**Creating an Autonomous Car Using Unity’s ML-Agents**

**Steven Smith**

**Matric No: S1315451**

**Abstract:** This report contains a discussion on artificial intelligence as a whole and also its use within videogames, it then discusses machine learning and the different techniques that exist before choosing a project to create using Unity’s ml-agents. A walkthrough is then supplied to show the set-up and creation of both the environment and the agent before showing how the agent is trained and discussing the positives and negatives of machine learning and the techniques used in the making of this project.

**Introduction:**

Artificial intelligence (AI) can raise difficult philosophical questions such as “what is true intelligence?” and ethical conundrums that could prove endless to answer. The main goal of AI is, for the computer scientist at least, to create a machine which behaves like a person and is capable of displaying some form of intelligence (Ertel, W. 2018). AI has applications in many fields such as helping to train pilots in aviation and predicting market sales in finance, the focus for this essay, however, is AI in videogames. Early on in the landscape of videogames, complex AI techniques were rarely used, this was in part due to developers having less experience with the field of AI and also due to the limitations of technology at that time (Moffat, D. C. & Romano, D. M.).

Modern videogames, however, may use a large amount of AI within their developed game. The use of AI in modern videogames varies wildly and has many applications. One of the most common applications of AI in modern videogames is to create behaviours for non-playable characters (NPCs) within the game environment. In early videogames NPCs would have scripted behaviours and, as such, would perform the same interactions with the player every time they were encountered. The use of modern AI techniques now allow for an NPC to generate interesting and new behaviours that look almost lifelike in the confines of their respective game worlds (Yannakakis, G. N. & Togelius, J. 2017). With this improved technology, the user could encounter the same NPC twice and have two extremely different encounters, this may aid the replay-ability and boost the overall user experience. AI in videogames can also be used to simulate an opponent for the user in games such as chess, provide a means of creating entire game levels with procedural content generation and more.

Not only can AI be seen as a way to possibly improve a videogame but videogames themselves provide an incredibly strong platform for the testing and advancement of AI technologies (Gold, A. 2005), this is due in part to the fact that modern videogames often strive for realism in their lighting and graphics etc. This has led to the development of very powerful game engines such as the Unity engine and Unreal engine. These can be used as a starting platform to develop or test AI without having to create everything from scratch, making the development of AI a far more accessible subject. The form of AI being focused on in this essay, however, is machine learning. As with the previous examples, it comes as no surprise that machine learning itself has many approaches and applications. At its core, machine learning can be seen as providing a computer system the ability to “learn” with data while not being explicitly programmed to do so. This can be useful in many fields such as data analytics it can be a powerful tool for predicting outcomes using large datasets of current and past data, this can even lead to new relationships and insights being discovered within the data being processed. In videogames, machine learning can be used to create an AI capable of playing games that can rival and even beat human players (Narula, G. 2017). Several well-known machine learning AIs exist, the most famous ones being designed to play the game of chess, such as the original Deep Blue or the newer Stockfish, recently though, with AI becoming more and more accessible, several developers have taken to creating machine learning AI that play different games of their choice such as MarI/O which is an artificial neural network designed to play the popular game Super Mario.

**Focus:**

The aim of this report is to document the development process of a similar AI discussed previously using machine learning to play a simple racing game. When developing an AI for the purpose of playing a game or videogame, there are two categories that the game will fall into. These categories are games of perfect information where all information is known, an example of this would be the game of chess as the player is aware of the full state of the board and all moves available at any one time. The other category is games of imperfect information, this means that not all aspects of the games are known to the player, a simple example of this is the game Battleship as one player does not have knowledge of the placement of the other players battleships. In the game that will be developed as a testing bed for the training of the AI, the player will not have full information of the track as it cannot be fully seen at any one time, this means the game being developed is one of imperfect information.

As machine learning is a very large subject, it contains many areas and techniques to use such as genetic algorithms and decision tree learning. The technique being focused on in this essay is reinforcement learning, this focuses on an agent that takes actions within a given environment to maximise a reward. In this example, the agent will be the car, the environment will be the race track and the actions will be the controlling of wheel colliders to change the speed and direction of the car. The reward will be balanced between not striking the side of the track and the forward velocity of the agent’s car. As discussed in the introduction, game engines have allowed for easier access to developing AI, recently Unity has released a package with example environments for training machine learning agents titled “ml-agents”. Unity has also included packages to allow unity environments to work with Pythons Anaconda prompt and Tensorflow, because of this, Unity will be used to construct the training environment and set up the agent’s car.

