AI Self-Driving Car

Shubh Ravishankar Gawhade

MSc Computer Games Technology, 2023-24

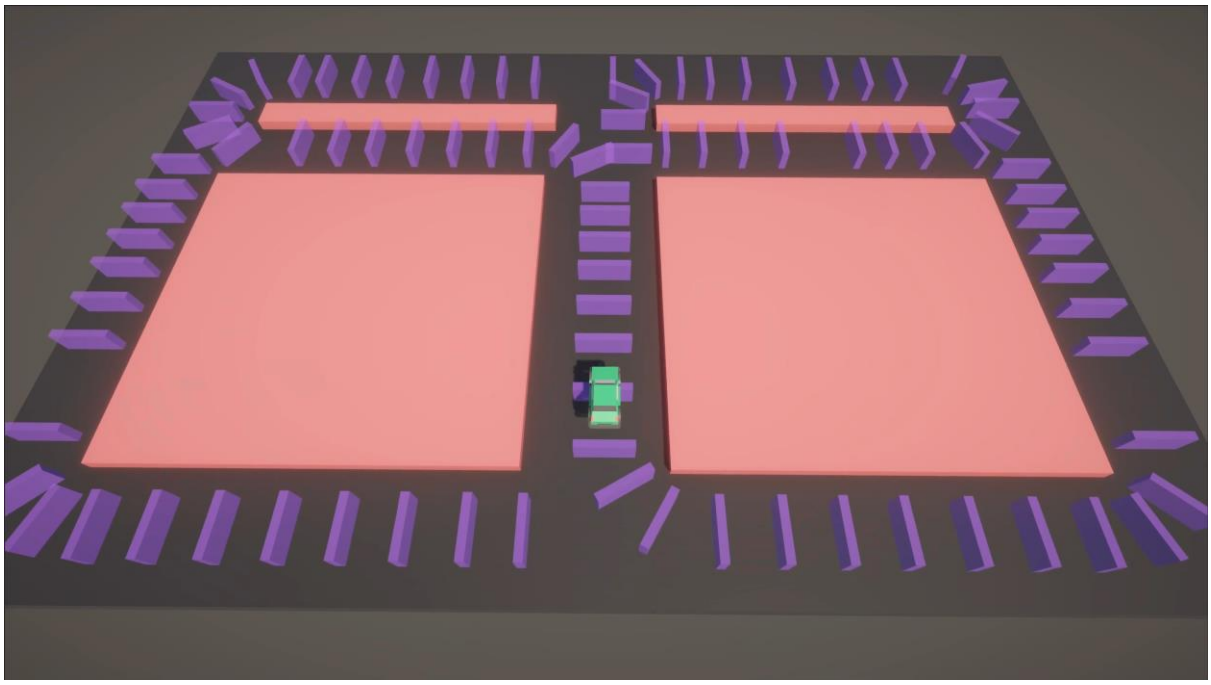

Figure 1: Self-Driving Agent Screenshot

School of Design and Informatics

Abertay University

Contents

List of Figures

Summary

To be able to choose from the wide array of machine learning algorithms, we first need to understand the different classes of machine learning algorithms. There are three main classes namely unsupervised learning, supervised learning, and reinforcement learning. Each class uses different types of data to learn from and have their own use cases in the real world. [1]

Unsupervised learning: Algorithms which use this method do not use input labels but are given raw data and are expected to find patterns and groups within the data. It is very helpful in data-driven tasks where large amounts of data have to be sorted to form deductions. Some applications include categorizing in search engines, medical imaging, fraud detection and user data analytics. Some algorithms which come under this class include k-means, clustering algorithms, and association algorithms.

Supervised learning: These algorithms use input labels to guide the learning process by comparing predictions to actual values and learning from different data sets to produce accurate predictions. They usually call under two categories, regression, where the output is a real value or data type and classification where the output is a category. There are many algorithms which fall into either category such as linear regression, SVM, k-nearest neighbours, decision trees and naive Bayes. Some applications of these algorithms are object recognition, customer analysis and predictive analytics. [2]

Reinforcement learning: They are similar to supervised learning in the sense that they receive feedback for their actions but not necessarily for every input or state. These algorithms randomly explore different states and attempt to learn actions that reward them by giving a reinforcement signal that it has reached a certain goal state. It is analogous to Human learning where feedback is given when a reward is warranted. They include algorithms which are similar to Q learning but with different characteristics but follow a similar approach but tweak different variables in the algorithm to include probability and curiosity. They are ideal for making decisions in an uncertain environment.

