

ICS 2000: Autonomous Baby Shark GAPT Documentation

Kian Parnis (0107601L), Kieron Sultana (0330701L), Evangeline Azzopardi (33600H)
B.Sc IT Artificial Intelligence

22nd May 2022

Contents

1	Introduction	3
2	The Challenge & Approach chosen	3
3	Literature Review	5
3.1	A* Algorithm	5
3.2	Reinforcement Learning	5
3.2.1	Introduction	5
3.2.2	Deep Q-Learning	6
3.2.3	Proximal Policy Optimisation	6
3.2.4	Policy Optimization - Policy Gradient Methods	6
3.3	Imitation Learning	7
3.3.1	Introduction	7
3.3.2	Behavioural Cloning - Introduction	8
3.3.3	Behavioural Cloning from Observation	8
3.3.4	Inverse Dynamics Model Learning	8
3.3.5	Behavioural Cloning	9
3.3.6	Generative Adversarial Imitation Learning	9
4	Explanation of the Implementation	10
4.1	Details of development environment	10
4.2	The Main Game Environment	10
4.3	Visuals & UI	11

4.4	AI Implementation	12
4.4.1	Unity: AI Setup	13
4.4.2	Game Environment: AI Setup	14
5	Testing, Results & Evaluation	16
6	Distribution of Work	21
7	Bibliography	22

1 Introduction

The aim of this project is to investigate some of the possible different methods of pathfinding to avoid obstacles and obtain collectables. The three main methods investigated to achieve path finding in this report are the A* algorithm, Reinforcement Learning and Imitation Learning. Of these three options, Reinforcement Learning and Imitation Learning were implemented and had their results compared.

2 The Challenge & Approach chosen

The challenge presented is one that entails designing a game which utilises pathfinding through various techniques for an agent to navigate its environment in real time whilst avoiding obstacles and obtaining collectables, which are spawned randomly, to maximise a score.

As will be explained in a future section going over the details of the implementation as well as the literature review, Reinforcement Learning and Imitation Learning were chosen. This was decided as the setup of the environment to accommodate these two techniques would allow for randomly spawning obstacles and collectables. A* would have required an environment based on grid maps, also covered in the literature review, which limits the randomness of obstacles' or collectables' placement.

Furthermore, these two learning techniques were selected in order to compare and contrast the results obtained. This will be discussed in its respective section.

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

~~I/~~ We*, the undersigned, declare that the [~~assignment~~/ Assigned Practical Task report /~~Final Year Project report~~] submitted is ~~my~~ / our* work, except where acknowledged and referenced.

~~I/~~ We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.


* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).


Evangeline Azzopardi
Student Name



Signature

Kian Parnis
Student Name


Signature

Kieron Sultana
Student Name


Signature


Student Name


Signature

ICS 2000
Course Code

GAPT: Baby Shark
Title of work submitted

22nd May 2022
Date

3 Literature Review

By reference throughout this literature review pathfinding can be accomplished through the use of: A* Algorithm, Reinforcement Learning and Imitation Learning.

3.1 A* Algorithm

Grid maps are used to discretise/represent the game's map in terms of a search graph. The process partitions the map into cells, also called tiles. Depending on the topology of the map, a tile is labelled as traversable or blocked with traversable tiles becoming nodes in the graph representation. A mobile unit can only occupy one traversable tile, or node, at a time. Nodes are connected by adjacent traversable nodes in 4 cardinal directions or in 4 cardinal directions and 4 diagonal directions [1].

A* is the de facto standard for pathfinding [1]. It is an informed generic searching algorithm which is often used for its completeness, optimality and efficiency [2]. This algorithm determines its path by maintaining a tree of paths, determined by the grid map explained above, from the starting point to the terminating point. Each iteration of the main loop in A* determines which path to extend based on the cost of the path, hence the algorithm selects the path that minimises:

$$f(n) = g(n) + h(n)$$

For the above: n is the next node in the path, $g(n)$ is the cost of the path from the starting node to n , and $h(n)$ is a heuristic function which estimates the cheapest cost of the path from n to the goal/ terminating node. The algorithm terminates once a path is found from the start to end or if there is no possible eligible path.

A* is a popular choice for pathfinding in video games [2]. Utilising it to produce optimal results is dependent on the nature of the internal environment. Consider the following case [2]: a rectangular grid of 1000x1000 squares would produce a search space of 1,000,000 squares. The speed of the algorithm is not optimal in such a case; hence A*'s efficiency is dependent on the size of the search space [2].

3.2 Reinforcement Learning

3.2.1 Introduction

An **Artificial Neural Network**, ANN, is an information-processing system which accepts individual input values and predicts an output value based on said input, in this case the output is a path to the target.

The input is passed into a neuron which has an associated weight value. A **Feed Forward Neural Network**, FFNN, trained through backpropagation can be applied to real-time pathfinding due to its speed [4]. An alternative to a FFNN is to utilise **reinforcement learning** to train the ANN through a **Genetic Algorithm** [4], GA, where the AI agent is rewarded for following a set of rules. The agents' scores are ranked and the weights from the top-ranking agents are fed into lower ranking agents as a method of training.

There are two requirements which the ANN needs to be trained for:

- Choose the correct path from the goal to the target.
- Avoid any obstacles littering the path.
- Collect collectables to obtain the highest score possible.

The ANN is initially trained to do these tasks separately and then training is dedicated to do them simultaneously.

3.2.2 Deep Q-Learning

Within reinforcement learning Q-Learning is also applicable [5]. Q-Learning is an off policy/ model-free reinforcement learning algorithm which does not require knowledge of its surrounding environment, hence model-free. Furthermore, it can handle scenarios with stochastic transitions without requiring adaptation.

The ‘Q’ in Q-Learning stands for quality, referring to how useful an action taken is towards gaining a future reward. Consider the following: An agent performs an action a_t , receives a reward signal r_t , and moves from a state s_t to another state s_t' . The Q-value for the state action pair $Q(s,a)$ is updated to $Q_{new}(s_t,a_t)$ using the equation:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_t + \underbrace{\gamma \cdot \max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

temporal difference

Figure 1: Equation obtained from [6]

It has been proven that for a finite Markov Decision Process, MDP, the Q-value will converge [7]. However, it is important to note that convergence is not guaranteed in a practical setting due to training time constraints [7]. The Q values are stored in a data table.

If the state space is significantly large then the required amount of memory and the number of iterations required to reach convergence makes the use of a data table ineffective [7]. This can be solved through Deep Q-Learning which utilises the ANN, specifically a Convolution Neural Network, CNN, mentioned previously. The Q-values are interpolated/inserted/estimated by the CNN where the input is a state s and the output is a vector of all possible Q-values for all possible actions from s [7].

3.2.3 Proximal Policy Optimisation

Proximal Policy Optimisation is an advancement within the field of Reinforcement Learning and it provides an improvement on the Trust Region Policy Optimisation, TRPO [12]. A policy is defined as a mapping from an action space to a state space, a means of instructions to a reinforcement learning, RL, agent. The agent is evaluated, through the policy, to gauge its success, in this case in navigating the environment. This is achieved through calculating the policy gradients [12].

