

College of Business, Technology and Engineering

**Department of Computing
Project (Technical Computing)
[55-604708]
2020/21**

Author:	Mohammed Mahmoud
Student ID:	27010929 / b7010929
Year Submitted:	2021
Supervisor:	Alessandro di Nuovo
Second Marker:	Yomi Otebolaku
Degree Course:	Computer Science for Games
Title of Project:	Developing smart A.I through machine learning techniques and applying them to game-like scenarios.

Confidentiality Required?

YES / NO

NO

I give permission to make my project report, video and deliverable accessible to staff and students on the Project (Technical Computing) module at Sheffield Hallam University.

YES / NO

YES

1 – Introduction	4
1.1 – Aims and Objectives.....	4
1.1.1 – Machine Learning in Games.....	4
1.1.2 – MLAgents	4
1.2 – Project Outline	4
2 – Research.....	5
2.1 – Unity & MLAgents	5
2.2 – Machine Learning.....	5
2.2.1 – Reinforcement Learning.....	5
2.2.2 – Neural Networks	6
2.2.3 – PPO	7
2.2.4 – SAC	8
2.2.5 – Chosen Algorithm.....	8
2.2 - Designing Agents.....	9
2.3 – Designing Reward Systems	9
2.4 -Training Environments.....	9
2.5 - Similar Products	9
3 – Design.....	11
3.1 – Unity, C# & Visual Studio	11
3.2 – MLAgents	11
3.3 – Anaconda3	11
3.4 – Tensor board	11
3.5 – User Interface.....	11
3.6 – Required Hardware	11
4 – Deliverable	12
4.1 – Practicality.....	12
4.2 – Creating Agents.....	12
4.2.1 – The Car	12
4.2.2 – Agent Scripts	12
4.2.3 – Agent Vision	13
4.2.4 – Rewards.....	14
4.3 – Simple Task.....	14
4.4 – Configuration Files and Parameter Tuning	14
4.5 - Simple Driving Task.....	15
4.6 - Intermediatory Driving Task	16
4.7 - Complex Driving Task.....	17

4.8 - Adaptability.....	19
4.9 – Training Process	19
4.10 – Final Design	20
4.11 – Difficulties	20
5 – Testing.....	21
5.1 – Hyperparameter Testing	21
5.2 – Heuristics.....	21
5.3 – Reward Testing.....	21
5.4 – Final Results	22
6 – Critical Evaluation & Reflection.....	28
6.1 – Personal Reflection	28
6.2 – Development Reflection	28
6.3 – Testing Reflection.....	29
6.4 – Ethics	29
6.5 – Were the Project Aims Met?.....	29
6.6 – Future Developments	30
Bibliography	31
PROJECT SPECIFICATION - Project (Technical Computing) 2020/21	38

1 – Introduction

Artificial intelligence has continued to evolve over time as new ideas, algorithms and methods are being developed and refined. The majority of newly developed software incorporate various artificial intelligence techniques and games are one of the more common applications to do so. As hardware performance rises, the ability to use artificial intelligence also increases as the computational time decreases. Algorithms are also being continuously researched to look for innovative and practical improvements when it comes to processing speeds and memory management.

Machine learning has become much more popular in recent years due to higher GPU and CPU performance but also because of the vast amount of data that is available to learn from. Within the gaming industry, more experimental A.I. techniques like machine learning are rarely used and are often neglected in preference for hand crafted A.I. behaviours.

The Idea and main focus of this project was initially crafted from experiments on how machine learning could work within a game centred environment. This project investigates where and when machine learning can be used and highlights the advantages of using said techniques over more traditional A.I. methods in games.

1.1 – Aims and Objectives

The following Aims and Objectives have been researched and developed to successfully complete this project.

1.1.1 – Machine Learning in Games

Multiple machine learning algorithms were researched to see which method provides the most accurate and optimal result. This project focuses on reinforcement learning and briefly compares Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC).

1.1.2 – MLAgents

The ML-Agents Toolkit is an open-source project that allows developers and researchers to use environments within the Unity game engine to train intelligent agents. It uses current algorithms to aid the user in development and allows the user to focus on designing proper reward systems, training environments and behaviour tuning. The toolkit will be used to create an intelligent agent that is able to successfully race around any track made out of smaller pieces. The user will be allowed to build a race track and will be able to see the agent race around it to showcase the adaptability of machine learning.

1.2 – Project Outline

To achieve the desired results and ultimately succeed with this project, there are a few key stages that need to be completed.

1. Research Phase
2. Design Phase
3. Development Phase
4. Testing Phase
5. Evaluation / Reflection Phase

2 – Research

The following section will highlight research material that was gathered to aid with the development of the project.

2.1 – Unity & MLAgents

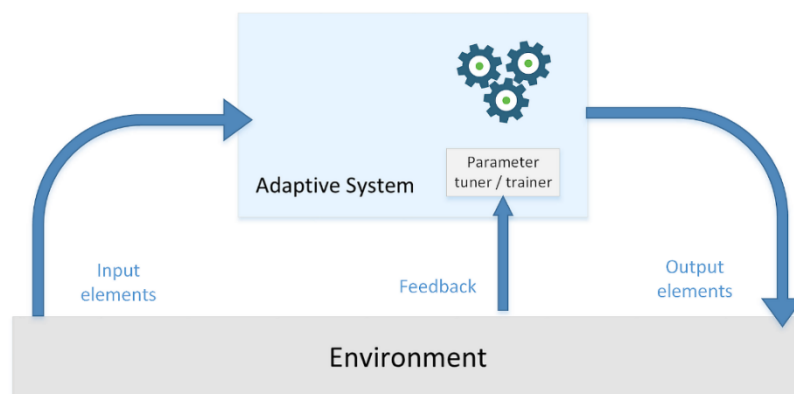
UnityML was released a few years ago, and slowly grew to encompass a range of features that enable a game engine to serve as simulation environments for training and exploring intelligent agents and other machine learning applications (Buttfield-Addison et al., 2021).

The primary benefit of using MLAgents is that it separates the machine learning from standard game development in some ways. Whilst the developer must create an environment designed around the agent, there is no machine learning implementation required. The machine learning algorithms and simulations are handled by a background python API which can be configured by the user. This is extremely useful as the field of machine learning is vast and requires lots of prior knowledge on complicated maths and algorithmic theory. However, to fully understand what is happening behind the scenes and use the package effectively, a base level of machine learning, reinforcement learning and neural networks research is required.

2.2 – Machine Learning

Machine learning focuses on applications that learn from experience and improve their decision-making or predictive accuracy over time (IBM Cloud Education, 2020a).

Within machine learning, the algorithms use large amounts of data to find patterns in order to make accurate predictions with any newly given data. Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones (Marsland, 2015).



(Bonaccorso, 2017)

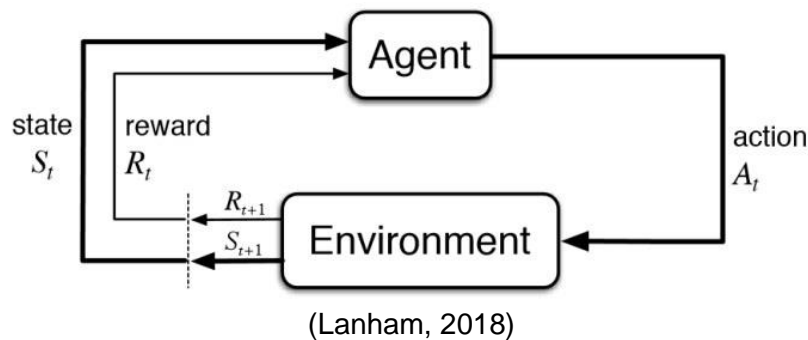
For the simplicity of the project and research, the focus will primarily be on Reinforcement learning, Deep learning and Neural networks. With Unity Machine Learning Agents (ML-Agents), you are no longer “coding” emergent behaviours, but rather teaching intelligent agents to “learn” through a combination of deep reinforcement learning and imitation learning (Unity, 2018).

2.2.1 – Reinforcement Learning

Unlike many other machine learning models, reinforcement learning does not use a data sample as input but rather learns on the go and tries to make the best decision based on the immediate reward. Through trial and error, a reinforcement learning model will understand

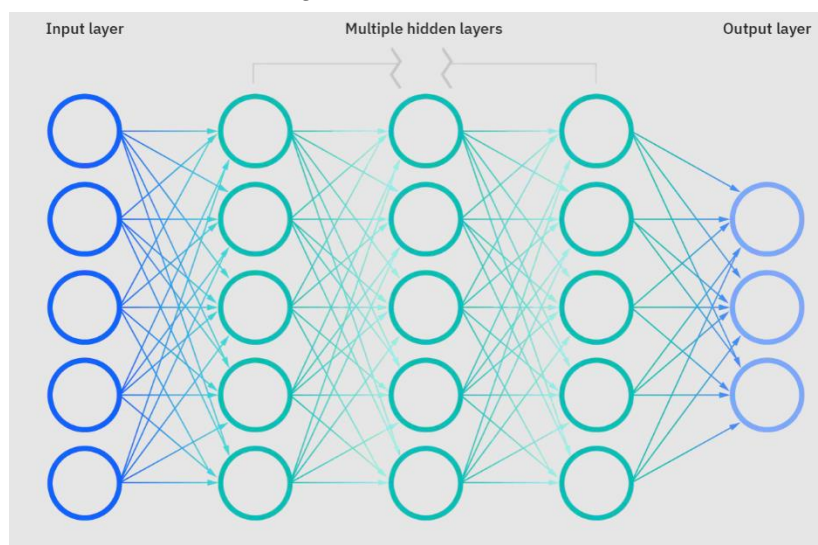
what actions are beneficial based on the rewards it receives. It is important to mention that the behaviour of an agent is not implicitly told to take certain actions but rather learns what action is the best to take at any given time.

Reinforcement learning is particularly efficient when the environment is not completely deterministic, when it's often very dynamic, and when it's impossible to have a precise error measure (Bonaccorso, 2017).



2.2.2 – Neural Networks

Neural Networks are computational structures that are defined by a layer of network calculations. They consist of an input layer, one or more hidden layers and an output layer. In the diagram presented, each input in the input layer is a neuron which essentially holds a number between 0 and 1. Each neuron is connected to each neuron in the hidden layers and each neuron has an associated weight and bias.



(IBM Cloud Education, 2020b)

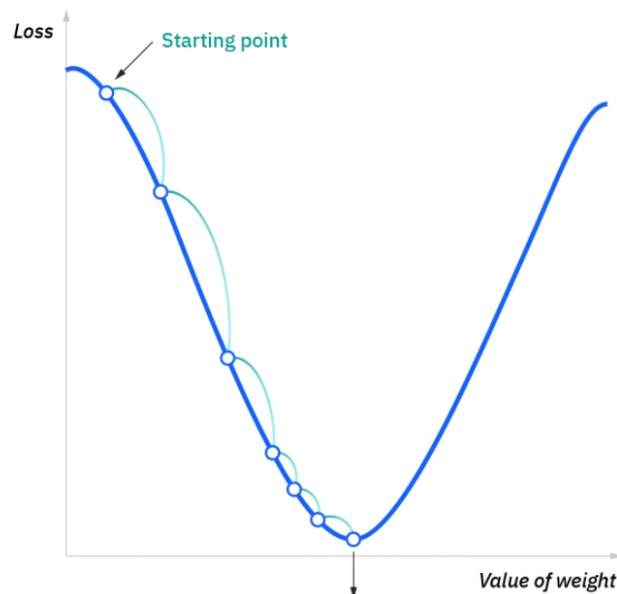
These weights help determine the importance of any given variable, with larger ones contributing more significantly to the output compared to other inputs. All inputs are then multiplied by their respective weights and then summed. Afterward, the output is passed through an activation function, which determines the output (IBM Cloud Education, 2020b).

Each node in the hidden layer has an output and if the output exceeds a given threshold, the node is “activated”. This means that the node is fed into the next layer of the network and used as further input.

To find out if a neuron is activated you have to get the weighted sum for all of the previous layer's neurons that are attached to the current neuron.

A cost function is used to evaluate the overall accuracy of the model, to check if it has produced the desired output based on the given inputs. The cost is the difference between the actual results and the expected results. Minimising the cost essentially means improving the accuracy of the model and the process of doing so is called gradient descent. The goal of gradient descent is to reach a local minimum where the cost function is the smallest.

(IBM Cloud Education, 2020b)



The cost changes when the value of the weights for each node change as they in turn effect which nodes are activated which alters the final output. In simple terms, for a network to learn it essentially means to minimise the cost function.