Deep learning: These are a family of algorithms that can be used to address complex problems. They've gained popularity in the past few years for their ability to learn complex tasks from large amounts of training data. It can be used for tasks such as image recognition, natural language processing and dish recognition. These architectures include deep neural networks, convolutional neural networks, and recurrent neural networks to name a few.

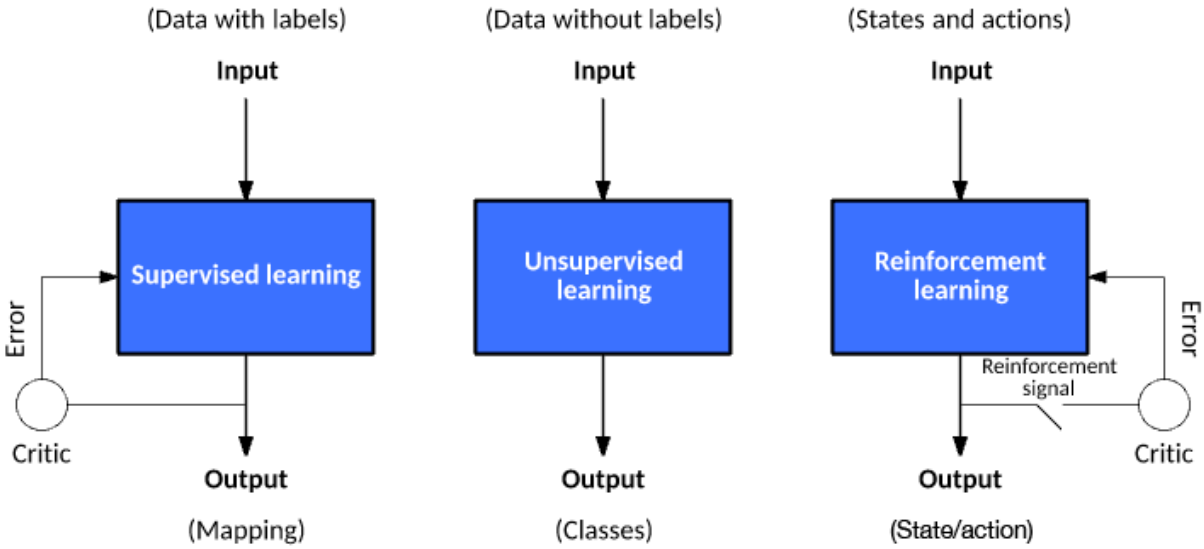

Figure 2: ***Three learning models for Algorithms***

Reinforcement Learning

I planned on implementing a self-driving car which drives to a random destination in a simulated environment while avoiding stationary and non-stationary obstacles. This in itself is quite a complex task as the ai has multiple factors to account for. Using the above classification, reinforcement learning best fits our use case. It can handle complex and dynamic situations such as random obstacles and unknown destinations which change every run.

Reinforcement learning has its roots in animal psychology: learning by trial and error. Early researchers thought that this mechanism could be implemented within machines to learn to map states with actions.[3] Marin Minsky built the earliest example of reinforcement learning in 1951 to mimic a rat learning how to solve a maze (implemented with vacuum tubes that represented the 40 neurons of the simulated rat brain). As the robotic rat solved the maze, the synapses were reinforced based on its ability to escape. This method did not see any success until 40 years later an IBM researcher, Gerald Tesauro developed 'TD-Gammon' that learnt to play backgammon using temporal difference learning to train a neural network with 80 hidden units. The AI learnt the game without any prior knowledge and identified new strategies to play the game showing the possibilities of problem solving with reinforcement learning. Reinforcement learning algorithms are generally a variant of Q-learning algorithms. [4] This algorithm incorporates Q values for every state-action pair which indicates the reward for following a state path and is an estimate of the expected future reward for taking an action in a state. Each state encompasses the actions which lead to better states and finally the goal. These values are updated in every state on performing an action. Actions are performed by probabilistically selecting them based on learning rate, discount factors and curiosity which allows the exploration of the state-action space. The Q value function calculates the Q value by considering the immediate reward and the discounted future reward for taking an action in a state. It is calculated by the reward provided by the move along with the maximum Q value available for the new state by applying the action to the current state discounted by a discount factor. This value is further discounted by a learning rate that determines if the information is more valuable than the old ones. A higher discount factor shows a preference for long-term rewards over short-term ones. Q values are updated over several epochs to reach the goal state by leveraging the probabilistic selection of an action for new states. After the training is completed, the actions with the largest Q values produce optimal results. Other algorithms have slightly different characteristics such as the selected action might not always be the one with the highest Q value. However, the underlying concept remains the same.