3.2.4 Policy Optimization - Policy Gradient Methods

Policy gradient methods compute an estimate of the policy gradient and use it a stochastic gradient ascent algorithm. The most commonly used gradient estimator is:

$$\hat{g} = \hat{E}_t [\nabla_{\theta} \log \pi_{\theta} (a_t | s_t) \hat{A}_t]$$

where π_{θ} is a stochastic policy and \hat{A}_t is an estimate of the advantage function at a timestep t . $\hat{E}_t[\cdot]$ is the expectation which, is the empirical average over a finite pool of samples, in an algorithm that will alternate between sampling and optimisation. In the case that automatic differentiation software is used by constructing an objective function with a gradient that is the policy gradient estimator, \hat{g} is calculated through differentiation of the objective:

$$L^{PG}(\theta) = \hat{E}_t [\nabla_{\theta} \log \pi_{\theta} (a_t | s_t) \hat{A}_t].$$

This performs multiple optimisation steps on this loss L^{PG} while using the same trajectory, however this can lead

to destructively large policy updates [11].

The algorithm for PPO:

Algorithm 1 PPO, Actor-Critic Style

```

for iteration=1, 2, ... do
  for actor=1, 2, ...,  $N$  do
    Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps
    Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
  end for
  Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
   $\theta_{\text{old}} \leftarrow \theta$ 
end for

```

Figure 2: Equation obtained from [11]

3.3 Imitation Learning

3.3.1 Introduction

Imitation learning is also a contender for pathfinding. The authors of [8] state that “computer games provide an excellent testbed for the learning of complex behaviours. For most genres, the behaviour of game characters will be composed of reactive, tactical and strategic decisions.” As a result, gameplay can be learnt by an AI agent.

Imitation learning is ideal when an expert in the area of interest can demonstrate the desired behaviour which the AI agent follows. This means that the reward function to generate the same behaviour is not specified or the policy is not learnt directly [9].

The environment for imitation learning consists of a Markov Decision Process, MDP [9]. This is a 5-tuple consisting of set of states S , a set of actions A , a transition model $P(s' | s, a)$, where the probability of s' is dependent on being in state s and conducting action a , an undefined reward function $R(s, a)$ and the discount factor γ [10]. The AI agent will perform different actions based on a policy π . Through the expert’s demonstrations, $\tau^*(s_0, a_0, s_1, a_1, \dots, s_n, a_n)$, also referred to as trajectories, the actions are automatically based on an optimal imitation policy π^* . The set of state transitions which an agent follows when executing the policy is denoted by $T_\pi = \{(s, s')\}$. In relation to the set of state transitions T_π , there is the inverse dynamics model, $M_a^{s_i s_{i+1}} = P(a | s_i, s_{i+1})$, which denotes the probability of a taken action, a , given the agent is transitioning from state s_i to s_{i+1} .

The agent’s ability to learn the policy is based on the training time provided and the quantity and quality of the demonstrations [9] [10]. For a task-independent model, it is assumed that some of the state features are specifically associated with the task at hand and others to the agent [10]. This entails that a state s can be partitioned/coupled into an agent-specific, s^a , and task-specific, s^t , state which are members of the sets S^a and S^t respectively, $S = S^a \times S^t$ [10]. Through the partitioning/coupling mentioned above an agent-specific inverse dynamics model can be defined and denoted using M_θ : $S^a \times S^a \rightarrow p(A)$. This model maps a pair of agent specific transitions $(s_i^a, s_{i+1}^a) \in T_\pi^a$ to a distribution of actions that are likely to have resulted in said transition.

As previous explained, imitation learning can be defined in terms of a MDP without a pre-defined reward. Hence the learning problem for the agent is to determine the imitation policy by using a set of expert demonstrations, $\{\xi_1, \xi_2, \dots\}$, which it has access to. If the agent does not have access to the expert’s demonstrations, then the learning problem is referred to as imitation by observation [10]. In this case, the imitation policy is derived from a set of state-only demonstrations $D = \{\zeta_1, \zeta_2, \dots\}$ where ζ is a state-only trajectory $\{s_0, s_1, \dots, s_n\}$.

3.3.2 Behavioural Cloning - Introduction

Behavioural Cloning is considered the simplest method of imitation learning which focuses on learning the policy through supervised learning. Hence, it is also related to model-based learning and since this method is based on a learned model it is simple [9] and sample-efficient [10]. Furthermore, a learned model can be applied to various tasks of the same manner [10].

Algorithm 1 BCO(α)

```

1: Initialize the model  $\mathcal{M}_\theta$  as random approximator
2: Set  $\pi_\phi$  to be a random policy
3: Set  $I = |\mathcal{I}^{pre}|$ 
4: while policy improvement do
5:   for time-step  $t=1$  to  $I$  do
6:     Generate samples  $(s_t^a, s_{t+1}^a)$  and  $a_t$  using  $\pi_\phi$ 
7:     Append samples  $\mathcal{T}_{\pi_\phi} \leftarrow (s_t^a, s_{t+1}^a), \mathcal{A}_{\pi_\phi} \leftarrow a_t$ 
8:   end for
9:   Improve  $\mathcal{M}_\theta$  by modelLearning( $\mathcal{T}_{\pi_\phi}, \mathcal{A}_{\pi_\phi}$ )
10:  Generate set of agent-specific state transitions  $\mathcal{T}_{demo}^a$ 
    from the demonstrated state trajectories  $D_{demo}$ 
11:  Use  $\mathcal{M}_\theta$  with  $\mathcal{T}_{demo}^a$  to approximate  $\tilde{\mathcal{A}}_{demo}$ 
12:  Improve  $\pi_\phi$  by behavioralCloning( $S_{demo}, \tilde{\mathcal{A}}_{demo}$ )
13:  Set  $I = \alpha |\mathcal{I}^{pre}|$ 
14: end while

```

Figure 3: Equation obtained from [10]

3.3.3 Behavioural Cloning from Observation

Behavioural Cloning from Observation, BCO, is achieved through combining inverse dynamics model learning with learning an imitation policy [10]. Before the agent is allowed to observe any of the demonstration information it is given time to determine its own specific inverse dynamics model. The state-only demonstration information is then provided to infer any missing information that the expert did not encounter/include. The agent then performs imitation learning through a modified version of behavioural cloning.

3.3.4 Inverse Dynamics Model Learning

To fill in the gaps containing missing action information, the agent is given the opportunity to acquire experience through the use of an Inverse Dynamics Model [10], which can also be agent-specific according to the task at hand. The agent performs an exploration policy π , which can be set to random, and while executing this policy the agent interacts with its environment, referred to as \mathcal{I}^{pre} . In the case of an agent-specific inverse dynamics model, the agent-specific part of the states of \mathcal{I}^{pre} are extracted and stored as $\mathcal{T}_{\pi_e}^a = \{(s_i^a, s_{i+1}^a)\}$ with their associated actions $\mathcal{A}_{\pi_e} = \{a_i\}$. Given this, the remaining problem related to learning an agent-specific inverse dynamics model is to find the parameter θ for which \mathcal{M}_θ best describes the transitions observed. This issue can possibly be resolved as treating it as one maximum-likelihood estimation, seeking θ^* as $\theta^* = \operatorname{argmax}_\theta \pi_{i=0}^{|\mathcal{I}^{pre}|} p_\theta(a_i | s_i^a, s_{i+1}^a)$, p_θ is the conditional distribution over the actions induced by \mathcal{M}_θ given a specific state transition. Any supervised learning technique denoted as “modelLearning” in the algorithm above can be utilised to solve θ^* .