2.2.3 – PPO

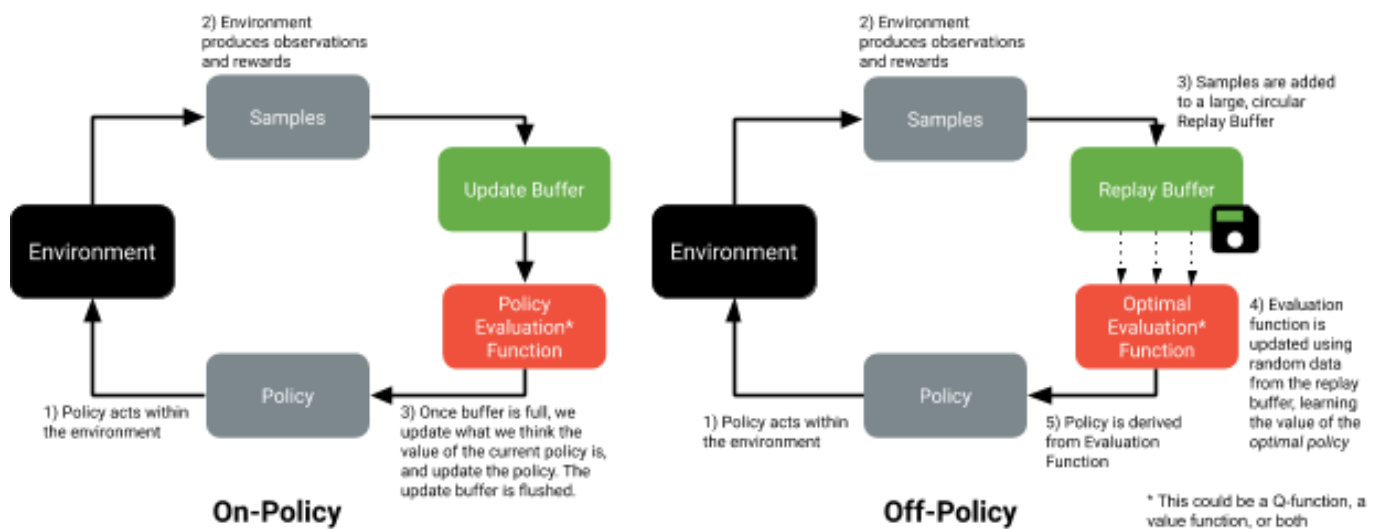
ML-Agents uses a reinforcement learning technique called Proximal Policy Optimization (PPO). PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state (Pierre, 2018).

The route to success in reinforcement learning isn't as obvious — the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small (Schulman et al., 2017).

Using probability, the algorithm pushes the model to take actions that resulted in a positive reward by increasing the probability of taking the same action when encountering the same state.

2.2.4 – SAC

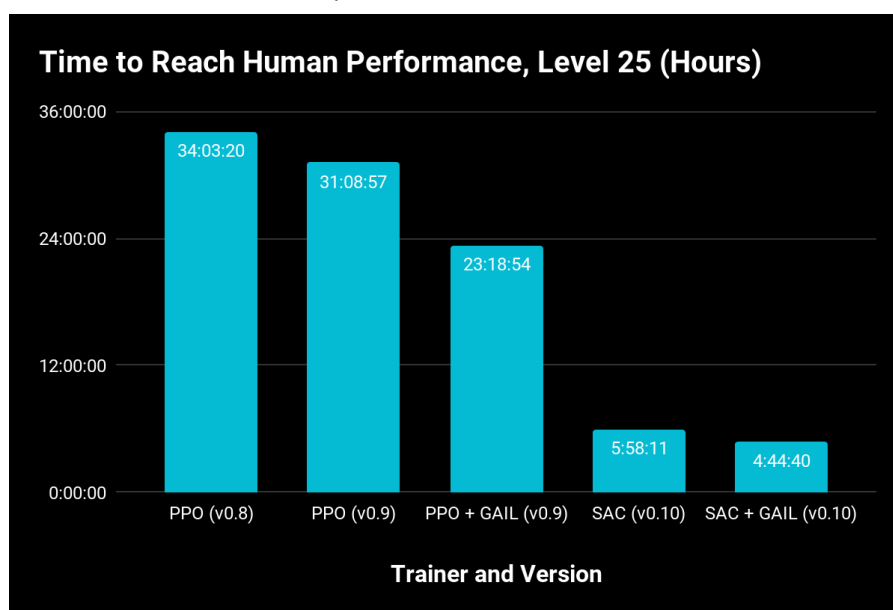
MLAgents also allows the use of another reinforcement learning algorithm called Soft Actor Critic (SAC). One of the critical features of SAC, which was originally created to learn on real robots, is sample-efficiency. For games, this means we don't need to run the games as long to learn a good policy (Teng, 2019). This is because SAC is an off-policy model which differs from PPO (an on-policy model). Rather than learning how good the current policy is, off-policy algorithms learn this optimal evaluation function across all policies. This is a harder learning problem than in the on-policy case—the real function could be very complex. But because you're learning a global function, you can use all the samples that you've collected from the beginning of time to help learn your evaluator, making off-policy algorithms much more sample-efficient than on-policy ones (Teng, 2019).



(Teng, 2019)

2.2.5 – Chosen Algorithm

Comparing the two research algorithms (PPO and SAC) it is quite clear that they differ in approach, implementation and both effect the training process drastically. PPO is an on-policy algorithm making it much slower than SAC as it is less sample efficient.



(Teng, 2019)

As shown in the diagram above, the time for the agent to reach human performance with PPO alone is at least 5 times slower than using SAC alone. However, the chosen algorithm for this project will be the PPO algorithm for the following reasons. First, Unity states that PPO is the most stable and reliable algorithm for the majority of cases. The next reason is that PPO is the industry standard approach for reinforcement learning (especially in games) and therefore there is much more documentation and reliable sources available. The final reason is the current understanding of PPO is much higher (because of the previously mentioned documentation) than SAC and so the development overall should be smoother.

2.2 - Designing Agents

Agents represent the actors that we are training to learn to perform some task or set of task-based commands on some reward (Lanham, 2018). Agents operate based on the received rewards and so, the agent must be able to earn said rewards to work properly. An agent has to be able to perform some actions that result in either a positive or negative reward for it to learn. The actions in this case would be to drive forward, steer or even to do nothing. However, just trying these actions randomly will take a lot of training to find the perfect sequence for a given task and will lead to potential overfitting. To fix this issue we have to give the agent vision, a way for an agent to see its environment. Adding vision to agents in Unity is done with a ray perception sensor 3d component which allows an agent to understand its surroundings through sensor collisions.

There are also many required scripts that are provided with the package that are necessary for an agent to work properly (more on this in the section 4).

2.3 – Designing Reward Systems

Designing a proper reward system is essential to the agent's ability to learn over time. The agent will prioritise the highest rewards so it's common to give larger rewards for the completion of a task and smaller rewards for the steps the agent needs to take prior to that. For example, an agent may receive a reward of 1 for completing a lap in a circuit but will receive 0.01 (or another small value) for crossing a checkpoint. Often an agent needs to wait a long time to observe the reward. Algorithmic tricks can help here, but fundamentally, altering the reward signal so it provides more frequent updates helps guide the agent toward the solution (Winder, 2020). Rewards given are highly dependent on the agent, environment, and the difficulty of the task so it is impossible to have a one-fits-all approach however, following basic guidelines can help to avoid issues with rewards. Reward issues can happen when you attempt to set the rewards too high or low, or when the opportunity for a reward is rare or sparse (Lanham, 2018).

2.4 -Training Environments

The environment created for an agent to use as a simulation is one of the most important elements when it comes to reinforcement learning. The environment dictates when and how the agents receives rewards and also allows the actual task to be completed. It eliminates the need to... undertake the tedious data labelling often required by Supervised Training. Instead, agents are modelled in the environment and receive rewards based on their actions (Lanham, 2018).

2.5 - Similar Products

One project that is currently being developed is OpenAIFive which is a team of 5 neural networks that have learnt to play Dota 2. Dota 2 is one of the most complex games as there are many characters, unique abilities and top level strategies. They aim to train the team of A.I to a level that it can compete at the professional level. This is all done with the PPO

algorithm and shows off its true capabilities. OpenAI Five plays 180 years worth of games against itself every day, learning via self-play. ...Using a separate LSTM for each hero and no human data, it learns recognizable strategies. This indicates that reinforcement learning can yield long-term planning with large but achievable scale — without fundamental advances, contrary to our own expectations upon starting the project (Brockman et al., 2018).

Another example is in the game Planetary Annihilation where the default A.I. is created through machine learning techniques. "Neural networks allow me to create a more dynamic AI that will behave based on what it learned rather than specifically what a designer wants. Much like in multiplayer, the AI in Galactic War [the game's single-player campaign] has a ton of flexibility to dynamically react to players and their strategic decisions. The neural networks are why." – Mike Robbins (Xav De, 2014)

Whilst both of these examples are much more complex than the project aims, these projects were highlighted to show off the power of machine learning and its viability in games.

3 – Design

This section will briefly highlight the software, plugins and programming languages used to develop the project.

3.1 – Unity, C# & Visual Studio

Using Unity was a given from the initial idea as the MLAgents package is a Unity plugin and allows the developer to use Unity environments for machine learning.

The scripts and functions provided with the package are written in C# and because of that it is the programming language that will be used to produce the deliverable.

Visual Studio is the chosen IDE because of its simplicity, ease of integration and debugging within Unity as well as the numerous source control options that are available.

This all works out well as both the Unity game engine and C# are very familiar. This will help development move along quicker as the tools Unity provides can be properly used from experience with previous works. Additionally, Unity has really fast build times compared to other game engines like Unreal and with a project that is very time consuming, small time bonuses can really add up.

3.2 – MLAgents

MLAgents is the primary package that allows the completion of this project as it opens up the complex field of machine learning to less experienced developers. Without it the project would be infeasible with the given time frame and current knowledge on the subject matter. It integrates seamlessly with Unity and setting up environments for training is straight-forward.

3.3 – Anaconda3

Anaconda3 is used to handle the python commands necessary for MLAgents. This includes installing the MLAgents python API, training the agents and outputting the results to tensor board.

3.4 – Tensor board

Tensor board is used to visualize the results of training through generated graphs. This will help tremendously when it comes to training and optimising both the agents' behaviours and the time taken to train the agents. It will also help when comparing what hyper-parameter changes effected the training process and will be used to evaluate if a change was positively impactful.

3.5 – User Interface

The UI for the project will be mostly mouse based with a few keyboard inputs mapped to specific actions. The interface should be clear to new users and should communicate the main features properly.

3.6 – Required Hardware

There is no specific hardware required however the training process is quite intensive so a high-end PC would aid in the development. Ideally two machines would be used simultaneously, one for training and one for project development. The project will be created with only one high-end PC.

4 – Deliverable

This section will focus on the process of creating the deliverable and will touch on development choices as well as the thought process behind them.

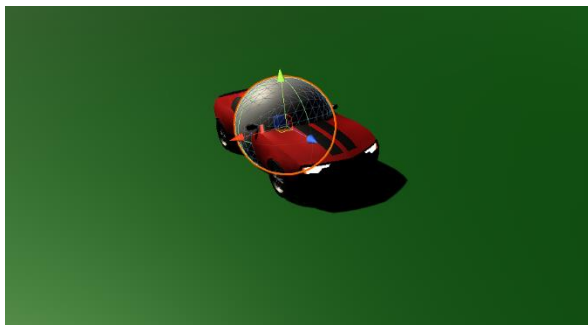
4.1 – Practicality

Machine learning is constantly evolving and its presence within games is still minimalistic. The reasoning for this may be the vast amount of knowledge required to implement such techniques as it is a subject that revolves around complex maths and theory. Because of this, libraries and supporting software like MLAGents are used to allow the developer to focus on an agent's behaviour. With use of this, the deliverable is possible with both the given time frame and current knowledge of the topic at hand.

4.2 – Creating Agents

4.2.1 – The Car

The Agents in this project are the cars that are able to drive around race tracks without being told how to do so. For this to work an Agent has to be set up in the right way to allow it to do the given task. For the car's movement, the car object is split into two objects, the first being the actual visuals for the car and the second being a sphere positioned in the middle of the car. The sphere has a collider and a rigidBody to handle physics and detect collisions. It was important to code the movement scripts in a way that the Agent scripts can access and use to perform the correct actions.



4.2.2 – Agent Scripts

To create an intelligent agent that can learn over time a custom agent script is needed. Creating an agent script follows the normal steps however, unlike a normal unity script instead of inheriting from MonoBehaviour it should be changed to inherit from the Agent base class **Appendix A.1**. This script is a class that comes with the MLAGents package and is what allows the developer to add observations and actions that the agent can take. Agents in an environment operate in steps. At each step, an agent collects observations, passes them to its decision-making policy, and receives an action vector in response (Unity, 2020a).