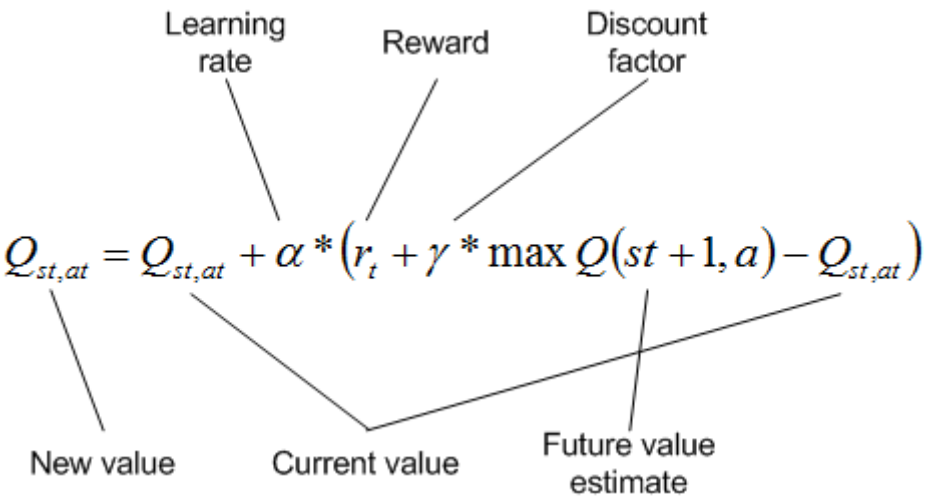

Figure 3: ***Typical Q-learning algorithm***

ML-Agents

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. It provides a variety of training models to choose from such as reinforcement learning, imitation learning, neuro-evolution, curriculum learning and even external custom models. [5] The concept in theory is quite simple. To train an agent, three entities need to be defined at every moment in the environment during training. They are observations: these can be numeric and or visual. Numeric observations of attributes in the environment from the agent's point of view. They can be either discrete or continuous depending on the complexity and functions of the agent and environment. Visual observations are images captured using a camera attached to the agent to view the environment at that point in time. Actions: They can also be continuous, discrete or a mix of both depending on the complexity of the agent's actions. They help control the agents by giving them actions They can perform. Reward signals: This is a scalar value which doesn't need to be provided at every step but can be to train very specific behaviour. Usually, they are provided when the agent does something good or bad. Maximizing rewards generated the desired optimal behaviour and generally should be ranged between -1 and +1. ML-agents have many built-in implementations of state-of-the-art algorithms. It provides the following training modes:

Built-in training and inference- This mode is based on tensor flow and uses observations from the agents to learn a tensor flow model which is embedded in the brain for inference. During training, the observations are sent to an external brain using the Python API which processes them and sends back actions for the agent to take. After training, during inference, the brain is then switched to internal which uses the embedded tensor flow model to generate actions based on observations.

Curriculum learning- This is an extension of the built-in training and inference model which is helpful to train complex behaviour by breaking the task into smaller pieces starting with easier tasks as the agent will not likely perform a complex task from the beginning. It's very similar to how humans learn by tackling small problems first and using that policy to build upon and learn more. When agents pass a certain threshold, they are moved to the next stage with increased complexity.

Imitation learning- Much like a human viewing a tutorial before performing a task, the agent will have a better idea of what actions it needs to perform to accumulate more rewards by observing the actions and forming a policy rather than a trial-and-error method.

The package also provides other tweaks to existing training modes such as single agent, self-play, memory enhanced, recurrent neural networks and multi-agent scenarios allowing for a wider array of problem-solving capabilities.