For an environment that consists of a continuous action space, a neural network for \mathcal{M}_θ [10] is utilised where the network will receive a state transition as input and will output the mean for each action dimension. The network can be trained, to find θ^* , using stochastic gradient decent, the gradient for each sample is computed by calculating the change in θ . This would increase the probability of a_i with respect to the distribution specified by $\mathcal{M}_\theta(s_i, s_{i+1})$. For an environment that consists of a discrete action space, a neural network for \mathcal{M}_θ [10] is utilised where the network will compute the probability of taking an action through a softmax function.

3.3.5 Behavioural Cloning

One of the long-standing issues is to find the appropriate imitation policy [10] from the set of state-only demonstration trajectories, $D_{demo} = \{\zeta_1, \dots, \zeta_n\}$ where each ζ is a trajectory $\{s_0, s_1, \dots, s_n\}$. To use the learned agent-specific inverse dynamics model, the agent-specific part of the demonstrated state sequences is extracted and forms a set of demonstrated agent-specific states transitions T_{demo}^a . For each transition $(s_i^a, s_{i+1}^a) \in T_{demo}^a$, the algorithm computes the model-predicted distribution over the demonstrator actions $M_\theta(s_i^a, s_{i+1}^a)$ and utilises the maximum-likelihood action as the inferred action \bar{a}_i . A set of state-action pairs is created using the inferred actions and now the imitation policy π_φ can be found. This is considered as a problem associated with behavioural cloning: given a set of state-action tuples, the task of learning the imitation policy is that of finding φ where π_φ best matches the set initially mentioned where this parameter is found using maximum-likelihood estimation.

3.3.6 Generative Adversarial Imitation Learning

GAIL is an Inverse Reinforcement Learning, IRL, algorithm based on Generative Adversarial Networks, GAN, and can be defined as model-free. Such an algorithm has been said to be effective at imitating complex behaviour in large and highly-dimensional environments and handle unseen situations well [13]. Within a GAN there are two networks: the generator and the discriminator. The generator creates new data points by observing and learning the distribution of the input dataset used, whilst the discriminator classifies a data point as created by the generator or if it is from the given dataset. Through the combination of IRL and GAN, GAIL is able to learn from a small number of expert trajectories. The goal is to train the generator to emulate the behaviour provided by the expert and the discriminator serves as a reward function which judges the quality of the behaviour.

Algorithm 1 Generative adversarial imitation learning

- 1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
- 2: **for** $i = 0, 1, 2, \dots$ **do**
- 3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
- 4: Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i} [\nabla_w \log(D_w(s, a))] + \hat{\mathbb{E}}_{\tau_E} [\nabla_w \log(1 - D_w(s, a))] \quad (17)$$

- 5: Take a policy step from θ_i to θ_{i+1} , using the TRPO rule with cost function $\log(D_{w_{i+1}}(s, a))$. Specifically, take a KL-constrained natural gradient step with

$$\begin{aligned} & \hat{\mathbb{E}}_{\tau_i} [\nabla_\theta \log \pi_\theta(a|s) Q(s, a)] - \lambda \nabla_\theta H(\pi_\theta), \\ & \text{where } Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i} [\log(D_{w_{i+1}}(s, a)) \mid s_0 = \bar{s}, a_0 = \bar{a}] \end{aligned} \quad (18)$$

- 6: **end for**
-

Figure 4: Equation obtained from [14]

4 Explanation of the Implementation

4.1 Details of development environment

The solution to this task was implemented within the Unity game engine. This platform presents out of the box features for better game creation and scripting by making use of the C# programming language. The Unity version used throughout this implementation is Unity 2020.3.33f1.

4.2 The Main Game Environment

The scripts for the main environment can be found within the directory: Assets/Environment Scripts.

Prior to the implementation process, the main concern addressed was with regards to how the agent would interact with environment and be trained successfully. If for instance, the A* algorithm for pathfinding were to be implemented as the core AI behind the agent, the environment would have to be set up in such a way for the agent to be able to navigate through the environment. A possible way to solve this would be, for example, setting up a grid map where the agent would be able to calculate a route based on each tile inside the grid map. However, with Reinforcement Learning, RL, the setup of the environment differs from that of traditional planning. For RL, the main challenge for setting up an environment is that it must be set up in a meaningful way where proper learning would be achieved. In this case what's most important is that the RL interaction loop is satisfied. The agent should be able to take an action a at time t and in real time receive a reward and an observation for it to continue learning.

The main thought process behind creating the environment is how the agent models the environment.

In order to calculate what the best action for the agent to take is at any point in time, the current state of the agent would have to be taken into consideration. One way to go about this would be to maintain a 'QTable', this table would store all the possible combinations of states and actions with their corresponding q-values. In the context of this scenario, making use of a q-table would not be feasible due to the number of state-action pairs that would accumulate within the table. Hence, in order to accommodate for the large quantity, function approximation techniques were used to group similar states together and generalise any states which are yet to be seen. Since the AI can generalise states and group them accordingly, there is more freedom with how the environment can be setup. When implementing the mapped coordinates for spawning each obstacle, a continuous y-axis value can be used as a result of the previous generalisation.

The process of implementing the environment is as follows:

Deciding the maximum environment size. The main agent only possesses three possible actions; moving upwards, staying still, or moving downwards, thus walls were added as boundaries at the top and bottom of the game, relative to the screen size, to deny the agent from leaving the dedicated area.

With regards to the agent, before any AI was implemented, a preparation script was created that lets a user manually control the agent. This was used for debugging purposes and further down the line be able to take control of the agent to record data for behavioural cloning. This movement script allows the user to apply all three actions UP, DOWN, STAY ON COURSE via the arrow keys.

As previously mentioned, obstacles can be spawned at any discrete y-value. These obstacles will need to be spawned at the far right of the screen and with each timestep move slowly towards and possibly move past the agent. This was achieved by first creating 'prefabs' of all six obstacles the shark would want to avoid or 'capture'. To each of these prefabs a script was attached that depending on the set speed multiplied by every frame, would continuously move the obstacles towards the right of the screen. The process of generating obstacles was implemented by having a controller game object start a timer for each obstacle. These timers would vary in the maximum time allotted and once a time zero is reached, a clone of the object would be initialised, and the timer would start again. A game object labelled 'Instantiate' was positioned to the far right of the screen which marks the x-axis the cloned

objects would spawn at followed by a random increase or decrease of where they can spawn at in the y-axis. The total randomisation of these obstacles both in the coordinates they spawn in, as well as what time they can spawn at is crucial. This is so that when the AI starts training it will have to learn to better generalise the scenario and not just learn to follow a set pattern of how these obstacles are generated. Additionally, a script was created that would automatically destroy any obstacle that would move out to the far left of the screen and prevent any potential build-up of objects from occurring. Furthermore, on the rare occasion that two obstacles were to be generated on top of one another, a script was created that would handle these occurrences by destroying both objects.