There are some key functions that are inherited from the agent class and are meant to be overridden so that the developer can implement the desired behaviours. The OnEpisodeBegin () function is called when the agent either times out or has reached an end goal. This is used to reset the episode so that the agent can train in the environment multiple times. The next function used is the CollectObservations (VectorSensor sensor) method which is used to add any observations the agent needs. For the car to race around the track properly it needs some information as to which direction it needs to travel. There are

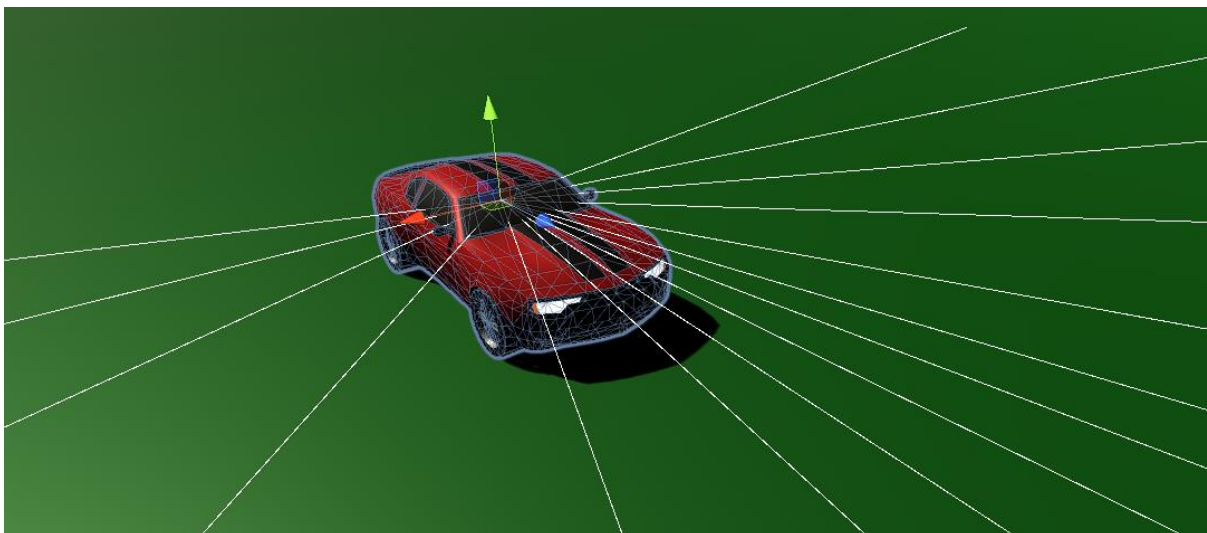
checkpoints around the track and a vector between the agent's current position and the next checkpoints position is added to the observations. The next and most important function is the `OnActionReceived` (`ActionBuffers` actions) method. This function allows the developer to create an action buffer and apply actions to the agents. Here the Agent script calls the acceleration and steering functions in the car controller. The last override method used is the `Heuristic` (in `ActionBuffers` actionsOut) function which allows the developer to self-test through specified inputs **Appendix A.2 – A.5**.

The next important script is the behaviour parameter class. At runtime, this component generates the agent's policy objects according to the settings specified in the Editor (Unity, 2020b). The first is the space size which is under the vector observation section. The space size is set to 3 as the `CollectObservations` (`VectorSensor` sensor) function added a 3-dimensional vector as an observation. Under the vector action section, the space type is set to continuous to allow more accurate values instead of discrete (rounded) values. The space size is also set to 2 as there are two possible actions the agent can take, either accelerate or steer **Appendix A.6**.

The last script is the decision requester class which asks the agent to make a decision every X number of steps. The lower the number, the more decisions will be requested and therefore more actions will be performed allowing for more accurate completion of tasks.

4.2.3 – Agent Vision

Even with the previous set up the agent would still struggle to complete any tasks that require vision. Extra observations can be added through ray casts that act like real life sensors. Adding a ray perception sensor 3d component to the agent can give the agent the necessary vision to complete tasks in a simulated environment. The two most important parts of the component are the detectable tags field and the ray layer mask. These two fields are used to tell the sensor what it can detect and what it should ignore, essentially giving the agent vision **Appendix A.7**.

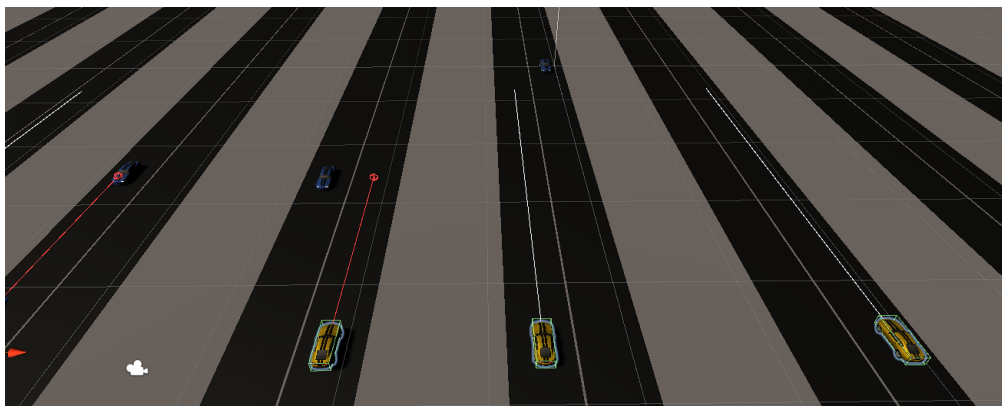


4.2.4 – Rewards

With MLAgents, adding rewards is as simple as calling the AddReward (float reward) function. The parameter is used to pass in a float value which can be either a positive reward or a negative reward. Whilst adding rewards is very simple it has to be thought out as the model may produce unintended behaviours because there are rewards available. For the race track agents, a small reward of 0.01 is given for reaching the next checkpoint and a larger reward is given for reaching the last checkpoint **Appendix A.8 & A.9**. It's important that the agent is only rewarded for reaching the next checkpoint and will not be rewarded for crossing the same one twice. This encourages the agent to follow the track properly to maximize the reward.

4.3 – Simple Task

To learn how MLAgents and the training process actually worked a small experiment was made to see what the package was capable of. A very simple environment was created where a car is positioned on the left of two lanes. Oncoming cars travel down both lanes and the task for the car is to dodge the traffic by switching lanes in time. The car has one sensor pointing in the forward direction to allow the car agent to see if an oncoming car is in the same lane. The car could take two actions, the first being to do nothing and the second is to move to the other lane. The agent was rewarded for each time it made the right decision but was issued a penalty for crashing. The first-time training, there was only one agent and so the training process took some time but was still relatively quick. Once it was confirmed that the training process worked smoothly the environment was duplicated multiple times so that many agents can run their own simulation at once, speeding up the training times drastically. After some time, the mean reward of the agents peaked and the standard deviation was a very low value, essentially meaning that all of the agents were fully optimal.



4.4 – Configuration Files and Parameter Tuning

The configuration file is one of the most important elements when it comes to training and exposes a number of variables that can change how an agent learns drastically. The file is split up into sections with the first being the behaviours which effect how the agent learns over time. The first parameter, trainer type, is set to PPO as that is the default algorithm due to stability and its wide range of applications. Then there are the hyper-parameters like the batch size and buffer size. The batch size dictates the number of experiences in each iteration of gradient descent. Meaning the agent will go through X number of experiences before changing the weights and biases to find a local minimum. The buffer size is usually a multiple of the batch size and defines the number of experiences to collect before updating the policy model. Typically, a larger buffer size corresponds to more stable training updates (Unity, 2021). It was found that setting the values to 512 and 16384 respectively allowed for other hyper-parameter changes and more stability when training **Appendix A.10**.

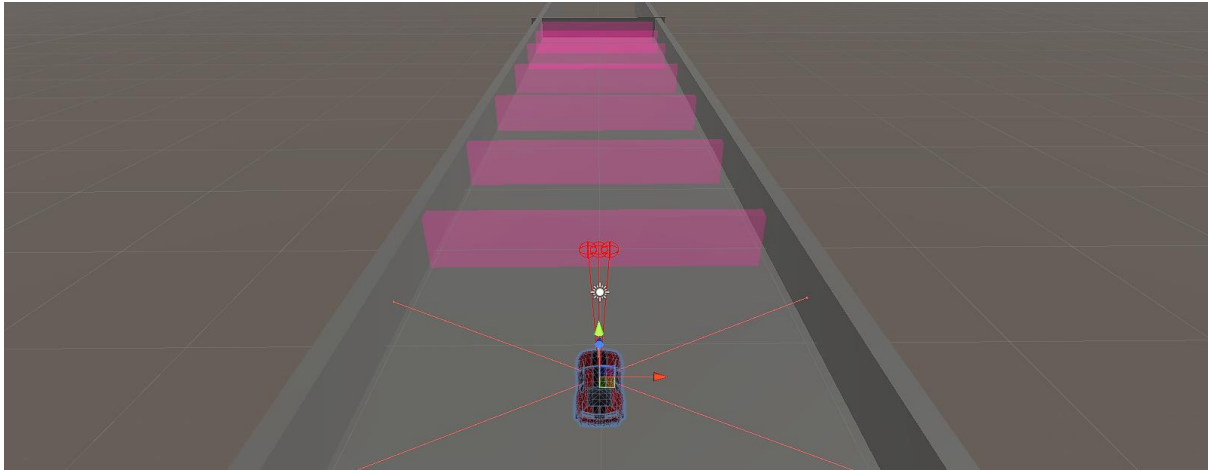
There are also PPO specific hyper-parameters such as beta, epsilon, lambda and number of epochs. The beta parameter is essentially the strength of randomness. This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken (Unity, 2021). As randomness was not an aim and the race tracks are linear in design the beta value was set to 0.001 (default = $5e-3$). Epsilon influences the speed of policy evolution and relates to how much change there can be between the old and new policy. The larger the value the quicker the training process is at the potential cost of less stable updates. The value was left at 0.2 as the training was going at an acceptable pace. The lambda is set to 0.95 (highest recommended value) as the agent should rely on the rewards and environment the developer has set up. Lastly, the num_epoch variable is the number of passes to make through the experience buffer when performing gradient descent optimization (Unity, 2021). Making this value larger can result in faster training as it will reuse the data gathered multiple times instead of gathering new data (it will make more passes through the gathered data). This can come at the cost of stability as reusing the same data can cause overfitting however, with a larger batch size, the experience buffer can hold a more diverse set of data and reusing it is not always bad. As the current environments are static the num_epochs can go up and is set to the value of 8 (default = 3) as it helps with training times massively. Increasing this value was the primary reason for increasing both the batch size and buffer size previously mentioned **Appendix A.11**.

The next section of the configuration file is the network settings which exposes parameters that the neural network (created from training) uses. The first parameter is a Boolean called normalize, this value states whether or not normalization is applied to the vector observation inputs. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems (Unity, 2021). The value is set to true as the agent collects continuous observations when it is travelling through the race tracks. The hidden-units hyper-parameter is the number of units in each hidden layer of the neural network. The value is set to 256 (default = 128) as the environment can be quite complex. The agent has to keep track of its position, the position of the next checkpoint, check if there are any walls or obstacles and process other observations. The number of layers (num_layers) is the number of hidden layers in the neural network. It was found that setting the layer count to 2 worked well for both accuracy and training speed **Appendix A.12**.

The last section is the reward signals which enable settings for extrinsic (environmental based) rewards. Gamma is the first parameter and represents how much an agent in the present will prepare for rewards in the future. The value is set to 0.99 as the agent should prioritise the end goal (finishing the race) rather than the immediate rewards. The Strength parameter determines the rewards multiplication factor and it is set to 1 to allow the developer to have full control of rewards given **Appendix A.13**.

4.5 - Simple Driving Task

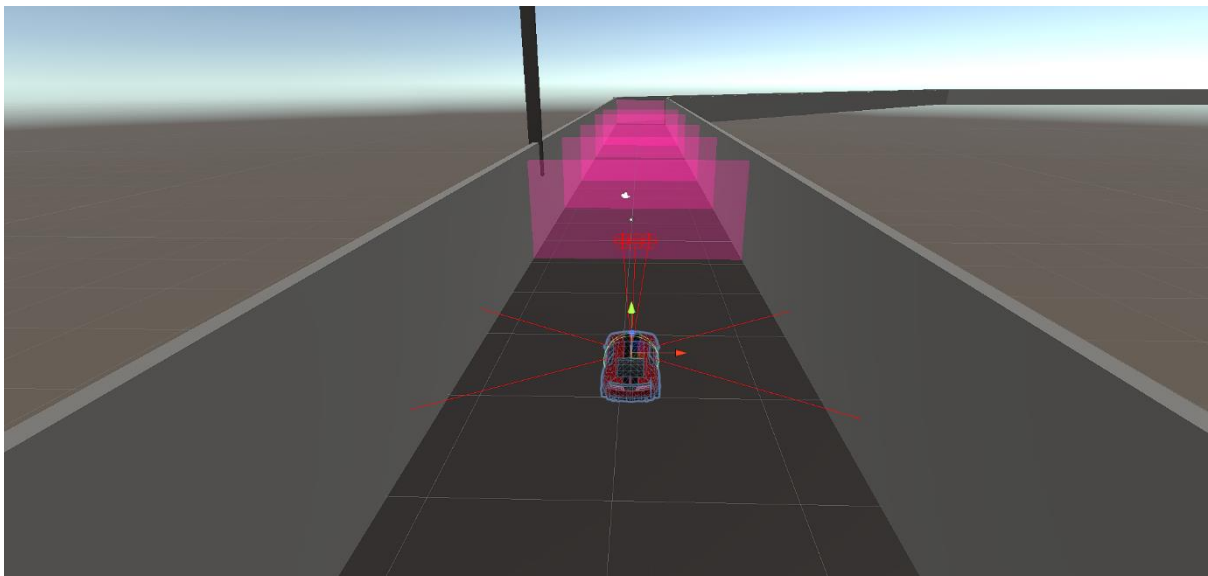
The next iteration of testing was a very simple straight line race track environment to see if the car was properly set up for a neural network to process actions. This time, like mentioned before the car was set up with sensors and could take two actions, accelerate or steer. The previously mentioned checkpoint system and the same related reward system was also used to encourage the agent to complete the track.