Reinforcement learning itself also contains a few different options for the learning algorithms, the one that has been mainly tested with Unity’s ml-agents is proximal policy optimization (PPO). This is a good choice for the learning algorithm as PPO has been shown to perform comparably or improves upon other modern approaches while being simpler to implement and tune and is in fact the default reinforcement learning algorithm for OpenAI due to its ease of use and overall performance. Now that the choices for the machine learning technique, reinforcement learning algorithm and game engine for the creation of the training environment have been chosen, the development of the project can begin.

**Practical Illustration:**

This section of the report will cover the project through its development from initial set-up to the finished product. This will be broken up into three sections: the initial set-up and installations, creating the unity training environment and rewards and finally the training using the finished environment and PPO algorithms in Tensorflow. The full project was created on Windows 10 and will be explained explicitly for this platform, other platforms may run into issues when following this report.

**Initial Set-up:**

A number of initial steps must be carried out before development of the environment can begin. The first step is to download the newest version of Unity 2017, the version used in the creation of this AI was Unity 2017.3.1. Next, NVidia’s CUDA toolkit must be downloaded and installed, the version used was CUDA Toolkit 9.0, there is also a neural network library called cuDNN that should be added to the current CUDA toolkit, the version used was cuDNN v7.1.2 for CUDA 9.0, once these are added new system paths must be set-up, the first being CUDA\_HOME which should be given the same path as the pre-existing CUDA\_PATH variable. Finally, two environment paths must be added under “path”, these paths should equal the following:

\*filepath\*\CUDA\v9.0\lib\x64

\*filepath\*\CUDA\v9.0\extras\CUPTI\libx64

Where the \*filepath\* is the users file path to the CUDA folder installed previously. CUDA allows for training to take place using a computers GPU instead of CPU which can greatly decrease the time it takes to train an agent, cuDNN is also specifically designed for the purpose of training deep neural networks, NVidia has reported that newer iterations of cuDNN with modern hardware can increase training speeds by up to three times faster. This toolkit along with the additional library will ensure that the training taking place will be very efficient which will save time when training.

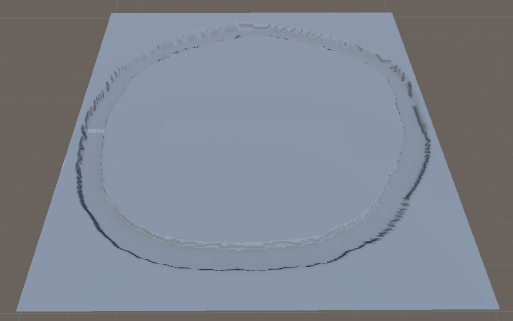
The next step is to download and install the Anaconda prompt, the version used in this report was Anaconda3 v5.1.0. Anaconda will allow for the installation of the Unity python packages and the set-up and installation for Tensorflow. Once Anaconda has been downloaded and installed, the Anaconda prompt should then be searched for and launched. The first command that should be run on the Anaconda prompt is to download and install Tensorflow as this will be needed when training the agent. The command used to do this is “conda create –n tensorflow-gpu python=3.5.2”, this command essentially tells Anaconda to create a new tensorflow-gpu environment and what version of python is being used. Once the environment is set up, the command “activate tensorflow-gpu” should be input to activate the environment followed by “pip install tensorflow-gpu”, these commands should carry out without errors if the paths have been created correctly. Next a python environment is started by inputting the command “python”. The command “import tensorflow as tf” should then be input into the python environment. This concludes the set-up of Tensorflow and Anaconda.

The final step of the set-up is to download or clone the ml-agents package from Unity’s GitHub. For this report the repository was downloaded. Once again an Anaconda prompt should be opened and this time should be navigated to the position of the ml-agents folder. This folder contains sub-folders, the sub-folder “python” should be navigated to. Finally the command “pip install .” should be run. Note: the Anaconda prompt may need to be run as administrator to apply this last command successfully.