With the successful implementation of both the agent and the obstacle generation system, another environmental factor was considered. Reinforcement learning trains episodically: an episode starts, the environment is initialised, and the episode plays out. Eventually a final state is reached, and the episode would restart. Thus, it had to be decided when an episode would end. An episode can only end once the agent is unsuccessful and hits an obstacle that should have been avoided as this would mark the end state.

The script ‘Reset Scene’ handles the functionality of restarting an episode and it is called whenever the agent collides with any of the three obstacles that need to be avoided, which are identified by their assigned tag. Once this script is called, it starts off by first destroying all the instantiated obstacles on the screen, resetting the current score that the current agent would have achieved and finally resetting the position of the agent back to the starting state. With this, an episode can play out, end and a new episode can start exactly like that of the one prior.

The final part of the environment setup was the functionality to let the user to choose any position on the screen and click anywhere on the screen to instantiate either of the two types of obstacles, causing the agent to replan its path accordingly. This was achieved by giving the user a choice between the two types of obstacles that they can spawn. This was done through the placement of buttons at the lower right-hand side of the screen; once the user presses on any of the two buttons, a visual indicator of what’s currently selected is given and then the user can simply right-click anywhere on the screen and spawn the desired obstacle.

4.3 Visuals & UI

The game is targeted towards a younger audience. In order to achieve this a cartoon aesthetic was chosen requiring custom characters and backgrounds to be drawn, along with appropriate text fonts. To bring these characters to life sprite sheets were drawn and later animated that would perform simple animations according to the current ongoing events of the game.

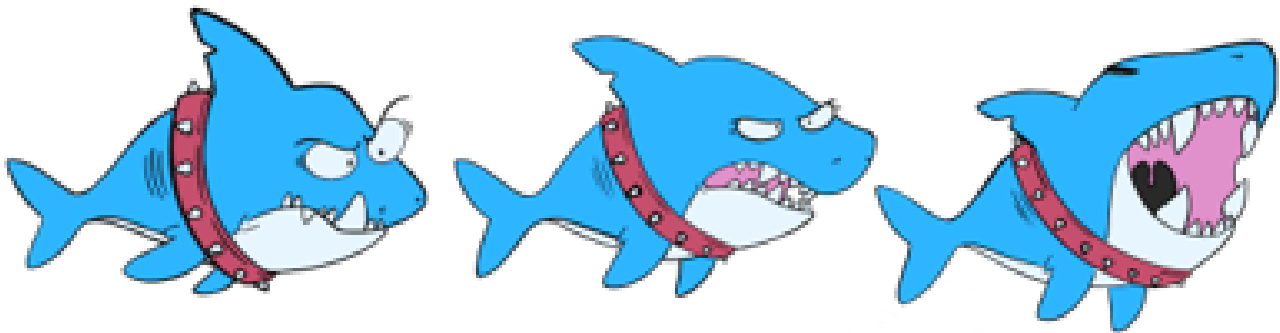


Figure 5: Baby Shark

For example, the baby shark character has two distinct animations. A swimming animation would be played when the character is navigating along the y-axis and a biting animation that would be played, alongside a matching sound effect, when the shark successfully manages to ‘eat’ a fish. Minor animations were created for the fish for

believability when transitioning across the screen.

With regards to the background, a parallax effect was implemented consisting of multiple backgrounds being layered on the z axis. A simple script was written that would scroll the background content at a lower speed than the foreground content to create an illusion of depth to the 2D scene.

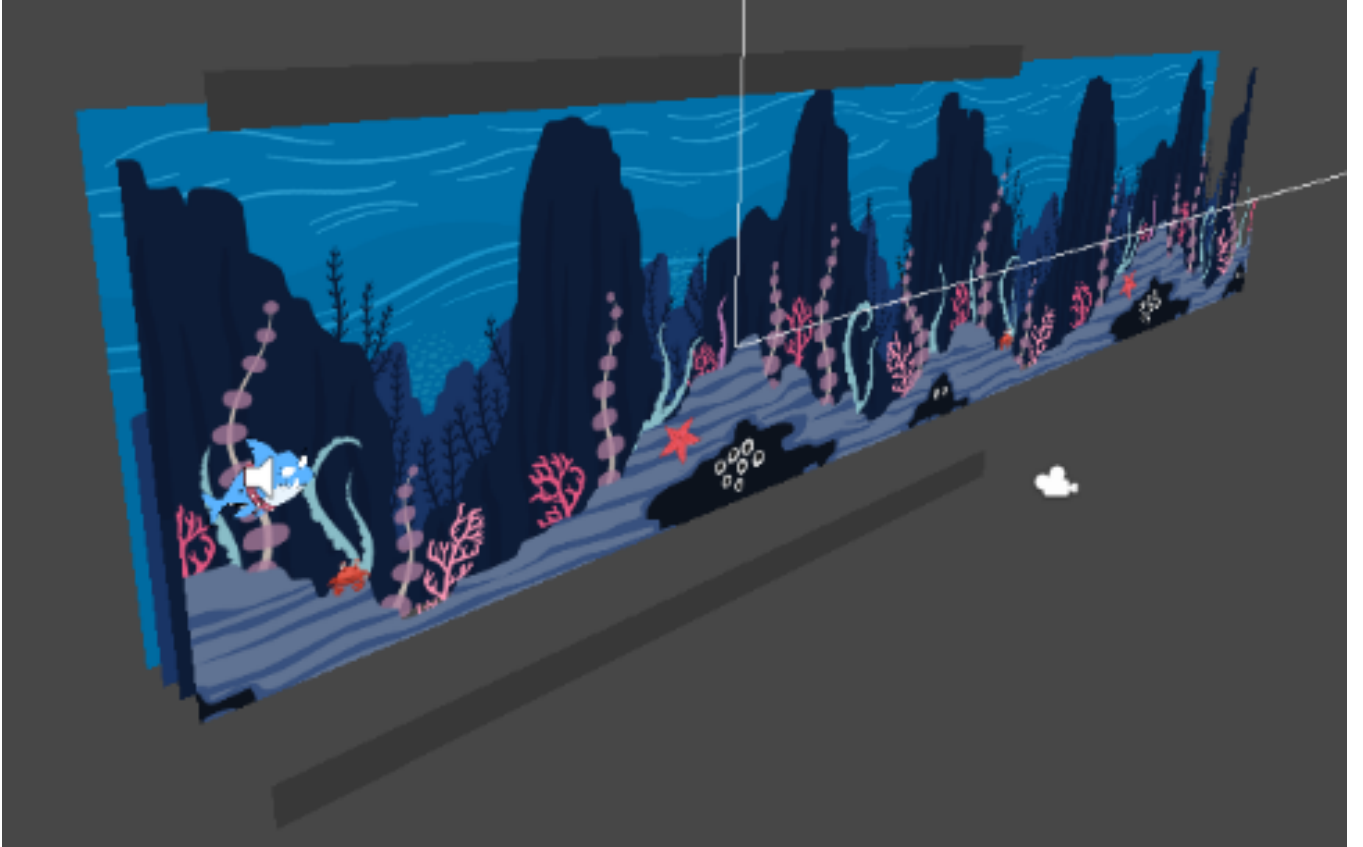


Figure 6: The parallax background implemented

A simple score script ‘Game Data’ monitors the current high score achieved by the agent. The current score is incremented when the agent successfully collides with a collectable, a fish. The score is incremented accordingly with more points allocated to the bigger sized fish. When the episode ends, this score would be compared to the current high score and if the former is higher than the latter the current high score is updated to the new value.