The sensors are coloured red to show that they are colliding with something in the environment. The forward pointing sensors are colliding with the checkpoint and the others are hitting the surrounding walls. The training process for this was very quick as it's easy to use multiple agents in the same environment. However, there was some strange behaviour from the agents after training as they wouldn't just drive straight and reach the finish line in the most optimal way. The agents would stray to the left and then go straight which would result in a longer time to complete. This was most likely because there was no penalty for going slowly and no reward for going faster and so once an agent found a set of actions that worked, they decided to stick with that sequence of actions.

4.6 - Intermediary Driving Task

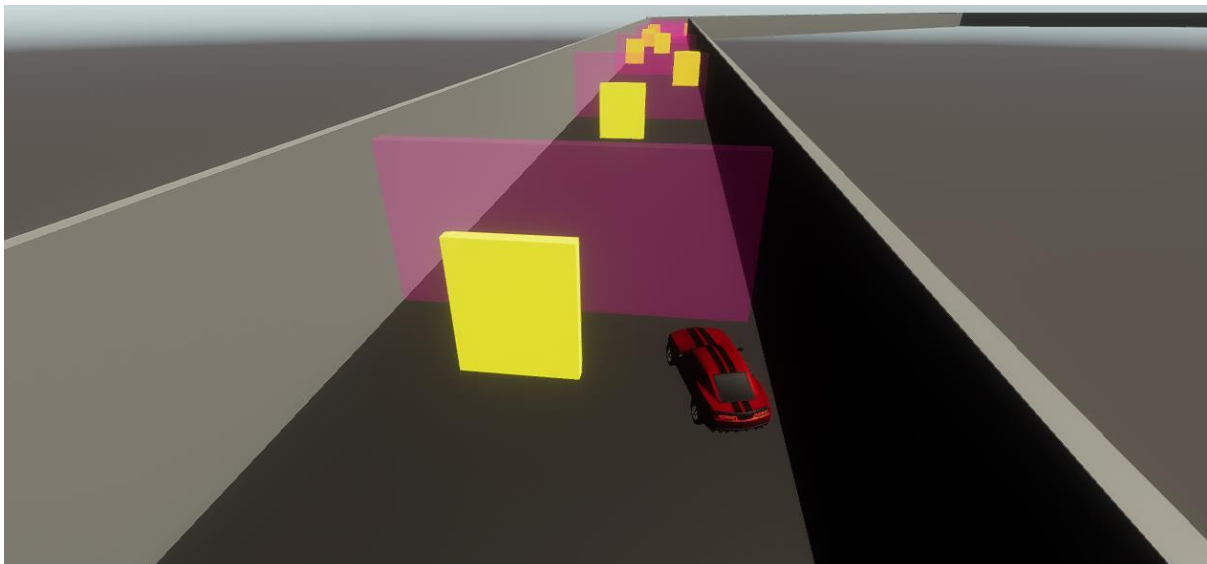
To increase the difficulty and reach a goal similar to the initial vision another track was made for the agent to train on. This time the track had corners and was octagonally shaped.



After more training the agent adapted to the corners and managed to complete the track multiple times without failing. At this stage speed was a focus as the previous agents would move very slowly. To help encourage the agents to complete the tasks faster a small negative penalty was given every few seconds. One issue that this model had was that it did not care if it crashed into walls. The agent learned that the best sequence of actions to complete the track was to crash into one wall and then into a wall on the opposite side. This was an unintended result of not setting up rewards/penalties for hitting walls. To solve this, the walls

were tagged with a wall layer and crashing into a wall would result in a penalty and would reset the episode (starting the track again). This ultimately resulted in an agent that could complete the track without any crashes however, at corners the car would slow down drastically as the checkpoints were blocking the sensors from seeing the walls. This meant that through repetition alone, the agent learnt to slow down at corners, not because it could see the walls but because it has experienced crashing into each corner. This issue was realised and fixed in later iterations of development.

Going forward, the agent was then trained on the same track to avoid partially random obstacles (gold/yellow rectangles) to see how complex of a task the agent could complete. By tagging the obstacles with the wall tag, the agents would also be reset on contact and eventually learned to avoid the obstacles with the wall tag. This however made the car perform much worse in terms of lap times as the car was constantly checking for obstacles. It was a poorly set-up environment as some obstacles were hidden behind checkpoints and so the sensors couldn't pick up on them. Eventually the agents experienced all of the possible obstacle locations and learned a sequence of actions to avoid them **Appendix A.14**.

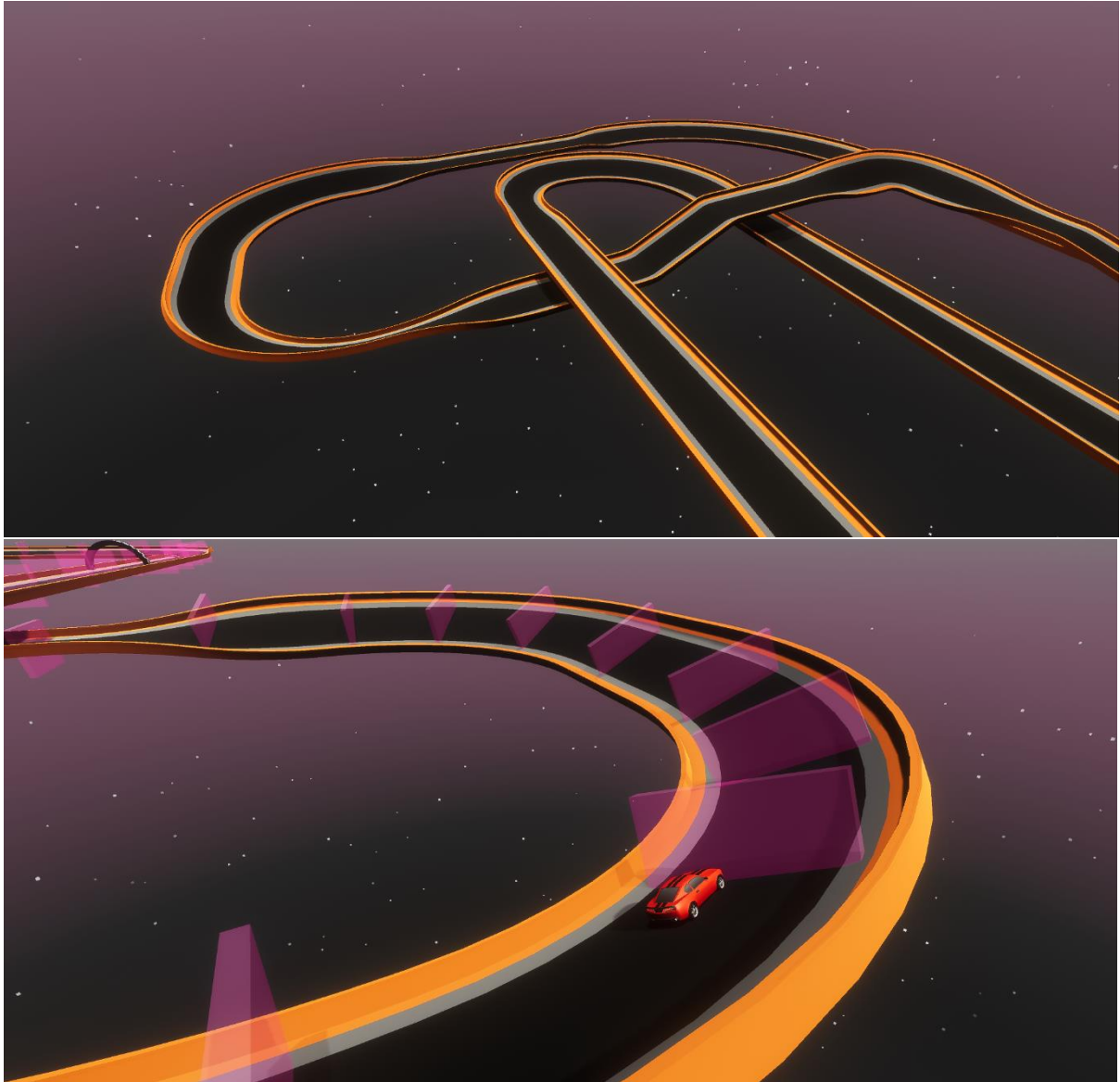


Whilst this did perform averagely the bigger issue is that the agent was not trained for generalisation. This means that on a different race track with obstacles it would struggle to perform well. On top of this there were some strange behavioural patterns displayed by the agents, for instance, if the agent had the option to go left or right to avoid an obstacle it would always go to the right. This actually makes sense as there were times when two obstacles would be positioned next to each other horizontally where the only escape route was on the far right. Essentially there was no obstacle that forced the agent to go to the left side so it learnt to stick to the right side or the middle.

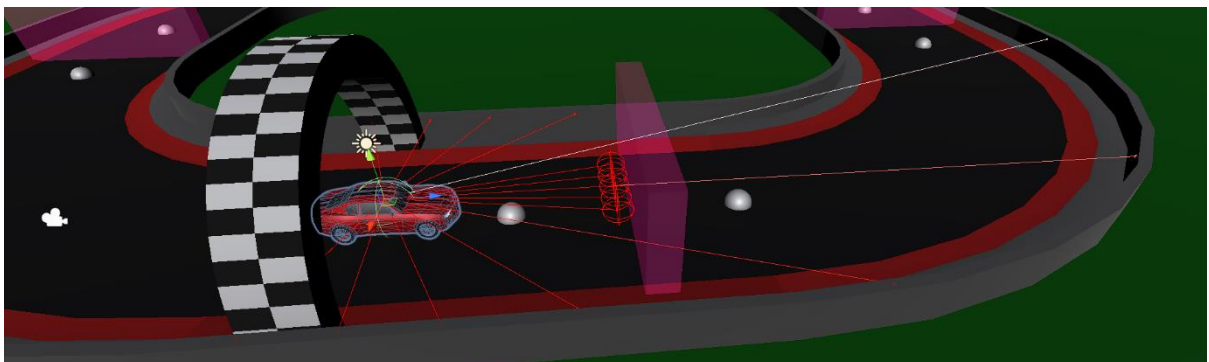
4.7 - Complex Driving Task

Testing the model on a race track with curves, bends, inclines and declines would give an indication of how it performs out of its known environment. Unsurprisingly it did not perform well as it has never experienced a race track similar and so it did not know what actions to take. However, after tweaking some hyper-parameters (mainly the num_epochs, batch size and buffer size mentioned previously) and training the agent on the new track it managed to learn how to complete it successfully. However, there was a significant change in training time as the agent would often get stuck on inclines or sharp turns. After this, the agent was then trained to avoid the walls of the track in the same way as the previous track although, this time the agent struggled a lot with the current set up. When an agent now collides with a wall there

will be an initial penalty and then every physics update where the agent is still in contact with a wall there will be frequent but smaller penalties. This change helped the agents to successfully race around the track with crashes happening but much less frequently.



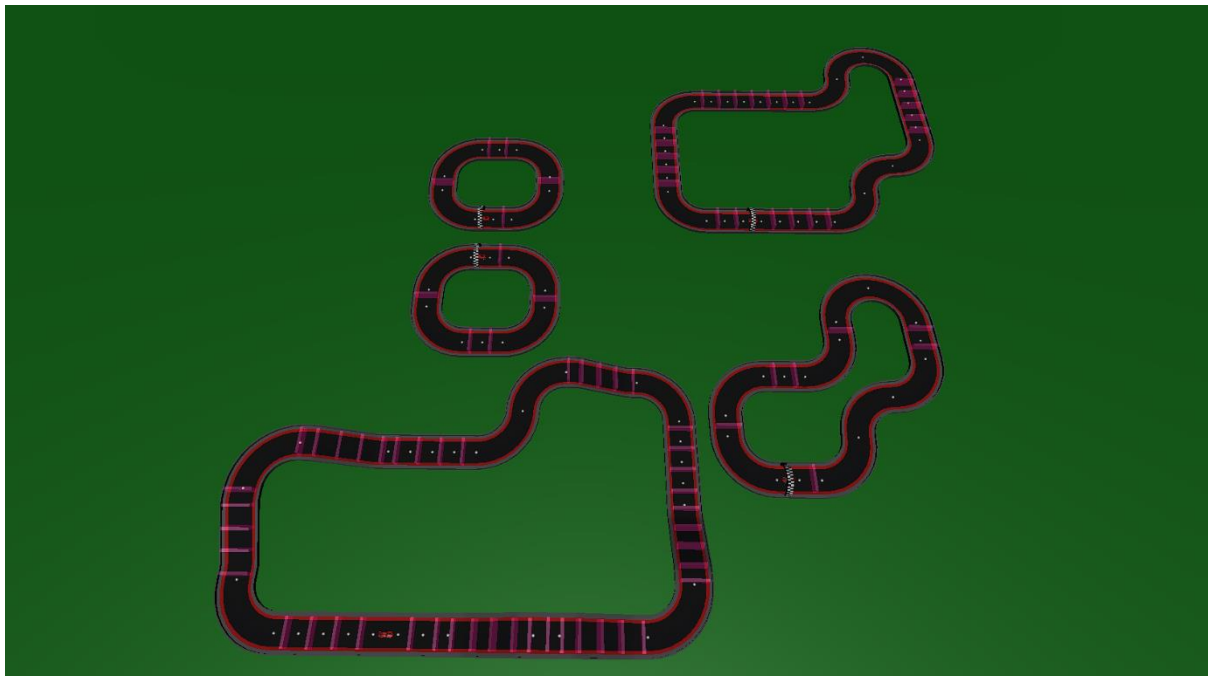
To fix the issue with agents crashing into the walls properly, the sensors on the car were changed so that the agent could both see the checkpoint and what was in front of the checkpoint. In the image below, the agent has more sensors with some colliding with the checkpoints and some passing straight through them. This was done with a layer mask which essentially hides the checkpoint from one sensor.