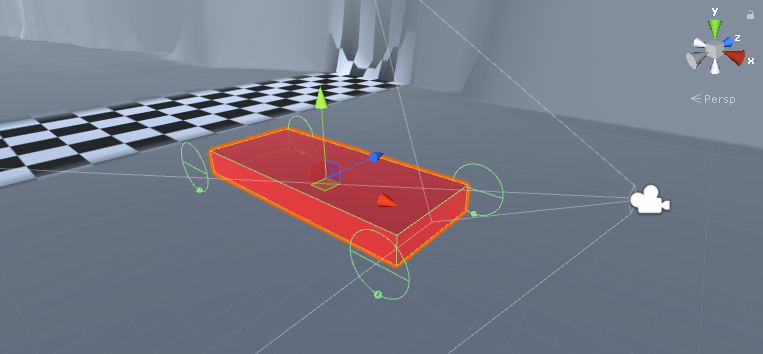
**Creating the Training Environment:**

To set-up Unity in order to work fully with the ml-agents package, Unity 2017 should be launched and a new project directory should be added, the Unity project is located in the ml-agents folder and is titled “unity-environment”. Once this project is launched the Tensorflow Sharp plug-in must be downloaded as a Unity package and added to the project, next navigate to the Edit -> Player Settings and add “ENABLE\_TENSORFLOW” into the Scripting Define Symbols for PC, this will allow external brains to be run within the Unity editor by using Tensorflow. Finally, navigate to Edit -> Project Settings -> Player and under Resolution and Presentation ensure that Run in Background is checked and Display Resolution Dialog is set to disabled.

A new Unity scene is now created, the first stage of creation is to create the environment that the agent will be trained in. This can range from a simple physics simulation to a full ecosystem. In this example, the environment is a very simple racetrack. A terrain object was added and, using the paint height tool, a rough track was carved out of the terrain, the result can be seen below:



The track above is a simple circle/oval shape as a complicated track would take far longer to train the agent. Next the car that will be driven by the agent must be created. First, a new empty game object is created and given an appropriate name, then a cube game object is created as a child to the empty game object. This cube is given the following scale: (X:4, Y:1, Z:10), four wheel colliders and a camera should then be made a child of this cube. The wheel colliders should be given appropriate names and placed in positions at the front right, front left, rear right and rear left of the car while the camera should be positioned slightly behind and above the car to ensure you can see the whole car through the camera, example below:

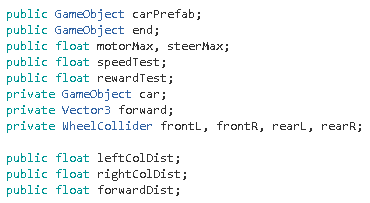


Save this game object as a prefab. Now the agent can be created and trained.

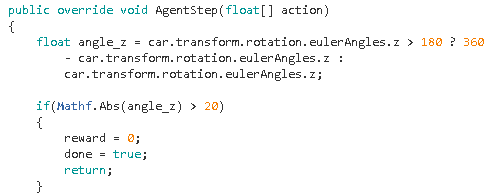
**Training:**

To train, an academy and brain must be added to the scene. This is done by adding an empty game object and naming it with the standard convention “your name” Academy and “your name” Brain1. Then add the template academy script to the academy object, renaming the script the same as the empty it is attached to. Then add the Brain script to the brain object, the variables of the brain will be changed later but the academy script will not change.

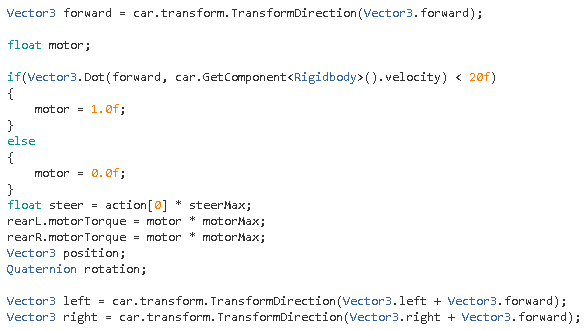
Next the agent script is created as a C# script, named “you name” Agent. The class type must also be ensured to be Agent and not MonoBehaviour. The initial variables to set up are shown below:



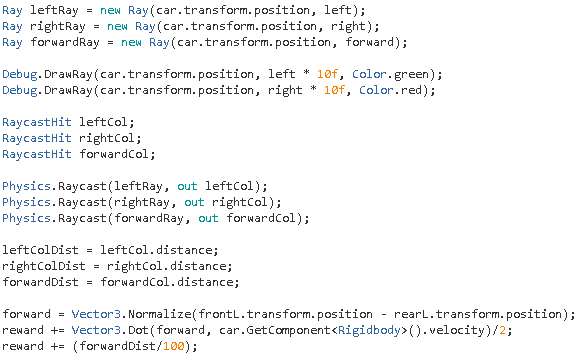
The initial game object is used to store the car prefab created earlier, the end game object is used to check if the agent has reached the end point of the track but as the track created is a loop this is not necessary. The floats for the maximum amount of steering and motor torque are then created, the next two floats were added to test the speed and reward of the car and can be ignored. Another game object for the car is then created and a vector 3 to hold the forward vector is created before creating the four wheel colliders to be used with the car prefab. Lastly three floats are created to store the distance of raycasts that will come out the car 45 degrees to the forward vector either side and on the forward vector, these are used to give the car sensors to enable it to tell if it is about to collide with the track.



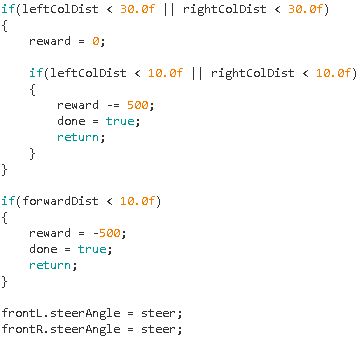
First to be created is the AgentStep method which takes a float as this is what is passed in by the external agent when training. Firstly a calculation is made to retrieve the angle of the car in the z axis and a check is made to ensure the angle is not over 20 degrees and if so the reward is set to zero and done is set to true before returning.



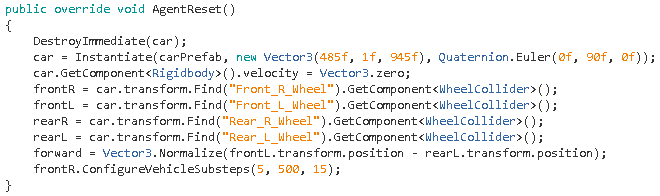
The forward vector is first collected before the motor float is created, a check is made to ensure the speed is below 20, if this is true the motor remains on and if not it is shut off, this is to ensure the car will not go too fast to control. Steering and the torque is then set before creating a vec3 and Quaternion to hold the position and rotation of the car. The left and right angle for the raycasts are then worked out.



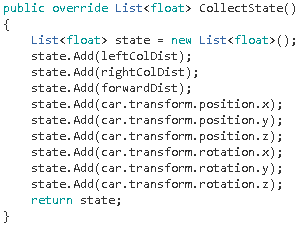
The rays are then fired, two were drawn for display purposes and the raycast hits are created to save the raycast information. The information is then retrieved and the distance from each ray to the edge of the track is collected. The forward velocity is then retrieved before rewarding the agent for its forward velocity as the agent must be encouraged to move forward. The agent is also rewarded on the forward distance it has as the agent should try to remain straight when on the track.



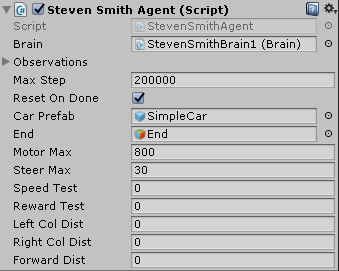
Checks are then made and the agent is given negative rewards if the car is too close to any edges of the track before finally the steering angle is set. The AgentReset method is shown below:



This destroys the car and repositions it at the beginning of the track, the vec3 and quaternion in the instantiation method should be changed to suit the track created when developing. The cars velocity is then set to zero and the wheel colliders are added before getting the forward vector and changing some variables for the vehicle to ensure it drives smoothly. This method is called in the AgentOnDone and InilializeAgent methods. Finally the states the agent will train with are supplied by using the CollectState method:



The states added are the distance of the colliders, the cars full position and full rotation. Now the agent script is constructed it should be added to an empty game object called “main” and all variables should be set as in the image below, all that is left is to set the brain variables and allow the agent to train in the environment created.