When the game is loaded by the user, a way to better communicate with how the AI/game actually plays out was achieved by first creating a main menu that would give the users several options: to start the game, a settings tab that allows the user to configure the sfx and music volume, an option to terminate the game and a ‘how to play’ option that informs the user more about the game. The main menu is loaded first when the user runs the game and to go back to this menu from the game, the user can press the escape key and prompt a pause menu that halts everything on screen and a button to return to the main menu would be present.

4.4 AI Implementation

The Unity ML-Agents toolkit was used as a foundation to implement both AI techniques. ML-Agents is an open-source project that enables easier implementation, based on PyTorch, of state-of-the-art algorithms such as reinforcement learning, imitation learning, and neuro-evolution to train intelligent agents for both 2D and 3D environments [15].

The following are the versions used for the main toolkit as well as its dependencies:

- mlagents: 0.22.0
- torch: 1.7.0+cu110
- tensorboard: 2.8.0

4.4.1 Unity: AI Setup

ML-Agents provides the base scripts that can be altered to fit the flow of how certain AI learn. When training an AI, what is considered first are the state feature vectors which will be used to reduce the state space and handle unseen states. These features are pieces of information that are important for any decision the agent needs to make. This can be the velocity or bearing of a vehicle, the position of each piece on a chess board etc [16]. In the case of this implementation the features which are most important are: the positioning of the agent relative to each opposing wall and the positioning of each obstacle or collectable. To encapsulate these features a ray perception sensor was utilised: these types of sensors provide rays emanating from any target position to any length and degree chosen. In this case, a total of 14 rays were used emanating from the agent at 95 degrees to better capture all important features listed prior.

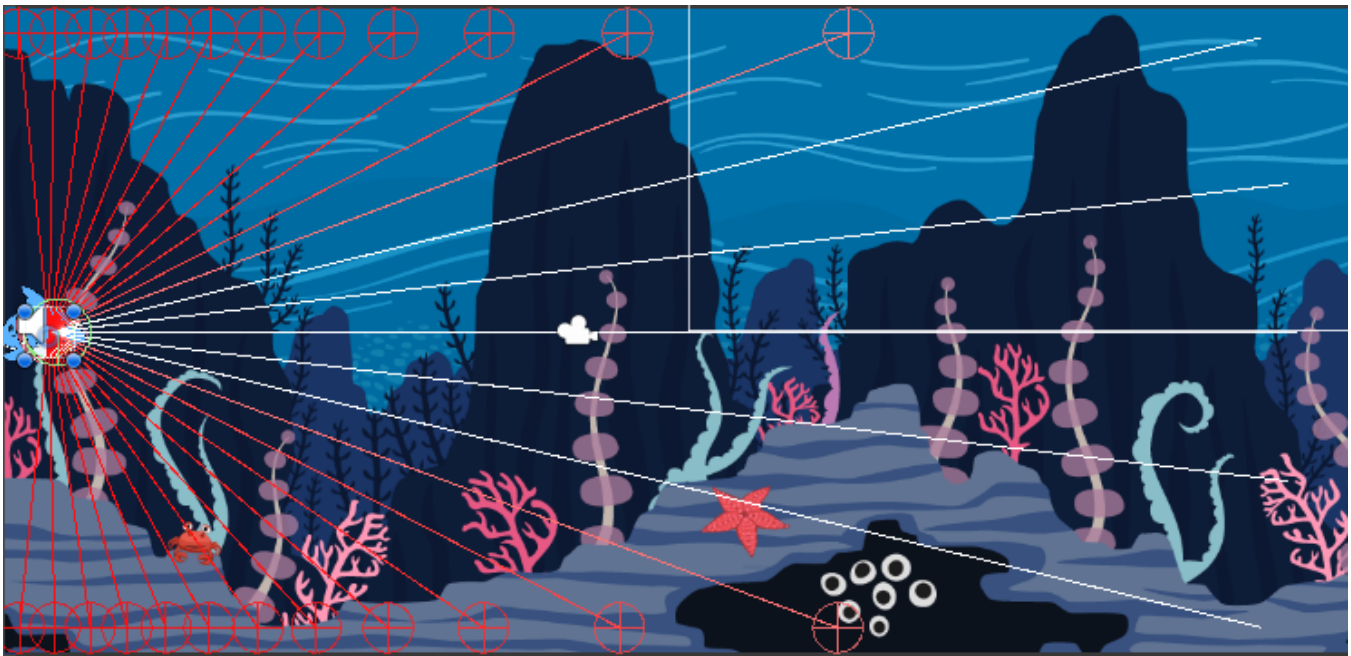


Figure 7: The ray perceptor sensors implemented

With proper feature detection added, the next thing the agent would need to receive is a specific intermediary-reward after taking an action in some state and a final reward which is given when an end state is reached. The ‘MLPlayer’ script was used from ML-Agents and customised to fit this need. ML-Agents provides helper functions such as ‘AddReward’ to better control what reward the agent gets throughout the course of an episode. The function ‘OnActionReceived’ is what allows the agent to know what actions it is allowed to take, where in this case these actions are: MOVEUP, MOVEDOWN, STAYONCOURSE.

The following are how each reward is distributed to the agent:

- A small reward of +0.001 is given to the agent just for staying alive, this is given to deter the agent maximizing potential regret and focusing more on maximizing the cumulative reward.
- A reward of -0.25 is given if the actions taken by the agent are MOVEUP, MOVEDOWN. This is so the agent prioritises the action STAYONCOURSE if no replanning needs to take place.
- To deter the agent from following a strategy of staying close to either the bottom or top wall a reward of -2.0 is constantly given until they get away from the wall.
- A heavy negative reward of -1000.0 is given to the agent if they collide with any of the obstacles which should be avoided. This is done to promote the idea of avoiding these obstacles at any cost and will lead to the cumulative reward starting from a large negative value and slowly progress and reach a positive value.
- To learn to prioritise collectable, they have the following rewards: Big fish +12, Medium fish +8 and Little fish +4.

The final scripts utilised from ML-Agents are the ‘Decision Requester’ and ‘Demonstration Recorder’ scripts which allow the functionality of recording data by letting a player play so that when it comes to training Imitation Learning, data is collected and then utilised by the imitation algorithm to train the agent and compare the agent’s results to the users.

4.4.2 Game Environment: AI Setup

When setting up the necessary packages, a python virtual environment was used to isolate the libraries used from any other libraries present on a system. Within this environment all the necessary packages were installed for ML-Agents. A config folder was created that contains two .yaml files. These files are used to mark what parameters the algorithms take, such as the learning rate, episode count, epsilon value etc.

For Imitation Learning extra parameters were specified for the configuration of Generative Adversarial Imitation Learning, GAIL, and Behavioural Cloning, BC, algorithms. When training, different values were used and tested such as varying the number of episodes, changing the strength of the parameters for GAIL and BC, as well as altering reward quantities to improve the AI’s convergence.