This small change helped massively with the issue of wall detection and will also allow for more dynamic obstacle detection in the future. The down side to this approach is that more sensors equate to more observations which further slows down the training time.

4.8 - Adaptability

The advantage that machine learning and smart agents have over traditional A.I. is adaptability. By training an agent on a set of race tracks it should then be able to perform well on any other race track that is similar. The agent currently can perform well on complex tracks but cannot adapt to successfully complete a new track. There are various reasons for this but the first and most important one is the way the environment is set up. The checkpoints were placed at random along the track and there are no intentional patterns. After creating some prefabs for each track piece, checkpoint objects were attached as children to the track piece. This means that every time the agent drives on a straight track piece or a corner piece the checkpoints will be positioned and rotated the same way. Essentially this allows for the creation of any possible race track created from the separate pieces to be completed successfully. To test this, the agents were trained on various tracks created from the piece prefabs until they reached a peak reward and stopped learning.



Then, a build system was created that allows the user to cycle through the track pieces and place any piece as long as it is attached to the previous piece. The agent was able to complete the majority of the dynamically created tracks, showing the adaptability of the model. The only times the agent failed were when certain piece combinations proved to be difficult however, more training would solve the issue.

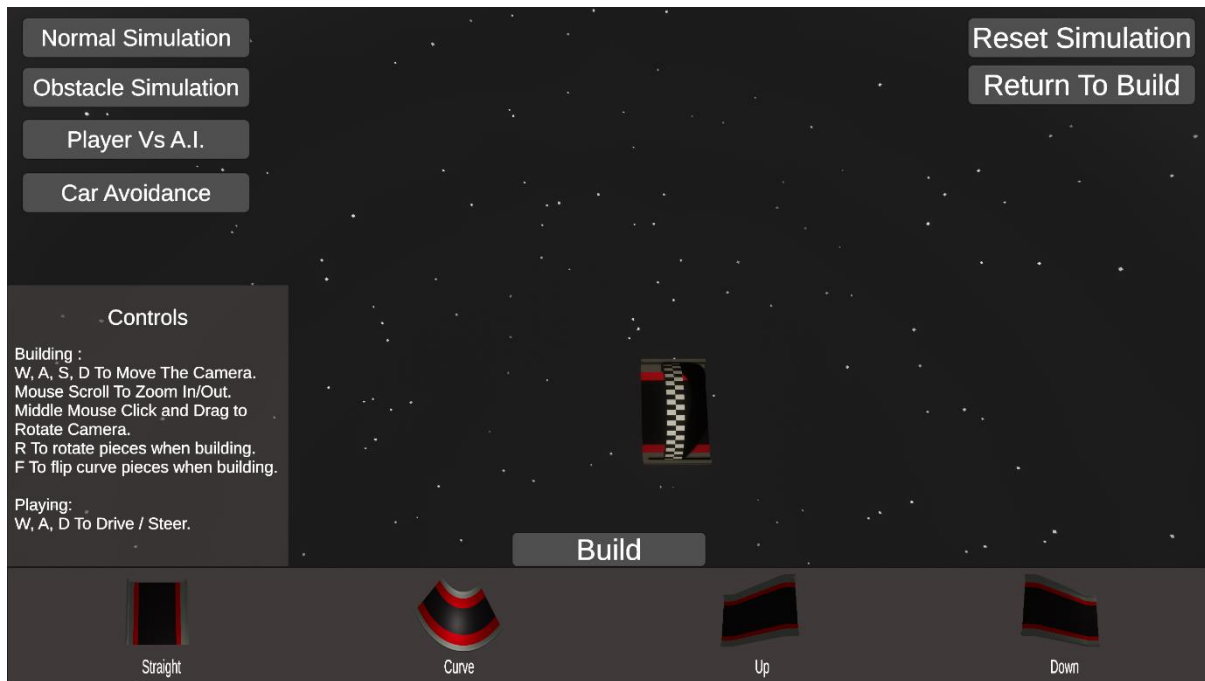
4.9 – Training Process

The training process for almost every model previously talked about (and testing models) were all trained in the same way. Using Anaconda Prompt, the user has to CD into the project file directory. Using the command `mlagents-learn configFileName.yaml --run-id= "FileName"` Would start the training with the parameters in the specified configuration file and will produce a neural network that can be used in Unity. The step count, mean reward and standard deviation of the agent's performance would be output to the console. When the mean peaked and stayed at the peak consistently and the standard deviation was a low value it usually meant that the training was somewhat complete. If at any point an agent needs more training,

using the same command with `--resume` on the end would start the training from the last step it previously trained to. The console can also output a local host web address to show various graphs representing the performance of the agents over time. These have helped massively with testing, parameter tuning and locating potential issues. The training process really doesn't change much but because a large number of observations are being collected, the time taken to train the more complex models took a very long time.

4.10 – Final Design

The final design of the deliverable uses UI elements such as text and buttons to communicate what the user can do.



Here you can clearly see that the user has the ability to build by selecting the track pieces and can run / reset simulations with the buttons on screen. A small panel for the controls is also included for those who're unfamiliar with standard PC navigation key binds.

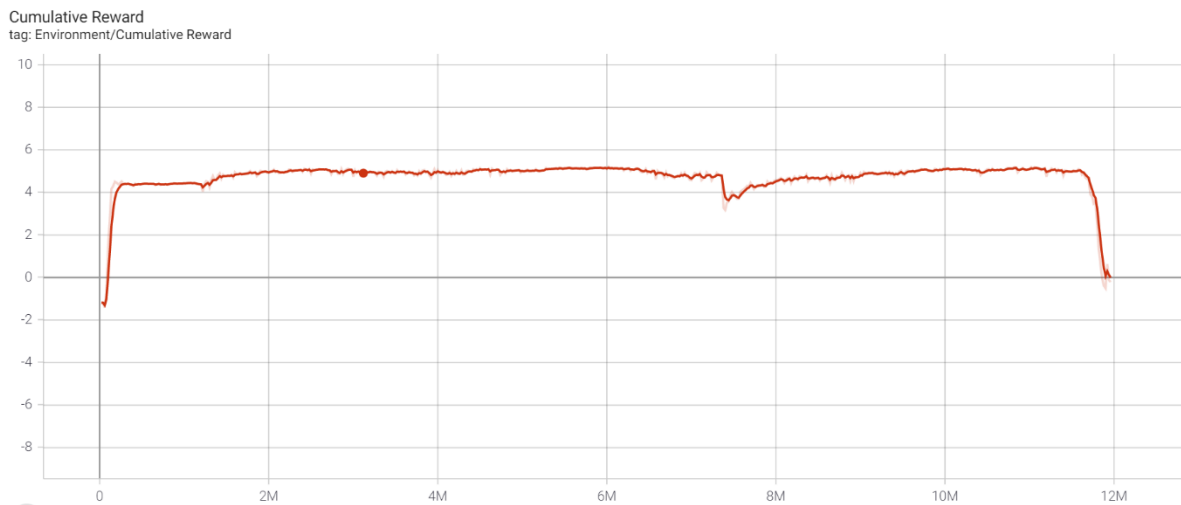
4.11 – Difficulties

As the project evolved over time a few key issues were found which caused considerable difficulty and slowed the overall development largely.

One issue is the car controller itself, as the user can control it with the heuristic option, it is clear to see that the car controller is poor and at times too responsive, making it hard to control. The turning on the car is particularly bad and because of time management issues this was never altered. The cascading result of this is that the agents have to take actions (choosing to accelerate or steer) with an overly responsive controller which makes the task harder to complete. Ultimately, this negatively impacts the performance of the agents heavily.

The next issue is lost progression when training. At times when training an agent, the agents in the simulation would suddenly perform much worse and gradually get even worse over time. This is potentially an instability problem with the version of MLAGents used in the project as it is still in preview (meant only for research). This also occurred when the PC used got pushed to its performance limits and would result in the loss of progress. This meant that several hours spent training agents was lost and simply unrecoverable. The time that was lost here severely

impacted the overall quality of the deliverable and caused a lot of difficulty. To work around this, the training would only happen with a small number of agents in the simulation at once to ensure the PC used was at optimal levels of stress.



The image above is just one example of this happening where the training is going well and then there is a sudden drop to a reward of 0 across all agents. This was at 12 million steps which in this case took around 6 hours of training.

5 – Testing

This section will go over the testing methods used to evaluate if an agent is performing well and will highlight discovered areas that could be improved from the results.

5.1 – Hyperparameter Testing

Lots of hyperparameters were tweaked and tested to see if there were any substantial improvements to the training process and end results. The main change was the number of epochs and as mentioned in Section 4, the value was changed from 3 to 8. This helped improve the training times significantly whilst having no noticeable effect on stability. Higher values were tried however the agent behaviour become more inconsistent and worse overall so it was decided that the value of 8 was appropriate.

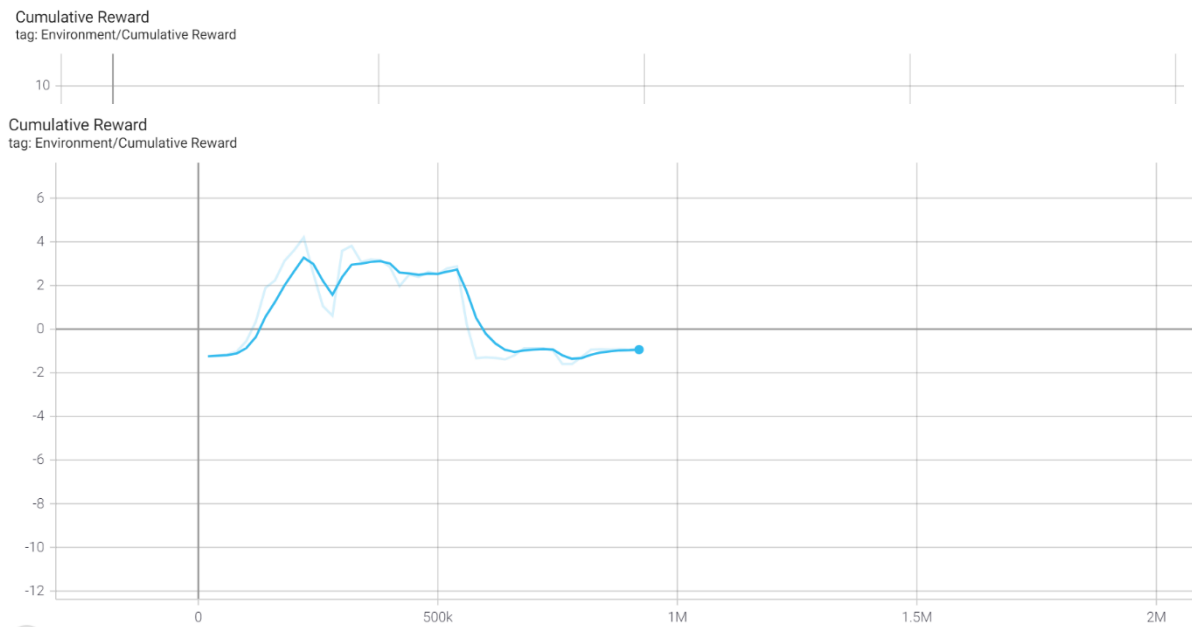
5.2 – Heuristics

Heuristics in MLAgents allow the developer to test the environment and agent with set up player controls. This is extremely useful as training often takes a long time and ensuring that the environment is set up properly first is important. This helped with testing the track, making sure it was possible and checking if all of the checkpoints were in the right order. On top of this it provided a fair way of controlling the car which helped to gather data on human players.

5.3 – Reward Testing

Testing if a reward system is set up well is quite difficult as there is no solution for all purposes and there is no way to tell if the current system will work well before trying it. By using trial and error an appropriate system can be created that will encourage the intended behaviour. One example is an agent will receive an extra positive reward for completing a lap faster than its previous record speed. By doing this, the agents will constantly try to go faster until it reaches a balance of speed and consistency that results in the maximum reward.