Proximal Policy Approximation (PPO)

Proximal Policy Approximation (PPO) uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take at a given state. OpenAI uses PPO as their default reinforcement learning algorithm and gives a reason that using policy gradient methods is challenging as they're very sensitive to the parameters given such as a smaller step size might make the learning too slow, too large and the agent is overwhelmed by the noise. Using supervised learning ensures excellent results with lesser hyper-parameter tuning by the implementation of the cost function and running gradient descent on it. It strikes a balance between ear-off implementation, tuning and sample complexity while ensuring that the deviation from the previous policy is relatively small. OpenAI boasts about their new novel objective function not found in other algorithms. [6][7]

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta))\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)]$$

θ is the policy parameter

$\hat{E}_t$ denotes the empirical expectation over timesteps

r_t is the ratio of the probability under the new and old policies, respectively

$\hat{A}_t$ is the estimated advantage at time t

ε is a hyperparameter, usually 0.1 or 0.2

PPO is a policy-gradient algorithm that belongs to the reinforcement learning family. It trains a neural network to represent the agent's policy, which is the function that maps the agent's observations to actions. PPO updates the policy by maximizing a surrogate objective function that encourages the policy to improve while staying close to the previous policy. PPO is a good choice for self-driving car applications because it is simple, stable, and sample-efficient. PPO is simple because it does not require complex components, such as trust regions, off-policy corrections, or replay buffers, that are used in other RL algorithms. PPO is stable because it avoids large policy changes that could harm the agent's performance or cause instability. PPO is sample-efficient because it can reuse the same data multiple times to update the policy, reducing the amount of data needed to learn.

Using PPO in a simulated environment can help accelerate the learning process and reduce the risks and costs associated with real-world testing. The BMW Group used Unity to develop a graphical scenario editor that vastly simplifies the process of testing and validating automated driving (AD) features in development. [8] Nearly 95% of all BMW's test miles for autonomous driving are driven by virtual vehicles in virtual worlds. A simulated environment can provide a realistic and diverse set of scenarios for the agent to learn from, such as different destinations, obstacles, and traffic patterns. A simulated environment can also allow for easy evaluation and comparison of different policies and parameters. Therefore, using PPO in a simulated environment is a great approach for implementing a self-driving car that drives to an unknown destination while avoiding obstacles.

The task is to train a self-driving car which drives to an unknown destination in a simulated environment while avoiding stationary and non-stationary obstacles. A mathematical definition of RL for this task can be defined as follows: [9]

Let there be a set of agents $A = a_1, a_2 \ldots, a_n$, where n is the total number of agents. Each agent a_i is a self-driving car that interacts with the environment in a sequence of discrete time steps t=1, 2…, T. At each time step t, each agent a_i takes an action $a_{i,t}$ from a set of actions A_i, which include acceleration, braking and steering and the environment transitions to a new state s_{t+1} and provides each agent with a scalar reward $r_{i,t}$. The agent's goal is to learn a policy $\pi_i(a_{i,t}|s_t)$, which is a probability distribution over actions given the current state, that maximizes the expected cumulative reward over time, also known as the return, defined as:

$$J(\pi_i) = \mathbb{E}\pi_i\left[\sum t = 0^{T-1}\gamma^t r_{i,t}\right]$$

where $\gamma \in [0,1]$ is a discount factor that determines the importance of future rewards and $r_{i,t}$ is the reward associated with reaching the destination or collecting rewards along the path by avoiding collisions and penalties. This can also be defined by considering the state value function $V_{\pi i}(s_t)$ which represents the expected time to reach a destination starting from state s_t and following policy π_i:

$$V_{\pi_i}(s_t) = \mathbb{E}\pi_i\left[\sum k = 0^{T-t-1}\gamma^k r_{i,t+k}\middle|s_t\right]$$

and the action-value function $Q_{\pi i}(s_t, a_{i,t})$ which represents the expected time to reach the objective starting from state s_t, taking action $a_{i,t}$ and following policy π_i:

$$Q_{\pi_i}(s_t, a_{i,t}) = \mathbb{E}\pi_i\left[\sum k = 0^{T-t-1}\gamma^k r_{i,t+k}\middle|s_t, a_{i,t}\right]$$

ML agents provide the reinforcement learning technique Proximal Policy Approximation (PPO). PPO is the default algorithm for ML-Agents, and it has many hyper-parameters that can be tuned to affect the performance of the trained agents as desired. Some of these parameters are Gamma: This is the discount factor for future rewards which is the agent's likeliness to act for immediate or future rewards. Lambda: The parameter which controls how much an agent relies on the current value estimate which can be high bias or on the actual rewards received which can be high variance to provide a stable learning process. Buffer size: This is the number of experiences that should be collected before learning or updating the model. Batch size: The number of experiences used for one iteration of a gradient descent update. Time horizon: This is the number of experience steps to collect per agent before adding it to the experience buffer. Beta: This controls the strength of entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Epsilon: This is the acceptable threshold of divergence between old and new policies and controls the stability of training. [10]