With everything setup, training took place by passing the location of the config files and an ID of the what the training session name should be:



```
(venv) C:\Users\Owner>mlagents-learn C:\Users\Owner\Desktop\Baby_Shark\config\Brain.yaml --run-id=PPOTrain1
-
Version information:
ml-agents: 0.22.0,
ml-agents-envs: 0.22.0,
Communicator API: 1.2.0,
PyTorch: 1.7.0+cu110
2022-05-12 11:51:02 INFO [learn.py:275] run_seed set to 7992
2022-05-12 11:51:04 INFO [environment.py:205] Listening on port 5004. Start training by pressing the Play button in the Unity Editor.
```

Figure 8: A screenshot of the command prompt used to execute the training process

After this, training starts when the game is played within the Unity editor. Options to pause, resume training and continue training from an already existing brain are also available and were utilised when training each model. When the models finished training, they were saved within a results folder named after the set ID housing the 'Brain.onnx' file containing all the necessary data for our model.

5 Testing, Results & Evaluation

Tweaking and testing of the hyperparameters was conducted and the following are the results when comparing IL and PPO algorithms. A balance had to be established between using hyperparameters to maximise the cumulative reward of a singular algorithm and having a number of hyperparameters of the same values to have a fair comparison. The following are the results of the agents trained with two different approaches, Imitation Learning (IL) against Proximal Policy Optimization (PPO). Due to limited time, these are the best hyperparameters that could be found, as training these agents took a lot of time each time a single parameter was changed. Given more time possibly better hyperparameters would have been discovered to achieve even better results.

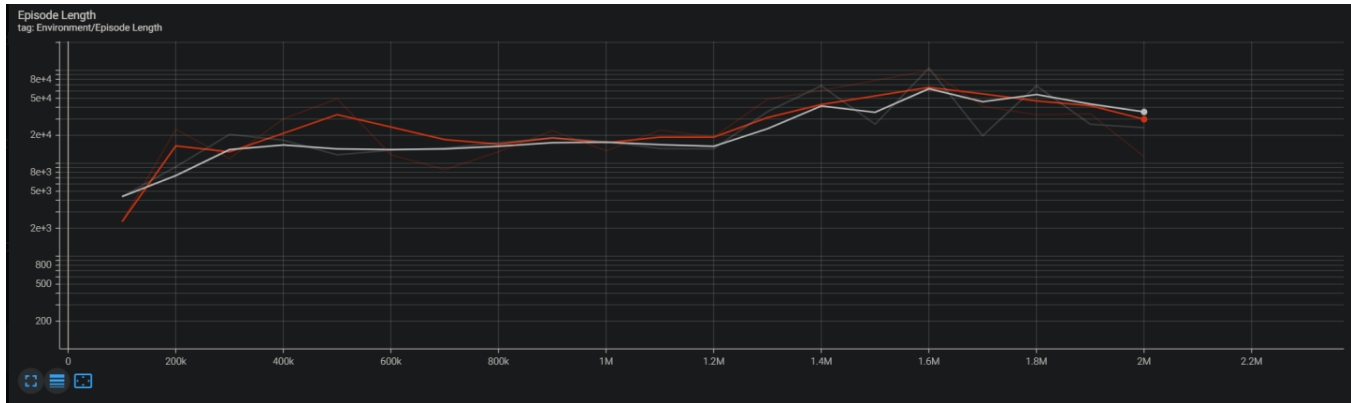


Figure 9: A graph comparing the episode lengths of the two learning techniques

The learning algorithms use artificial intelligence mathematical models to find the best path whilst maximising the reward. A number of hyperparameters effect the accuracy of the results and note that the results will never guarantee the best outcome since after all it is estimating the best path to take.

The first hyperparameter that had to be decided on was the episode length. Picking shorter episodes would bias the learning algorithms to favour earlier actions performed by the agent. Example of such bias is clearly shown in the behaviour of the resulting brain, the brain inferred that the jigging of the fish was found to have more beneficial results in shorter episodes. On the other hand, choosing lengthier episodes might influence the agent to learn specific strategies that work for only a limited number of episodes, and hence, hinders the process of finding the approximation of the optimal target policy. When episode length was set to be substantially large, the brain learnt that picking a side, that is, picking the top or bottom of the screen was a good strategy to adopt. This is because since the fish would still be alive, it would have the opportunity to continue catching fish in the close proximity of the side that it chose. After testing multiple lengths, it was found that for the optimal results to be achieved, an episode length of two million steps would be ideal to train both of the models, as this ensured that the length is not too short nor too lengthy. Additionally, by both algorithms having the same episode length, this allowed for a fairer baseline to be held when comparing both models.

Another hyperparameter that needed a lot of investigation and tweaking in search for the optimal results, was the extrinsic value. Firstly, it is crucial the relationship between intrinsic and extrinsic values. Intrinsic and extrinsic values are inversely proportional, if one increases the other one decreases. It was sighted that the agents were performing overall better when there was a balance between extrinsic values, or better known as policy driven, and intrinsic values. Depending on which is bigger, determines the methodology adopted by the agent for tackling the same problem. These mappings prove to be the main difference between PPO and IL, the extrinsic value directly determined the whole learning process of the decision making of these reinforcement learning algorithms. The environment was initial programmed to reward good behaviour, such as eating a larger fish, and heavily penalized bad decisions like colliding with trash. IL takes a greedy approach in terms trying to maximise the intrinsic rewards, that is, eating the most fish possible even if it dies in the process. PPO takes a more logical approach in terms of prioritising the shark to stay alive rather than risking colliding with trash, this method produces larger extrinsic values, this can be observed in [figure 2]. A problem present in both algorithms was having large extrinsic values, due to the large extrinsic values the shark would completely disregard huge fish, with great reward, a short distance

away from trash as it took a safer approach and contributed way too heavily to adopting safer playing styles, as the agents would prefer to stay alive rather than risking to chase a fish and increasing the possibility of dying in the process. To combat this bias, several techniques were used, and their main objective was for the intrinsic values for both algorithms to be increased and inherit decreased extrinsic values due to the inverse proportionality relationship between extrinsic and intrinsic values. In addition to adjusting hyperparameters for both methods, for the IL model, the simple fact that the agent randomly copies actions like the ones available in the database provided, inherently increased the intrinsic value, this is because following the actions like the recorded games that were played gave a small reward. For a strict Proximal Policy Optimization algorithm, a similar concept could not be used as it does not make use of imitation learning concepts. To encourage the agent to take risks and combat this bias, the rewards for successfully eating a fish was greatly increased whilst colliding with trash was slightly decreased. This is why in [figure 2], when comparing the extrinsic rewards between the IL and PPO, one can notice that PPO has a much higher extrinsic value. Though both methods started with relatively small extrinsic values, they both converged to having almost similar extrinsic values, yet that small variation was enough to provide a large difference in the approach chosen.