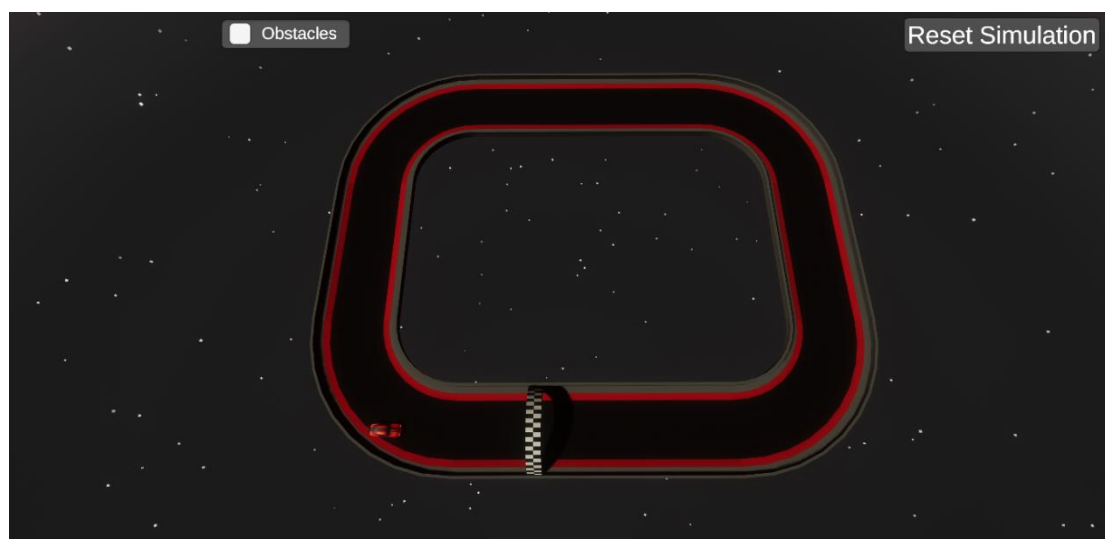
Tensor board also helped with this as it graphically shows how long agents take to learn and shows when they hit their peak cumulative reward. This can be used to evaluate if tasks are too hard or are if the training is taking too long.



The graph above shows that the agent has a steep learning curve indicating that it is learning quickly and is not struggling with the given task. After reaching a peak it flattens out which shows that the agent is able to consistently complete the given tasks. Then, there is a sudden drop meaning that the agent was no longer able to complete the tasks. This drop is where the agents started to be trained to have the ability to climb inclines. From this graph and the behaviour on screen it's clear that the task was too difficult and therefore needed to be simplified.

5.4 – Final Results

The performance of the agents was recorded and for comparison player data was also recorded within the same environment. The race tracks that the data was gathered from were tracks that the agents had not previously trained on to evaluate the adaptability of the agents.

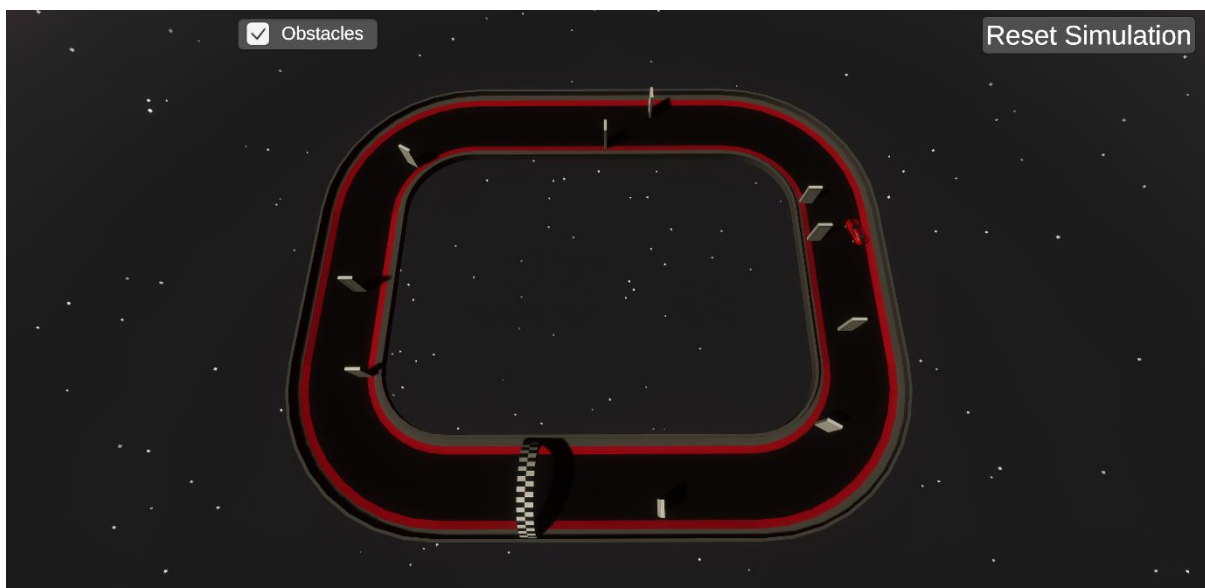


	Failed Attempts: 0	Normal Model	Difficulty	Easy
		Lap Number	Lap Times	Training Step
		1	6.919994	5.28M
		2	6.619995	5.28M
		3	6.279995	5.28M
		4	6.359995	5.28M
		5	6.979994	5.28M
		6	7.139994	5.28M
		7	6.399995	5.28M
		8	7.259994	5.28M
	Average Time	9	6.439995	5.28M
	6.7619945	10	7.219994	5.28M
	Failed Attempts: 0	Player Normal	Difficulty	Easy
		Lap Number	Lap Times	Training Step
		1	7.039994	0
		2	6.439995	0
		3	8.119996	0
		4	7.239994	0
		5	7.339994	0
		6	8.880013	0
		7	10.96006	0
		8	7.539994	0
	Average Time	9	7.799994	0
	8.0520054	10	9.16002	0

Above are the results gathered on the easy track for both the agent and player with no obstacles.

The track is very simple and both the agent and player completed all ten laps without any failures. The average time for the agent was 6.76 seconds which is faster than the player by roughly 1.3 seconds as their average was 8 seconds. This means that the agent is on average 18% faster than a human player.

The introduction of obstacles changed the data quite significantly. In the Image below the white cuboid shapes represent obstacles that the car should try to avoid whilst completing the track.

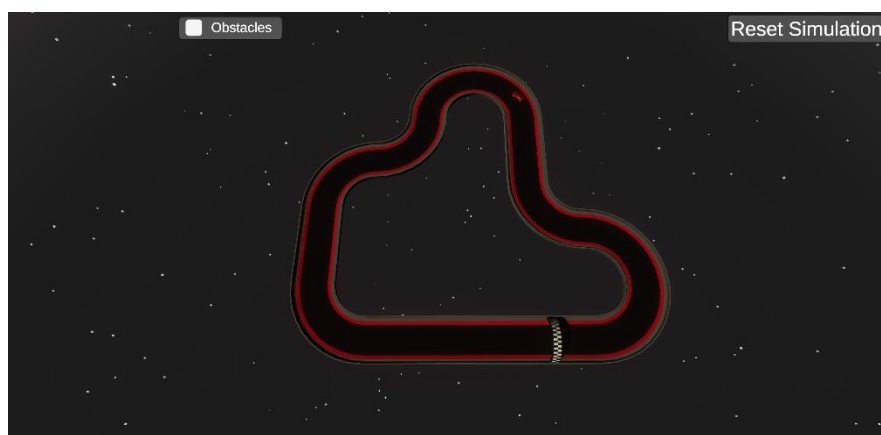


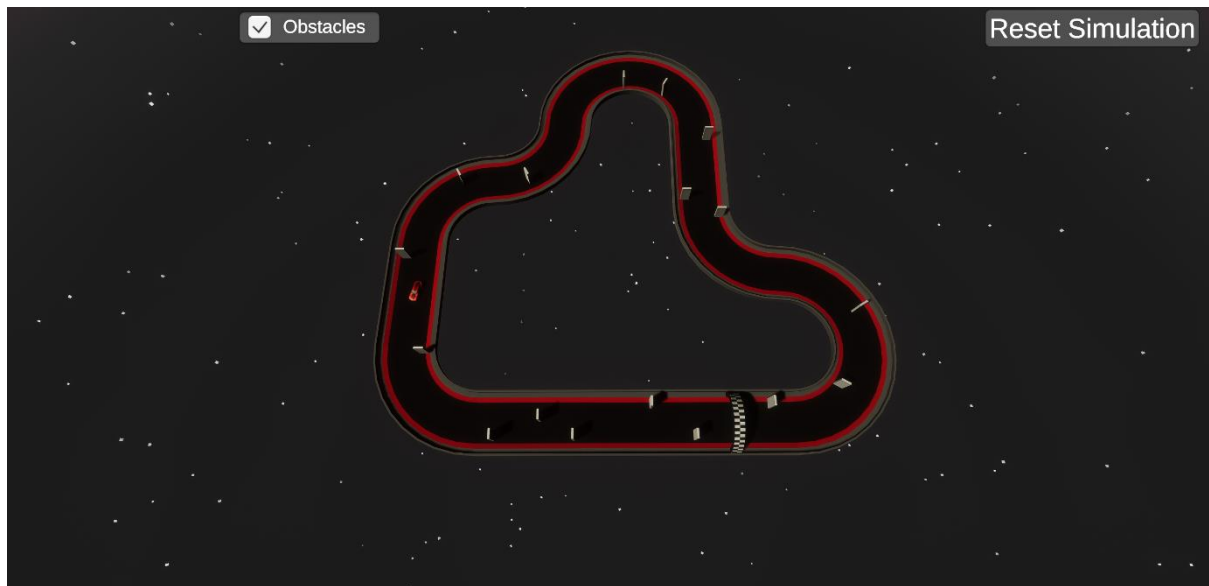
Failed Attempts: 0	Obstacle Model	Difficulty	Easy
	Lap Number	Lap Times	Training Step
	1	9.560029	3.76M
	2	10.04004	3.76M
	3	9.500028	3.76M
	4	8.660008	3.76M
	5	9.360024	3.76M
	6	8.820012	3.76M
	7	10.56005	3.76M
	8	8.099996	3.76M
Average Time	9	8.28	3.76M
9.1560196	10	8.680009	3.76M
Failed Attempts: 0	Player Obstacle	Difficulty	Easy
	Lap Number	Lap Times	Training Step
	1	12.04009	0
	2	10.88006	0
	3	14.54014	0
	4	9.400025	0
	5	13.36012	0
	6	14.66015	0
	7	13.36012	0
	8	13.12011	0
Average Time	9	16.08018	0
12.9721085	10	12.28009	0

Just from the introduction of obstacles the average time for the agent went from 6.76 to 9.15 seconds, an increase of 35%. This is because of two potential factors. The first being that the model was trained for less time than the other and so it is worse at driving around the tracks in general. The second is the fact that the agent avoids the obstacles which means that it cannot take the fastest route to completion and because it is always checking for obstacles the agent will slow down going into and coming out of corners.

The player time also increased because the obstacles acted as barriers that stopped the player from moving freely. On top of this, because the player does not have to avoid the walls it can hug the side walls and dodge most of the obstacles. Even with that advantage, the agent outperformed the player by around 40%.

The same method of gathering data was used on tracks of medium and hard difficulty.





Failed Attempts: 0	Normal Model	Difficulty	Medium
	Lap Number	Lap Times	Training Step
	1	10.18004	5.28M
	2	9.820035	5.28M
	3	9.540029	5.28M
	4	10.00004	5.28M
	5	10.22004	5.28M
	6	9.960038	5.28M
	7	9.860036	5.28M
	8	10.12004	5.28M
Average Time	9	9.940038	5.28M
10.0440396	10	10.80006	5.28M
Failed Attempts: 1	Player Normal	Difficulty	Medium
	Lap Number	Lap Times	Training Step
	1	11.14007	0
	2	10.28005	0
	3	11.12006	0
	4	11.62008	0
	5	11.54007	0
	6	13.20011	0
	7	12.4801	0
	8	13.18011	0
Average Time	9	12.8001	0
11.884082	10	11.48007	0

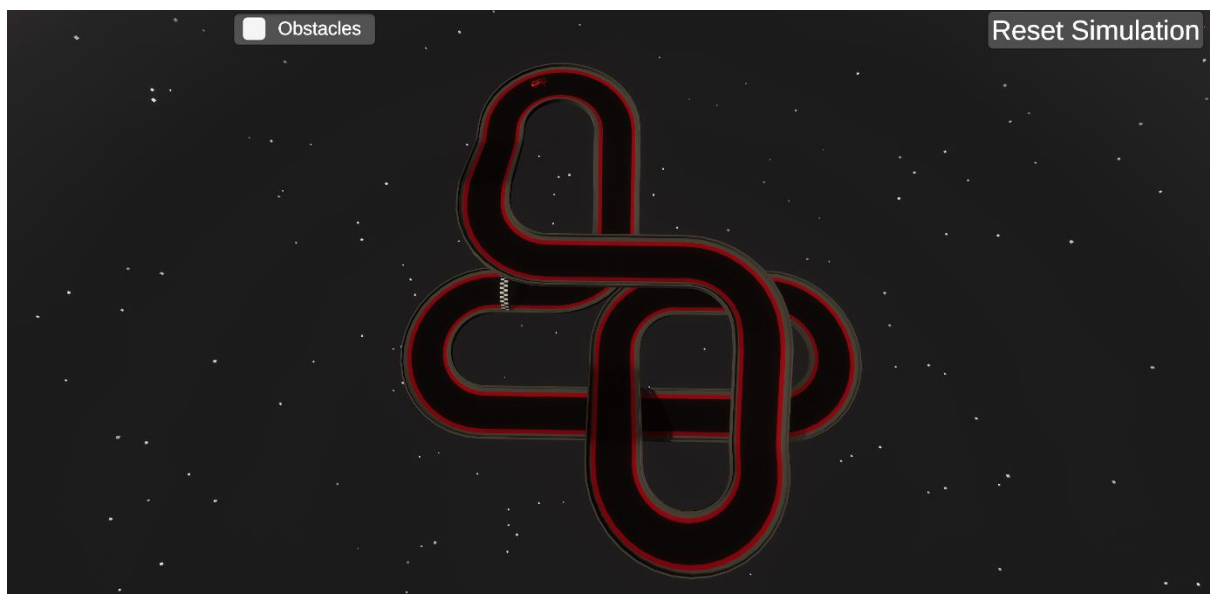
On the medium track in the images above, without obstacles it's very similar to the easy track. The agent outperformed the human data by 19% however, this time the human player had one failed attempt.

