameters is the activation of the use_recurrent function which allows the agent to store a vector of floats which are used when making future decisions, providing the agent with a "memory".

Next, it was observed that the agent would play the game by constantly selecting the jump action once the player was grounded. To discourage this, we added a statement to the player script that fires once the player lands on the ground and checks whether the platform they landed on is the same as the platform they were previously on. If the platforms differ, a positive reward is given. On the other hand, if the platforms are equivalent a negative reward is given to discourage the player from jumping for no reason. In addition to this, a slight positive reward was given to the player whenever they choose to perform no action and a slight negative reward was given when they choose to jump. After implementing these changes, the agent still jumps frequently but there are occasions in the game when he decides to perform no actions and roll prior to jumping.

To increase the agents overall performance, a statement was added to accumulate the agents reward based on the distance he travels from his starting position. The further the agent travels, the higher is his reward gain to encourage him in choosing actions which take him longer distances towards the checkpoint. Additionally, the background was removed from training scenes to try improve the agents performance as it was using a parallax moving background pre-

```
RunnerLearner:
    use_recurrent: true
    sequence_length: 64
    num_layers: 2
    hidden_units: 128
    memory_size: 256
    beta: 1.0e-2
    num_epoch: 3
    buffer_size: 1024
    batch_size: 128
    max_steps: 5.0e6
    summary_freq: 2000
    time_horizon: 64
```

Figure 18: Training hyper-parameter configurations set in the *yaml* file.

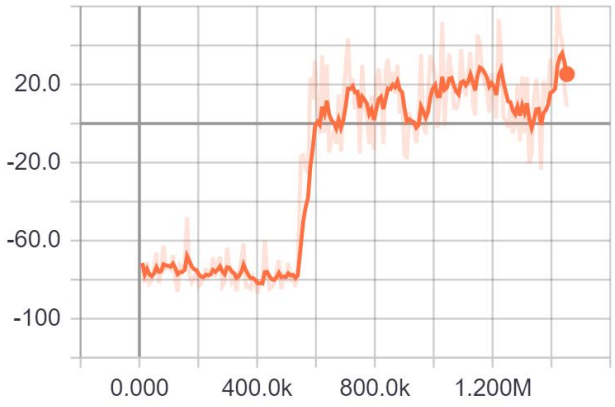

Figure 19: This graph showing *Environment Steps vs. Reward* shows one of the training instances carried out. The sharp incline seen towards the middle of the graph shows the point where points were introduced to the training session. The incline is due to the extra reward which is now being given to the agent whenever it collides with an object marked as a point.

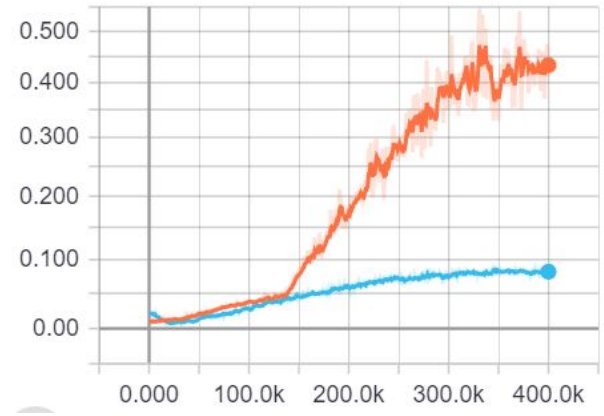

Figure 20: An environment step vs rewards graph showing the high rewards obtained when training using the same brain, represented by the orange plot and the lower rewards obtained when training 2 different brains, represented by the blue plot.

viously which could hinder his training progress. Moreover, another technique we tried for each level involved removing the points and obstacles, so that there are only platforms for initial training then introducing them slowly once the agent has adjusted to the environment so that it could adapt accordingly. This could encourage the performance to find connections between game objects during training rather than introducing them all at once.

Although significant improvement was made by the agent from his starting implementation, there is still a lot of room available for further improvement. For instance, a possibility would be restricting the agents z rotation in a way that it wouldn't affect the agents gameplay mechanically but would make a visual difference so that the agent could learn better.

4.2 Table Tennis Results

Training for this mini-game was quite a challenge due to the game's difficulty. Several training methods were tested out in order to obtain the best agent. Due to the fact that this is a two-player game where the actions of one agent will affect the other agent, different agent combinations were tried out. Training was done using multiple environments at a time in order to obtain better results. First, the agents trained using the same brain. Although not providing the best agents, this method provided the agents with the best rewards. This is due to the fact that since they have the same brain, they get used to playing together, hence at the end of the training, none of the agents were missing any hits or gaining any points. Next, two different brains without a set model were tried out. This method provided better models than that from the first test, even though the reward scores were much lower, as can be seen in the graph depicted in Figure 20.

Some other testing was done by training one brain with a model and one without and training only one brain. Training graphs for some of these tests are shown in Figure 21. When testing the models obtained, it was noted that the rewards were quite random and in the case of this mini-game, they did not signify anything about the performance of the agent. Hence, performance testing was carried out by making agents play against each other.