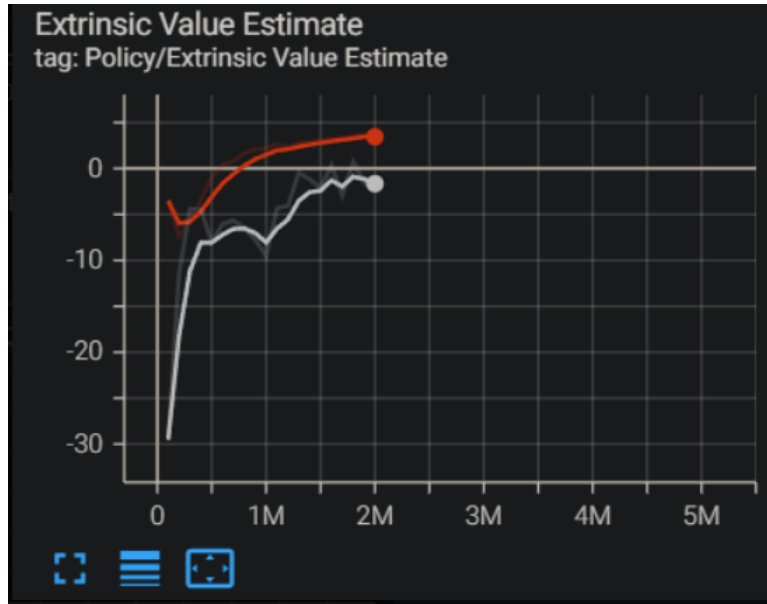


Figure 10: A graph comparing the extrinsic value estimate of the two learning techniques

Policy loss describes the maximum reward that could have been gained versus the actual reward gain by said policy at timestep t , where t represents the current timestep. The lower the policy loss mapping, the better. If the policy loss mapping would result in a zero, that means that the agent has learnt the algorithm to maximise the reward completely. For this task, receiving such output would be highly unlikely due to; there is not a function that can guarantee this reward and such rewards would be due to overfitting, which was handled very cautiously with the hyperparameters chosen. The margin of performance is quite small and almost negligible, but there are some key differences. Throughout the learning for PPO, started with a relatively small policy loss and oscillated marginally to stabilise at around 0.0975 at the end of training, very close to zero, indicating that the agent managed to learn how to take the actions that would reap a substantial reward from the current available reward. The minimal point for the policy loss graph, can be found at 0.095 and is achieved by the PPO algorithm. The IL algorithm started training by having the largest loss, and throughout training produced a higher frequency of oscillations with smaller amplitudes, this helped it to converge at a smaller value and faster rate relative to the PPO methodology.

Policy loss describes the maximum reward that could have been gained versus the actual reward gain by said policy at timestep t , where t represents the current timestep. The lower the policy loss mapping, the better. If the policy loss mapping would result in a zero, that means that the agent has learnt the algorithm to maximise the reward completely. For this task, receiving such output would be highly unlikely due to; there is not a function that can guarantee this reward and such rewards would be due to overfitting, which was handled very cautiously with the hyperparameters chosen. The margin of performance is quite small and almost negligible, but there are some key differences. Throughout the learning for PPO, started with a relatively small policy loss and oscillated

marginally to stabilise at around 0.0975 at the end of training, very close to zero, indicating that the agent managed to learn how to take the actions that would reap a substantial reward from the current available reward. The minimal point for the policy loss graph, can be found at 0.095 and is achieved by the PPO algorithm. The IL algorithm started training by having the largest loss, and throughout training produced a higher frequency of oscillations with smaller amplitudes, this helped it to converge at a smaller value and faster rate relative to the PPO methodology.

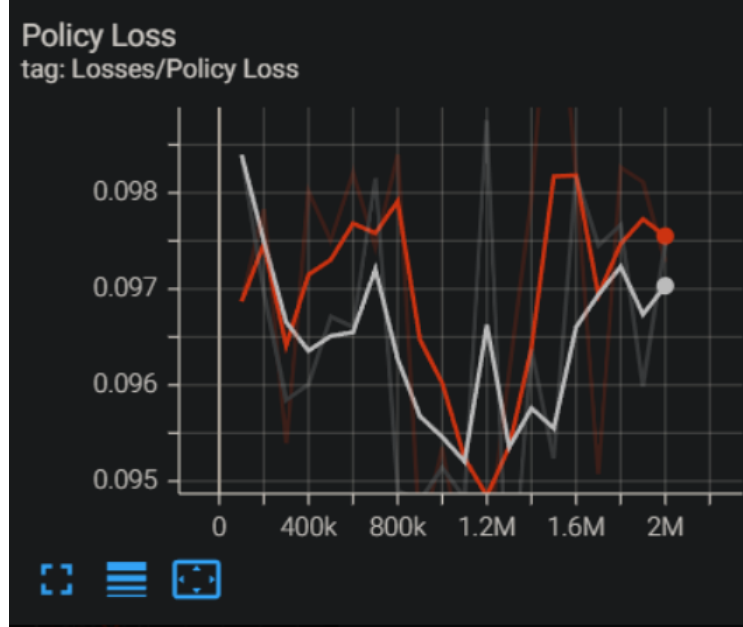


Figure 11: A graph comparing the policy losses of the two learning techniques

An interesting result to compare algorithms is entropy. Entropy describes how much the agent is predictable in choosing its next action, having a high entropy indicates that the agent's actions are chosen at a less predictable rate, whilst having a low entropy indicates that the agent's next action can be almost assumed. Theoretically, for the problem at hand did not matter that how high or low the entropy for each agent is as neither agent is competing against another user or agent. In practice entropy still played a significant impact on the results. It is interesting to note how by randomly choosing an action based on some probability to either follow its policy or to replicate an action given in its database increases the entropy.

Cumulative reward graph demonstrates the sum of all rewards until the current timestep t , essentially the higher the cumulative reward the better is the performance (P) of the algorithm. In formal terms: $G \propto P$. Let G_{IL} denote the cumulative reward for the imitation learning algorithm, and G_{PPO} denote the cumulative reward for the PPO methodology. Analysing figure five, one can observe, that G_{IL} started the training process at a higher value than G_{PPO} . The previous is true as PPO works by randomly choosing which actions to undergo, and IL, whilst is build on a similar concept, has a probability of copying the actions taken from the provided database filled with recorded games played by a human. Therefore, the IL agent did not try to learn to stick to a side of the screen to avoid the obstacles whilst the agent with PPO algorithm did. This resulted in G_{IL} having an initial higher value than G_{PPO} . Furthermore, the PPO algorithm further declined in performance for the first few iterations due the reason mentioned above.

Considering the middle portion of the Cumulative Reward graph. G_{IL} experiences a steady increase in the rewards gained and controllably oscillates to finally converge to a reward around -300. This means that the agent eats the majority of the fish, but still misses quite a bit since there is a negative reward allocated to not eating fish. Although the shaky start of the G_{PPO} , it makes a great comeback and learns at a steady rate to maximise the reward and eats even more fish than the IL algorithm. After approximately 1.7 million steps the PPO agent, for some reason, started missing a lot of fish since the graph shows a drastic change in the gradient. Yet this was still enough to outperform the G_{IL} .

It is interesting to note the way both agents prepare to eat fish, PPO trained agent prefers to eat the fish by first getting in the same y-axis as the fish, this is a safer approach, that in reality maximised cumulative reward (G).