Failed Attempts: 0	Obstacle Model	Difficulty	Medium
	Lap Number	Lap Times	Training Step
	1	12.5201	3.76M
	2	12.7801	3.76M
	3	15.78017	3.76M
	4	13.98013	3.76M
	5	15.66017	3.76M
	6	14.98015	3.76M
	7	14.06013	3.76M
	8	14.42014	3.76M
Average Time	9	14.00013	3.76M
14.108133	10	12.90011	3.76M

Failed Attempts: 1	Player Obstacle	Difficulty	Medium
	Lap Number	Lap Times	Training Step
	1	17.2402	0
	2	14.00013	0
	3	14.32014	0
	4	18.28023	0
	5	16.14018	0
	6	16.50019	0
	7	20.54028	0
	8	17.2602	0
Average Time	9	20.90029	0
17.312206	10	17.94022	0

With obstacles, the agent completed the track with an average time of 14 seconds which is an increase of 40%. Similarly, the agent outperformed the human data by roughly 23% and the player failed once again.

Moving onto the most complex track that data was gathered from. The main difference here is that the hard track now uses elevation with inclines and declines.



Failed Attempts: 2	Normal Model	Difficulty	Hard
	Lap Number	Lap Times	Training Step
	1	22.90033	5.28M
	2	23.98036	5.28M
	3	22.58033	5.28M
	4	22.46032	5.28M
	5	22.12032	5.28M
	6	22.18032	5.28M
	7	22.46032	5.28M
	8	22.32032	5.28M
Average Time	9	22.52032	5.28M
22.676328	10	23.24034	5.28M
Failed Attempts: 0	Player Normal	Difficulty	Hard
	Lap Number	Lap Times	Training Step
	1	23.00034	0
	2	24.50037	0
	3	26.30041	0
	4	25.6204	0
	5	23.20034	0
	6	28.28046	0
	7	25.58039	0
	8	19.26025	0
Average Time	9	27.22043	0
24.566372	10	22.70033	0

Here, the agent has an average time of 22.7 seconds and the player has an average time of 24.6 seconds which is a percentage difference of only 8%. On top of this, the agent failed 2 times whereas the player didn't fail. This means that with the current data the agent has a success rate of 83%. This makes sense as introducing elevation increases the chances of falling off the track however, the agent only ever failed by spinning out when trying to go uphill. It is also worth mentioning that on lap number 8 the player had the fastest time out of all with a time of just 19.2 seconds. This means that it is very possible to outperform the agent however, most of the time the agent will perform better and more consistently.

The results on the same track with obstacles has a similar pattern where the agent is slower than without obstacles but still outperforms the player. Here both the player and agent failed 3 times so they both have a success rate of 77%. All three times the player failed it was because the car flew off the track by going up or down a ramp too quickly. The agent however, failed in the same way as before by spinning out on an incline.

It can be concluded that without obstacles a player can sometimes match or even beat the agent and with obstacles a player will most likely lose to the agent.

Failed Attempts: 3	Obstacle Model	Difficulty	Hard
	Lap Number	Lap Times	Training Step
	1	27.74044	3.76M
	2	28.14045	3.76M
	3	30.3805	3.76M
	4	27.92045	3.76M
	5	28.88047	3.76M
	6	30.64051	3.76M
	7	29.20048	3.76M
	8	30.3605	3.76M
Average Time	9	26.96043	3.76M
28.832468	10	28.10045	3.76M

Failed Attempts: 3	Player Obstacle	Difficulty	Hard
	Lap Number	Lap Times	Training Step
	1	32.00054	0
	2	37.20066	0
	3	30.0005	0
	4	33.58058	0
	5	29.60049	0
	6	35.08061	0
	7	30.54051	0
	8	32.68056	0
Average Time	9	29.58049	0
32.524555	10	34.98061	0

These results highlight which areas of the agent need to be worked on as there is a persistent issue with the inclines. To solve this, both making the incline less steep and increasing the amount of training would be necessary. Due to time constraints the agent will have to stay as is but this will be looked at more thoroughly in future developments.

6 – Critical Evaluation & Reflection

This section will evaluate and reflect over various areas of the project and discuss what went well, what didn't, what could be improved, future aspirations and will conclude whether or not the initial aims of the project were met.

6.1 – Personal Reflection

Poor time management was one of the major issues that negatively impacted this project. Much of the work for the deliverable and report were completed close to the deadline which resulted in a less complete project. With more time, effort and organisation it would be expected that the deliverable would be of higher quality and more research would also have been done to aid this development. On top of this, outside of the main aims, the direction for the project wasn't as clear as it should have been, meaning time was spent creating and developing unnecessary pieces of work. All of this combined with the loss of progression issues previously mention in section 4, the project was not as smooth of a process as it could have been. The loss of progression would make the little time put in feel wasted and resulted in a poor attitude towards further developing the project.

6.2 – Development Reflection

Developing the project went well (when the previously mentioned issues didn't persist) and a lot of knowledge about machine learning was gained over the year. It was difficult at first to adapt to a new way of developing A.I. behaviours however, after some time and recognising what worked through trial and error it became easier. The main issue that stopped the development from being efficient is the way the training works. Whilst training, the simulation

is in play mode, meaning no changes made will be saved. This means that with only one PC you can't train and develop at the same time and as training often takes hours, the development time for other sections had to be cut. On top of this it wasn't possible to have the training handled in a separate project on the same machine as the training would require a lot of the PC's processing power and most other applications had to be closed for it to work well on the PC used for this project. This could be solved by having a PC that separately handles the training whilst another PC can be used to develop on.

More time should have been spent on training, gathering results and using them to find issues as a lack of this resulted in inconsistent agent behaviour. If more time was spent on identifying issues and training agents, the behaviour of agents and overall deliverable would be of higher quality.

6.3 – Testing Reflection

The overall testing process went quite smoothly and has helped to provide good results by improving on previous iterations. After training a model for some time, the agent would then be tested on a separate race track that it hasn't trained on before. Evaluating the performance helped to spot any issues such as not being able to complete the track, being too slow and crashing into obstacles. The testing process was done mostly through trial and error by tweaking hyperparameters and rewards to see what combinations resulted in the best overall performance. Doing it this way takes a lot of time and could have probably been sped up significantly with more research in this area. On top of this, the majority of testing was conducted towards the very end of the project which meant that bugs and unwanted behaviours impact the deliverable. If more thorough testing was done throughout, the deliverable is expected to be of higher quality and the performance of the agents would be much more consistent overall.

6.4 – Ethics

The project did not tackle any ethical issues and there were no outside participants. Therefore, there is no relevant reflection on the ethics faced while developing this project.

6.5 – Were the Project Aims Met?

The majority of the Aims specified were met whilst developing the project but to lower quality than initially intended. Some of these aims were to use the MLAgents plugin to create an intelligent agent that can race around a track, avoid obstacles, have multiple agent behaviours and be used as an enemy A.I. to race against. All of these aims were met and completed properly.

Some of the aims specified were not met either because they were cut from the project or because of time constraints. The first being the agent isn't able to avoid dynamic obstacles which was not developed because of a lack of time. However, it is believed that the majority of work to develop this is done but the training was not set up in time and therefore was cut. The next aim was to use both reinforcement learning and imitation learning to develop agents. In this report only reinforcement learning was covered however, imitation learning was used but was deemed to be unsuitable for the deliverable and was also cut. Imitation learning was simply a way of speeding up training times and caused some other unwanted behaviour so it was removed. It was initially in the specification because it was mistaken to be another method of machine learning that could provide more human like behaviour.

One personal aim that was not stated on the specification was to improve upon knowledge of machine learning. As previously mentioned, a lot of knowledge on the topic has been gained and will serve as a good foundation for further machine learning education.

The overall aim however was to create an intelligent agent and evaluate if it is suitable within a game like environment. To conclude, machine learning is viable option to use when considering A.I techniques in game development. There are many advantages mentioned throughout this report and machine learning can be applied in a multitude of ways to provide a new experience of A.I. in games.

6.6 – Future Developments



The future development of this project can be taken in many ways. The idea that comes to mind is replicating a Mario like racing scenario with intelligent agents. All of the agents will try to win a race and would be able to collect usable power ups throughout the race to increase their chances of winning.

This project has opened my eyes into the many possibilities machine learning provides. When developing a game in the future, machine learning may be the choice out of all the possible A.I techniques or maybe even an entire game made up of intelligent agents would be a possibility to consider.

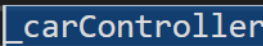
Bibliography

- Bonaccorso, G. (2017, July 24). *Reinforcement learning - Machine Learning Algorithms - Second Edition*. <https://learning.oreilly.com/library/view/machine-learning-algorithms/9781789347999/f90f6955-9aaa-4643-896f-af3a0e3ab1a4.xhtml>
- Brockman, G., Zhang, S., & ChanFilip, B. (2018, June 25). *OpenAI Five*. <https://openai.com/blog/openai-five/>
- Buttfield-Addison, P., Manning, J., Buttfield-Addison, M., & Nugent, T. (2021). *Practical Simulations for Machine Learning* (Early Release). <https://learning.oreilly.com/library/view/practical-simulations-for/9781492089919/#toc>
- IBM Cloud Education. (2020a, July 15). *What is Machine Learning? | IBM*. <https://www.ibm.com/cloud/learn/machine-learning>
- IBM Cloud Education. (2020b, August 17). *What are Neural Networks? | IBM*. <https://www.ibm.com/cloud/learn/neural-networks>
- Lanham, M. (2018). *Learn Unity ML-Agents - Fundamentals of Unity Machine Learning : Incorporate New Powerful ML Algorithms Such As Deep Reinforcement Learning for Games*. <https://ebookcentral.proquest.com/lib/SHU/reader.action?docID=5446051&query=#>
- Marsland, S. (2015). Machine learning : an algorithmic perspective. In *CRC Press is an imprint of Taylor & Francis Group* (2nd ed.). Boca Raton, FL : CRC Press is an imprint of Taylor & Francis Group.
- Pierre, V. (2018, October 19). *ML-agents/Training-PPO.md at master · gzejcx/ML-agents · GitHub*. <https://github.com/gzejcx/ML-agents/blob/master/docs/Training-PPO.md>
- Schulman, J., Klimov, O., Wolski, F., Dhariwal, P., & Radford, A. (2017, July 20). *Proximal Policy Optimization*. <https://openai.com/blog/openai-baselines-ppo/>
- Teng, E. (2019, November 11). *Training your agents 7 times faster with ML-Agents*. <https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/>
- Unity. (2018). *Make a more engaging game w/ ML-Agents | Machine learning bots for game development | Reinforcement learning | Unity*. <https://unity.com/products/machine-learning-agents>
- Unity. (2020a, October 15). *Class Agent | ML Agents | 1.5.0-preview*. <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.5/api/Unity.MLAgents.Agent.html>
- Unity. (2020b, October 15). *Class BehaviorParameters | ML Agents | 1.5.0-preview*. <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.5/api/Unity.MLAgents.Policies.BehaviorParameters.html>
- Unity. (2021). *ml-agents/Training-Configuration-File.md at main · Unity-Technologies/ml-agents · GitHub*. <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md>
- Winder, P. (2020). *Reinforcement Learning*. <https://learning.oreilly.com/library/view/reinforcement-learning/9781492072386/>
- Xav De, M. (2014, July 7). *Meet the computer that's learning to kill and the man who programmed the chaos | Engadget*. <https://www.engadget.com/2014-06-06-meet-the-computer-thats-learning-to-kill-and-the-man-who-progra.html>

A.1 – Car Agent Script

```
1   using System.Collections;
2      using System.Collections.Generic;
3      using Unity.MLAgents;
4      using Unity.MLAgents.Actuators;
5      using Unity.MLAgents.Sensors;
6      using UnityEngine;
7
8       Unity Script | 1 reference
9      public class CarAgent : Agent
10     {
```