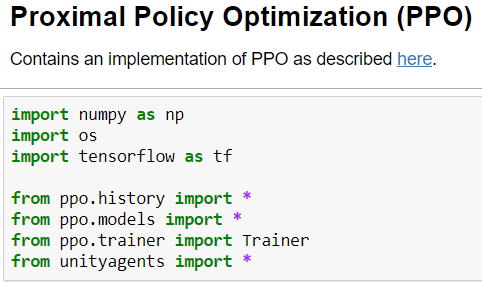


The brain variables should be changed to reflect the agent script, first the state size must be set to 9 as that is the number of states being sent to it, the action size is then 1 as it controls the steering. The action descriptions should then be set to 1 and the action space and state space should be set to continuous as it is a float being input before the brain type is set to external.

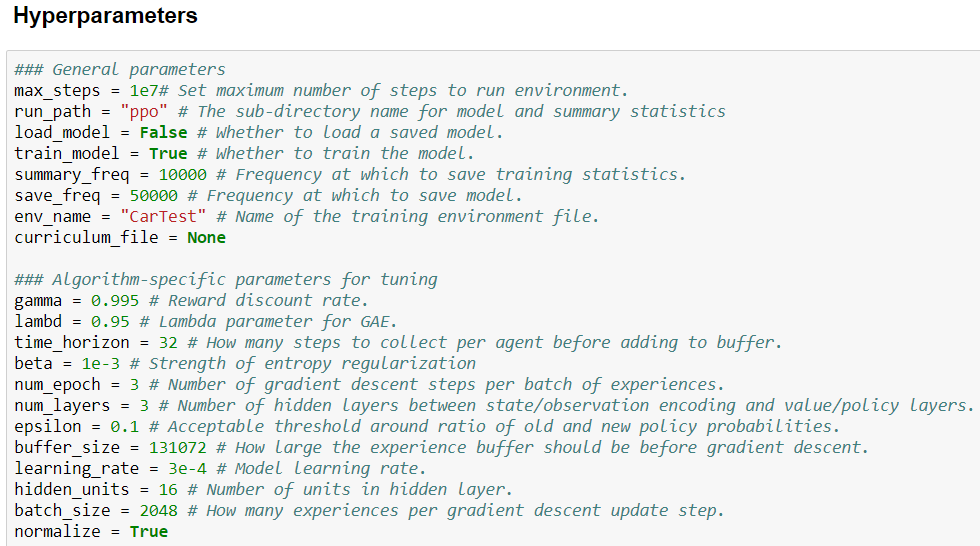
Finally, everything should be saved and built, the build file and executable should then be inserted into the python folder of ml-agents. Anaconda prompt should then be opened up before changing directory to the python folder and running the command “jupyter notebook”, this should open up a page as seen below:



The file PPO.ipynb should be opened and a page containing the PPO algorithms will open up, this page contains several sub-sections that are ran through by selecting the first one and pressing the “run” button and waiting for the specific section to finish running. The first sub-section of the PPO is used to import any external packages that are going to be used. This section can be seen below:



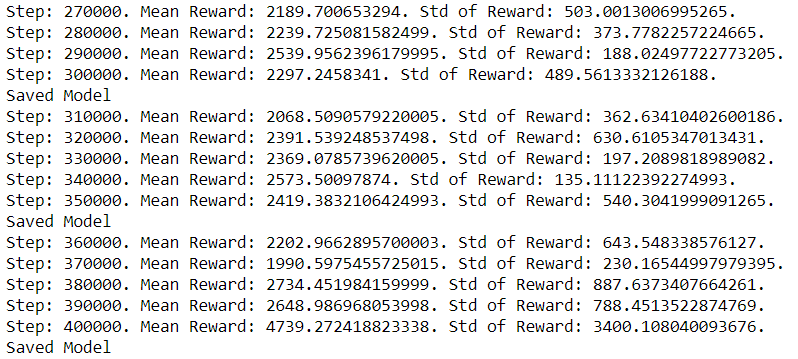
The next section is the hyperparameters, these are used to tune the training process to increase the possibility of successfully training a model. The hyperparameters can be seen below:



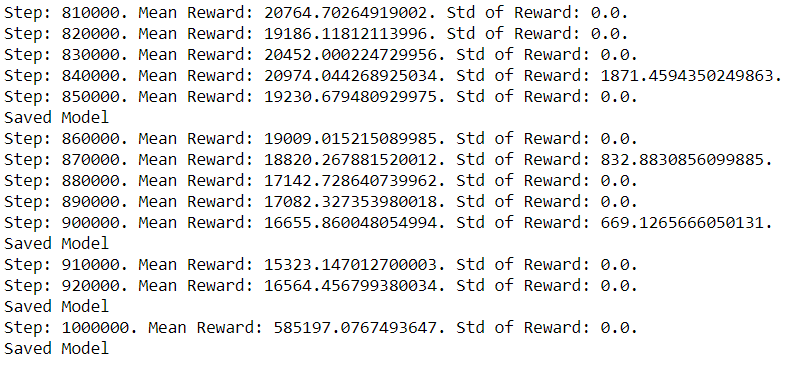
Max steps is used to set the maximum number of steps the simulation will run, this value should be increased for more complex problems. The run path supplies the directory to store the model created by training. Load model is used to set if a saved model is being loaded for training and train model tells if the model should be trained. Summary frequency is used to set after how many steps the models training statistics are saved while the save frequency states how many steps should run before the full model is saved. The environment name is then set, this should be the same as the executable that was built. These first parameters are used to edit the length of training and the frequency of saving mainly.

The next set of parameters is used specifically for tuning the algorithm as mentioned previously. Gamma corresponds to how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present to prepare for future rewards, this value should be large. Lambda corresponds to how much an agent will rely on its current value estimate when calculating its new value estimate. Time horizon corresponds to how many steps of experience to collect per agent before afdding it to the experience buffer. Beta corresponds to the randomness of the policy to ensure the agent will properly explore the action space. Number of epochs is the number of passes through the experience buffer during gradient descent, decreasing this will ensure more stable updates at the cost of slower learning. The number of layers corresponds directly to how many hidden layers are present within the neural network. Epsilon corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Buffer size equals how many experiences should be collected before the model is updated, typically larger buffer sizes lead to more stable training updates. Learning rate corresponds to the strength of each gradient descent update step, should be decreased if training is unstable. Hidden units correspond to how many units are in each fully connected layer of the neural network. Batch size is the number of experiences used for one iteration of a gradient descent update, should always be a fraction of the buffer size. Finally, normalize corresponds to whether normalization is applied to the vector observation inputs or not.

These hyperparameters had to be changed and tested a great many times before they began acting correctly and training began carrying out as hoped. Using the parameters seen above, training was carried out successfully and the agent learned how to navigate through the track. The next section of the PPO is to load the environment, if the environment name is set correctly and the executable build is in the correct folder this section will run and open a window of the training happening in a Unity window and update it every certain number of steps being carried out. The last section is the training of the agent, this section can be seen below:



This is near the start of the final successful training carried out, the step shows the step that is currently being carried out by the agent while the mean reward is the total reward the agent has acquired during the step updates. The standard of the reward is the average difference between the rewards the agent is receiving. The best result would be if the mean reward steadily increases while the standard of the reward decreases, this example shows a rather erratic standard which shows that the gamma should be reduced slightly in an attempt to stabilize the results. This training, however, did end up working as seen below:



The mean reward is very high here while the standard often equals zero which means that the agent has only one reward meaning that the agent has not been reset. Finally, the agent does not update for 80000 steps as the agent has not learned anything new, this is when the training was stopped as the agent can now fully navigate the created track. When the training is stopped, the model created is exported into the ml-agents>python>models>PPO folder as a bytes file, this file can then be used by going into the unity editor and setting the brain to internal and supplying it with the exported bytes file, this will allow the fully trained model to run within the unity editor and should show the car successfully navigating the track.

**Evaluation:**

When creating the machine learning agent, many simplifications had to initially be made such as locking the agents speed and changing the track to a loop, the hyperparameters and agent script were also changed many times until a working result was found. Through the large amount of testing, a lot of information on how reinforcement learning works was discovered and, while the training was fairly lengthy, the finished product has both increased my interest and knowledge in the field of reinforcement learning.