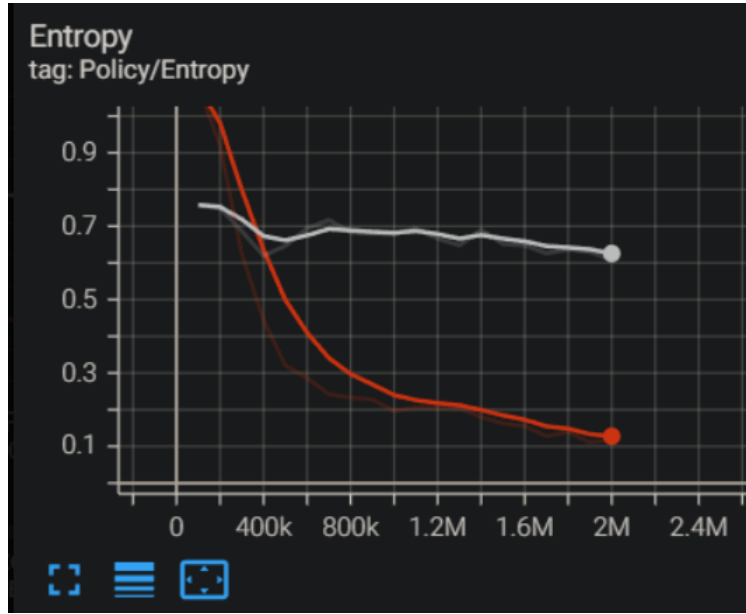


Figure 12: A graph comparing the entropy of the two learning techniques

Whilst the IL agent finds no problem eating fish by approaching them diagonally either from the bottom or the top of said fish. This was done on purpose by the allocated set of recordings to allow to further maximise the reward by replicating the user's playing style, and therefore IL is riskier than PPO in nature. The unpredictability of the IL algorithm, in the long run, would be of self-detriment. As due to this increase in randomness, it would surely make the agent collide with trash before the PPO trained agent. So, the IL methodology is motivated to eat the most fish possible before it dies, while the PPO method prefers to keep the shark alive irrespective of how many fish it misses.

Two videos of two and a half minutes of video footage were manually analysed of both agents playing and the following table was constructed.

Algorithms	Fish Eaten	Missed Fish	Total Fish	Missed Fish (%)
PPO	53	12	65	18.5
IL	61	11	72	15.3

Figure 13: PPO versus IL video analysis results

The Missed Fish percentage(%) was produced by dividing the missed fish by the fish eaten added to the missed fish to achieve the total number of fish that was available to be eaten. In mathematical terms, the following formulas were used for both algorithms:

$$TotalFish = MissedFish + FishEaten$$

$$MissedFish(\%) = Round(MissedFish / TotalFish * 100)$$

By examining the table, one can determine that in practice, even though the final G_{IL} is lower than the final G_{PPO} , the IL algorithm had a lesser percentile of missed fish when compared to the PPO algorithm. During execution

the fish added for both algorithms were in a particular configuration to motivate correct pathing. The difference in the total number of fish might have added a small bias towards the IL algorithm, since the more fish you have on screen the higher probability for the missed fish percentage to normalise at a lower value. On the whole both methodologies performed quite well in achieving the maximum number of fish and finding the optimal path. In conclusion one can say that if the optimal policy is π_o , and policies for IL and PPO are π_{IL} , π_{PPO} respectively, then mathematically: $\pi_{IL} \approx \pi_{PPO} \approx \pi_o$.

6 Distribution of Work

The distribution of the work on this GAPT is as follows:

Evangeline Azzopardi:

- Implementation of UI and Audio.
- Writing of: Introduction, The Challenge & Approach chosen and the Literature Review.
- Compilation of Documentation/Report.
- Editing and Compilation of the video.

Kieron Sultana:

- Training of Imitation Learning.
- Writing of: Testing, Results & Evaluation.
- Recording of gameplay for the video.

Kian Parnis:

- Team Leader.
- Setup and Implementation of the game environment.
- Implementation of Reinforcement Learning.
- Writing of: Explanation of the Implementation and the Script for the video.

7 Bibliography

References

- [1] Adi Botea, Bruno Bouzy, et al. *Pathfinding in Games*
<https://drops.dagstuhl.de/opus/volltexte/2013/4333/>
- [2] Xiao Cui, Hao Shi *A*-based Pathfinding in Modern Computer Games*
https://www.researchgate.net/publication/267809499_A*-basedPathfindinginModernComputerGames
- [3] Hongda Qui, *Multi-agent navigation based on deep reinforcement learning and traditional pathfinding algorithm*
<https://arxiv.org/abs/2012.09134>
- [4] Ross Graham, Hugh McCabe, Stephen Sheridan *Neural Networks for Real-time Pathfinding in Computer Games*
https://www.researchgate.net/publication/254184076_NeuralNetworksforReal-timePathfindinginComputerGames
- [5] Youssef S. G. Nashed, Darryl N. Davis, *FUZZY Q-LEARNING FOR FIRST PERSON SHOOTERS*
https://www.researchgate.net/publication/290060115_FuzzyQ-Learningforfirstpersonshooters
- [6] Wikipedia, *Q-Learning*
<https://en.wikipedia.org/wiki/Q-learning>
- [7] Tycho van der Ouderaa, *Deep Reinforcement Learning in Pac-man*
https://moodle.umons.ac.be/pluginfile.php/404484/mod_folder/content/0/Pacman_DQN.pdf
- [8] Christian Thureau, Gerhard Sagerer, Christian Bauckhage *Imitation Learning at All Levels of Game AI*
https://www.researchgate.net/publication/228474437_ImitationLearningatAllLevelsOfGameAI
- [9] Zoltan Lorincz *A brief overview of Imitation Learning*
<https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c>
- [10] Faraz Torabi, Garret Warnell, Peter Stone *Behavioral Cloning from Observation*
<https://arxiv.org/abs/1805.01954>
- [11] John Schulman, Filip Wolski, et al *Proximal Policy Optimization Algorithms*
<https://arxiv.org/abs/1707.06347>
- [12] GeeksforGeeks *A Brief Introduction to Proximal Policy Optimization*
[https://www.geeksforgeeks.org/a-brief-introduction-to-proximal-policy-optimization/#:~:text=Proximal%20Policy%20Optimisation%20\(PP0%20is,it%20was%20implemented%20by%20OpenAI](https://www.geeksforgeeks.org/a-brief-introduction-to-proximal-policy-optimization/#:~:text=Proximal%20Policy%20Optimisation%20(PP0%20is,it%20was%20implemented%20by%20OpenAI)
- [13] Alexandre Gonfalonieri *Generative Adversarial Imitation Learning: Advantages & Limits*
<https://towardsdatascience.com/generative-adversarial-imitation-learning-advantages-limits-7c87fc67e42d>
- [14] Jonathan Ho, Stefano Ermon *Generative Adversarial Imitation Learning*
<https://arxiv.org/abs/1606.03476>
- [15] Unity-Technologies *GitHub - ml-agents*
<https://github.com/Unity-Technologies/ml-agents>
- [16] Dr Josef Bajada *Function Approximation Slides, University of Malta - VLE*