A.2 – OnEpisodeBegin() Function

```
//Called each time it has timed-out or has reached the goal
0 references
public override void OnEpisodeBegin()
{
    _checkpointManager.ResetCheckpoints();
     _carController.Respawn();
}
```

A.3 – CollectObservations() Function

```
public override void CollectObservations(VectorSensor sensor)
{
    Vector3 diff = _checkpointManager.nextCheckPointToReach.transform.position - transform.position;
    sensor.AddObservation(diff);
    AddReward(-0.001f);
}
```

A.4 – OnActionReceived() Function

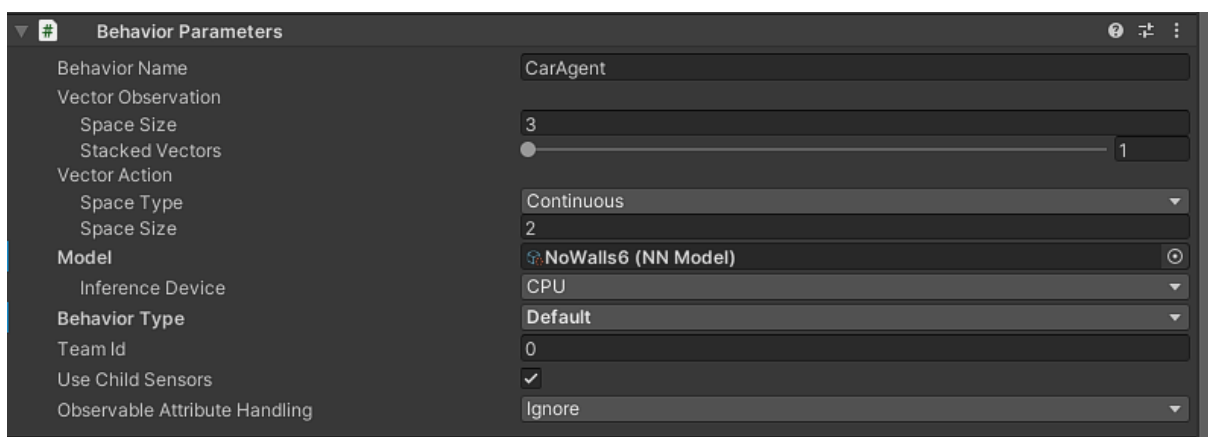
```
public override void OnActionReceived(ActionBuffers actions)
{
    var input = actions.ContinuousActions;
    _carController.ApplyAcceleration(input[1]);
    _carController.Steer(input[0]);
}
```


A.5 – Heuristic() Function

```
0 references
public override void Heuristic(in ActionBuffers actionsOut)
{
    var action = actionsOut.ContinuousActions;

    action[0] = Input.GetAxis("Horizontal"); //steering
    action[1] = Input.GetKey(KeyCode.W) ? 1f : 0f; //Acceleration
}
```

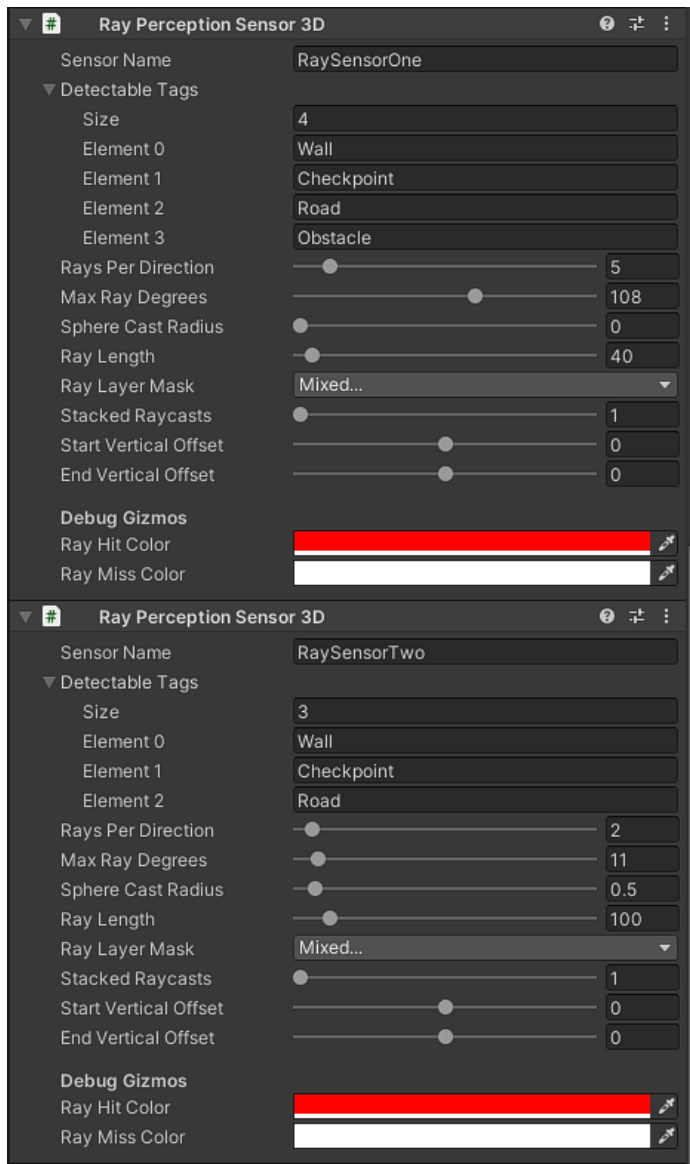
A.6 – Behaviour Parameter Component



The screenshot shows the 'Behavior Parameters' component inspector in Unity. The component is named 'CarAgent'. The 'Vector Observation' section has 'Space Size' set to 3 and 'Stacked Vectors' set to 1. The 'Vector Action' section has 'Space Type' set to 'Continuous' and 'Space Size' set to 2. The 'Model' section has 'NoWalls6 (NN Model)' selected. The 'Inference Device' is set to 'CPU'. The 'Behavior Type' is set to 'Default'. The 'Team Id' is set to 0. The 'Use Child Sensors' checkbox is checked. The 'Observable Attribute Handling' is set to 'Ignore'.

Parameter	Value
Behavior Name	CarAgent
Vector Observation	
Space Size	3
Stacked Vectors	1
Vector Action	
Space Type	Continuous
Space Size	2
Model	NoWalls6 (NN Model)
Inference Device	CPU
Behavior Type	Default
Team Id	0
Use Child Sensors	<input checked="" type="checkbox"/>
Observable Attribute Handling	Ignore

A.7 – Ray Perception Sensor 3D Component



A.8 – Small Reward Given

```
    else
    {
        carAgent.AddReward(0.1f);
        SetNextCheckpoint();
    }
}
```

A.9 – Large Rewards Given For Finishing Laps

```
if (CurrentCheckpointIndex >= Checkpoints.Count)
{
    carAgent.AddReward(2f);
    currentLapCount++;

    if (fastestLapTime == 0)
    {
        fastestLapTime = currentLapTime;
    }
    if (currentLapTime < fastestLapTime)
    {
        fastestLapTime = currentLapTime;
        carAgent.AddReward(1f);
    }

    //lapTimes.Add(currentLapTime);
    carAgent.EndEpisode();
}
```

A.10 – Config File Behaviours

```
behaviors:
  CarAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 16384
```

A.11 – Config File Hyper-Parameters

```
beta: 0.001  
epsilon: 0.2  
lambda: 0.95  
num_epoch: 8
```

A.12 – Config File Network Settings

```
network_settings:  
  normalize: true  
  hidden_units: 256  
  num_layers: 2
```

A.13 – Config File Reward Signals

```
reward_signals:  
  extrinsic:  
    gamma: 0.99  
    strength: 1.0
```

A.14 – Penalties for Collisions

```
Unity Message | 0 references
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Wall")
    {
        carAgent.AddReward(-0.1f);
        Debug.Log("wall hit!");
        Debug.Log(collision.gameObject.name);
    }
    else if (collision.gameObject.tag == "Obstacle")
    {
        carAgent.AddReward(-0.25f);
        Debug.Log("Obstacle hit!");
        Debug.Log(collision.gameObject.name);
    }
    if (canCollide)
    {
        if (collision.gameObject.tag == "CarCollide")
        {
            carAgent.AddReward(-0.25f);
            Debug.Log("Car hit!");
        }
    }
}

Unity Message | 0 references
void OnCollisionStay(Collision collision)
{
    if (collision.gameObject.tag == "Wall")
    {
        carAgent.AddReward(-0.01f);
    }
    else if (collision.gameObject.tag == "Obstacle")
    {
        carAgent.AddReward(-0.01f);
    }
}
```

PROJECT SPECIFICATION - Project (Technical Computing) 2020/21

Student:	Mohammed Mahmoud
Date:	20/10/2020
Supervisor:	Alessandro Di Nuovo
Degree Course:	Computer Science for Games
Title of Project:	Developing smart A.I through machine learning techniques and applying them to game-like scenarios.

Elaboration

For my project I want to evaluate if Machine Learning techniques can be used within game development. I want to explore if A.I can be trained with reinforcement learning (and maybe imitation learning) to create smart or more human like A.I. I think this will be an interesting project as it will allow me to have direct comparisons to other A.I methodologies and will provide an alternative approach to the A.I techniques that I'm aware of. I've always found A.I to be an interesting topic and I want to use my final year project as an opportunity to explore and learn new techniques.

Project Aims

- Use Unity and the MLAgents plugin to learn about how machine learning can be implemented into game development
- Create a basic demo where a car can switch lanes to avoid oncoming traffic. This will be a very simple project to help me learn the basics of Machine Learning within Unity
- Create an A.I model that has multiple sensors and learns how to go from one point to another without crashing
- Create an A.I model that can successfully race around a race track that has dynamically moving/ changing objects without crashing. (A model that can adapt to its environment)
- I want to have different A.I models to see how their behaviour can differ. For example, some car models may favour speed and will try to complete the track as fast as possible where as others may favour less crashes and more accurate driving.
- Use both reinforcement learning and imitation learning to see which provides a better A.I model to use alongside a player.
- Have player input so that they can race alongside the A.I to get a feel for how player-friendly the A.I is. (decide if It's realistic, too easy or too hard)
- Evaluate if these techniques are a viable method of A.I in games.

Project deliverable(s)

What are you going to produce and on what platform are you going to deliver it? What engineering approaches are you going to use?

I will develop a car racing demo where the A.I have been trained through different Machine Learning techniques allowing the A.I to make decisions based upon its environment. This allows the A.I to adapt dynamically and can produce smart and sometimes human like behaviours.

I will create this project in the Unity game engine using C# and a plugin called MLAgents. This allows unity to connect to the python API TensorFlow which handles the training aspect of the A. I. I will achieve this by attaching some virtual sensors the A.I models that allow the model to understand what its surroundings are. Combining this with a reward system that teaches the A.I what it has done right and wrong will result in an A.I that can adapt and learn from the data it collects.

Action plan

Task Deadlines

Task	Deadline Date
Learn about ML and the MLAgents plugin by creating a simple demo where a car avoids oncoming traffic. (Reinforcement learning)	Friday November 20 th 2020
Create an A.I model with multiple sensors allowing it to "see" its surroundings. (Reinforcement learning)	Monday December 7 th 2020
Train an A.I car to move from the start point to the finish line without crashing. (Reinforcement learning)	Sunday December 27 th 2020
Train an A.I car to race around the same track with objects that change dynamically. (Reinforcement learning)	Friday January 15 th 2021
Train another A.I model with different behaviours so there is variance. (Reinforcement learning)	Friday February 26 th 2021
Learn about Imitation learning to provide a different approach which creates more human like A.I behaviours. (Imitation learning)	Friday March 26 th 2021
Have Player Input so that the player can race against the A.I to see if the techniques used to train the A.I are fair, balanced and effective. (Heuristics)	Friday April 2 nd 2021

Milestone Deadlines

Task	Deadline Date
Create a basic prototype that teaches me about how machine learning works within the Unity game engine. (Milestone 1)	Friday November 20 th 2020
Create an A.I model that can successfully drive around a rack track without crashing. (Milestone 2)	Sunday December 27 th 2020
Evolve the A.I model so that it can dynamically adapt to static obstacles that spawn in at run time. (Possibly moving obstacles as well). (Milestone 3)	Friday January 15 th 2021
Train more A.I models that have differing behaviours to see how it effects the race. (Milestone 4)	Friday February 26 th 2021
Use imitation learning to provide an alternative approach to see if it creates a better A.I model for a racing game. (Milestone 5)	Friday March 26 th 2021

Module Deadlines

Task	Deadline Date
Project specification & Ethics form	Friday 23 rd October 2020
Information Review	Friday 4 th December 2020
Provision Contents Page	Friday 19 th February 2021
Critical Evaluation (Draft)	Friday 19 th March 2021
Project Report (Draft)	Friday 19 th March 2021
Project Report Hand In	Thursday 15 th April 2021
Project Deliverable Hand In	Thursday 15 th April 2021
Demonstration of Deliverable	Thursday 19 th April 2021

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.

Signature: Mohammed Mahmoud

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

Signature: Mohammed Mahmoud

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module and as an appendix in the handbook.

Signature: Mohammed Mahmoud