Unity’s ml-agents package along with other programs such as Tensorflow and Anaconda provide an engaging platform to learn, develop and test different machine learning techniques and as all programs used in this report are free to use, the development of these projects are extremely accessible and can be casually experimented with or used to create powerful tools for entire videogame worlds. In conclusion, Unity’s ml-agents provided grounds to help create the autonomous car seen in this report, which would not have been possible without the packages and documentation from Unity itself and could hopefully encourage a younger generation into creating engaging environments to create and test machine learning agents within.

**References:**

Ertel, W., 2018. *Introduction to artificial intelligence*. Springer, p.1

Moffat, D.C. and Romano, D.M., Artificial Intelligence and Games.

Yannakakis, G.N. and Togelius, J., 2017. Artificial Intelligence and Games (Second Public Draft). p. 16

Gold, A., 2005, April. Academic AI and Videogames: A Case Study of Incorporating Innovative Academic Research Into a Videogame Prototype. In *CIG*.

Gautam Narula. 2017. Machine Learning in Gaming – Building AIs to Conquer Virtual Worlds. [ONLINE] Available at: <https://www.techemergence.com/machine-learning-in-gaming-building-ais-to-conquer-virtual-worlds/>. [Accessed 29 March 2018].

**Bibliography:**

Unity: <https://unity3d.com/get-unity/download/archive>

CUDA: <https://developer.nvidia.com/cuda-toolkit-archive>

cudNN: <https://developer.nvidia.com/rdp/cudnn-archive>

Anaconda: <https://www.anaconda.com/download/>

Unity’s ml-agents: <https://github.com/Unity-Technologies/ml-agents>

Tensorflow Sharp: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Using-TensorFlow-Sharp-in-Unity.md>

Unity Documentation: <https://github.com/Unity-Technologies/ml-agents/tree/master/docs>

**Questions:**

Question 1: Which of the following is not a rule of flocking?

A-Alignment

B-Separation

C-Departure

D-Cohesion

Answer: C - Alignment, cohesion and separation are all behaviours that are used within flocking while departure is not.

Question 2: Machine-learning can only be applied to a game where all data is known by the learner?

A-True

B-False

Answer: B - Machine-Learning can be applied to games where not all data is known to the learner.

Question 3: What can procedural content generation be used for?

A-Creating textures and 3D models

B-Creating a games randomized “loot”

C-Creating random maps for a game

D-All of the above

Answer: D - PCG can be used to generate a wide variety of things from textures to full 3D maps for use in video games.

Question 4: Which sorting algorithm would be best applied to a large amount of data?

A-Insertion sort

B-Selection sort

C-Merge sort

Answer: C - Insertion and selection sort are simple and lose efficiency when paired with large amounts of data.

Question 5: Which of the following is NOT a path-finding algorithm?

A-A\*

B-D\*

C-Q-Learning

D-Dijkstra’s

Answer: C - Q-Learning is a reinforcement learning technique and not a path-finding algorithm.

Question 6: Which of the following could cause a machine-learning model to fail?

A-High amount of false positives

B-Overly complex models

C-Unstable models

D-All of the above

Answer: D - Unstable models can lead to unpredictable timings and high instability. Overly complex models can be used effectively on test data but can prove too difficult to implement into other scenarios. False positives can ruin the accuracy of a model.

Question 7: Which of the following statements on proximal policy optimization is untrue?

A-Proximal policy optimization is simpler to implement than other state-of-the-art algorithms

B-Proximal policy optimization lacks the performance of other state-of-the-art algorithms

C-Proximal policy optimization requires substantial tuning to get optimal results

Answer: B – Proximal policy optimization can perform comparably to other approaches

Question 8: In which of the following methods are rewards and punishments used on an agent?

A-Active Learning

B-Supervised Learning

C-Unsupervised Learning

D-Reinforcement Learning

Answer: D - Reinforcement learning is a technique that provides an agent with rewards and punishments to stimulate learning.

Question 9: Which of the following is also known as exploratory learning?

A-Supervised Learning

B-Active Learning

C-Unsupervised Learning

D-Reinforcement Learning

Answer: C - Unsupervised learning is also known as exploratory learning as there is no teacher for these algorithms, therefore, "unsupervised".

Question 10: Two types of game data for the training of AI exist, perfect and imperfect data, which type would a game of chess represent?

A-Perfect

B-Imperfect

C-Neither

D-Both

Answer: A - A game of chess would be a representation of perfect data as all states such as the state of the board are known to the player, there are no hidden factors.