4.3 Kart Racing Results

The training for this mini-game proved to be quite successful. After a couple of steps, agents were observed to bump into the wall and find it extremely difficult to get back onto the track. After all the training was completed, an agent which does not bump into any walls and plays the game almost flawlessly was created. In order to train this agent, several parameters were tried out and the re-

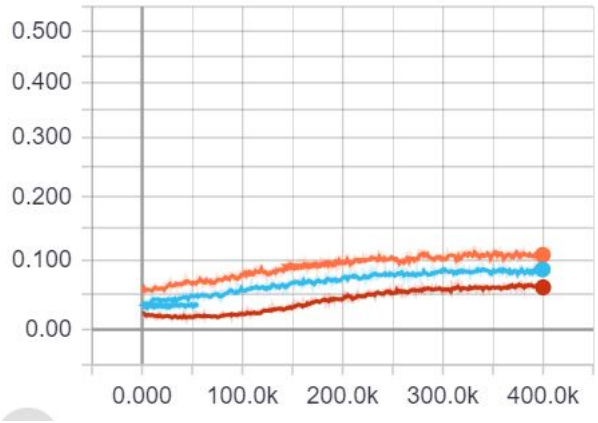

Figure 21: *Orange* : An empty brain training against a brain with a model which is not training. *Blue* : An empty brain training against a brain with a model which is also continuing to train. *Red* : A combination of all the best models trained multiple times.

ward system was changed multiple times. The training scene also included multiple agents using the same brain training at once in order to obtain faster results. An example of a test which was carried out was whether penalising the agent when he got close to the wall trained the agent better. Testing showed that models which were penalised when touching the wall and models which weren't obtained similar results. Some differences in playing were that the model penalised for touching the wall was staying away from it, even when it meant that a longer path had to be taken. The best model created takes advantage of both these models and was trained using the wall-penalisation model for the first steps and this model was then used to train with different rewards. The graph shown in Image 22 visualizes the training progress of the model.

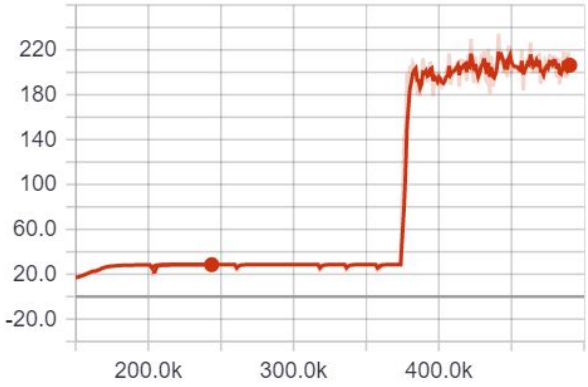

Figure 22: A smoothened training progress of the kart model. The dot in the first part of the graph symbolizes where the parameters where changed, as explained previously. The x-axis shows the number of training steps whilst the y-axis shows the cumulative reward.

One can see that after a certain number of epochs, the graph shows a sharp incline. At this point, the agents most probably started reaching the finish line, thus receiving a high reward, since they have completed the main goal. Previously, the maximum number of steps was being reached before the agents got to the finish line, thus no reward was given.

5. CONCLUSIONS AND FUTURE WORK

Both of the mini-games provided relatively positive results. Agents were trained such that they provide a challenge when playing against human players. Different levels of difficulty were also produced from training the agents and some interesting results were obtained from training the table tennis game using different brain combinations.

In the runner game it was concluded that training the agent with no obstacles or points initially then introducing them slowly yield positive results. In addition to this, removing parallax backgrounds, awarding the distance reached and reducing rewards for unnecessary jumping all positively impacted the agents training.

For future improvements, we plan on implementing a functional difficulty option for the game so that players could play against different difficulties of agents for the mini games. For the mini games, we plan on experimenting further with different training values and parameters to train agents with higher intelligence that are extremely difficult to beat. In the runner game, we would like to experiment further with the academy script and create a functional academy that could train multiple agents at once which could potentially yield AI with high performance. We also plan on investigating further rewards and parameters that could be used to discourage the agent from jumping unless needing to and reaching further stages of the game. Other improvements include:

- experimenting with different ML-Agents versions and perhaps find one in which tensorflow_gpu is supported and could increase training time
- adding a human-AI version of the runner game where the player races against an AI to reach the final checkpoint
- adding more levels to the game with more challenging obstacles and more mini games

6. REFERENCES

[1] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[2] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[3] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[4] EL Thorndike. Provisional laws of acquired behavior or learning. *Animal Intelligence (New York: The Mc Millian Company)*, 1911.

[5] Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain model problem*. Princeton University., 1954.

[6] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[7] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011.

[8] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[10] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[12] Ivan Sorokin, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, and Anastasiia Ignateva. Deep attention recurrent q-network. *arXiv preprint arXiv:1512.01693*, 2015.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[14] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[15] Patrick Nadeem Ward, Ariella Smofsky, and Avishek Joey Bose. Improving exploration in soft-actor-critic with normalizing flows policies. *arXiv preprint arXiv:1906.02771*, 2019.

[16] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.