AI Training

I started by developing a car that was controllable by a player in the unity environment which took in input values for forward movement to accelerate and decelerate, and sideways movement for turning. After setting up the ml-agents package, I used the inbuilt methods from it for the agent to send these inputs instead of a player. Instead of using the curriculum learning provided by ml-agents, I decided to create my curriculum by training the agent with simple tasks to begin with and increasing the difficulty when a certain “loss” threshold was met along with a low standard deviation which showed that most of the agents had learnt the behaviour. Starting with a simple plane without any obstacles and a target destination for the agent to drive to. The observation vectors provided at this stage were the agent's local position and the target's/destination's local position. A reward of +1 was given if the agent reached the target, a very small negative reward for each time-step so that the agent would be motivated to move. After some training, it was observed that the agent loved to go in reverse and often fell off the edge of the plane. To prevent such unwanted cases, a reward of -1 was given if the agents fell, and a very small positive reward for accelerating to make it to forwards. Using just these simple inputs, the agent took more time to learn than expected by occasionally missing the target. To make the agent more aware of what it was supposed to be doing, I added a small negative reward multiplied by the distance between them for the agent to move towards it. This worked well and the agent made fewer mistakes. As seen in the resulting training graphs below, the cumulative rewards stabilized along with the episode length when it reached the destinations.

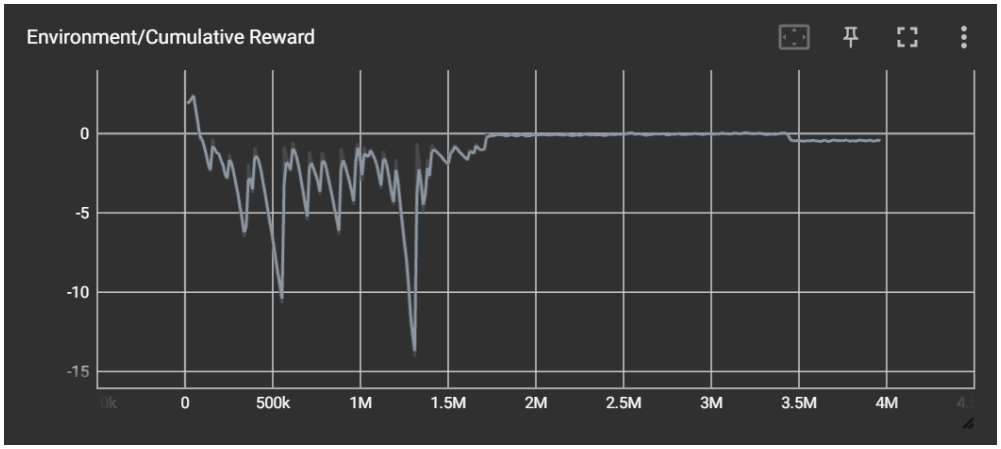

Figure 4: ***Reach a destination on an open plane*** *- Cumulative reward graph*

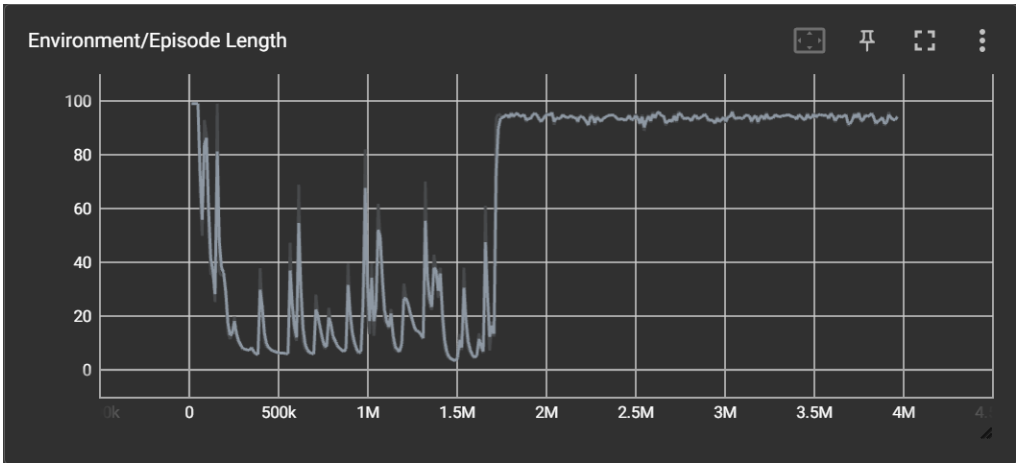

Figure 5: ***Reach a destination on an open plane*** *- Episode Length graph*

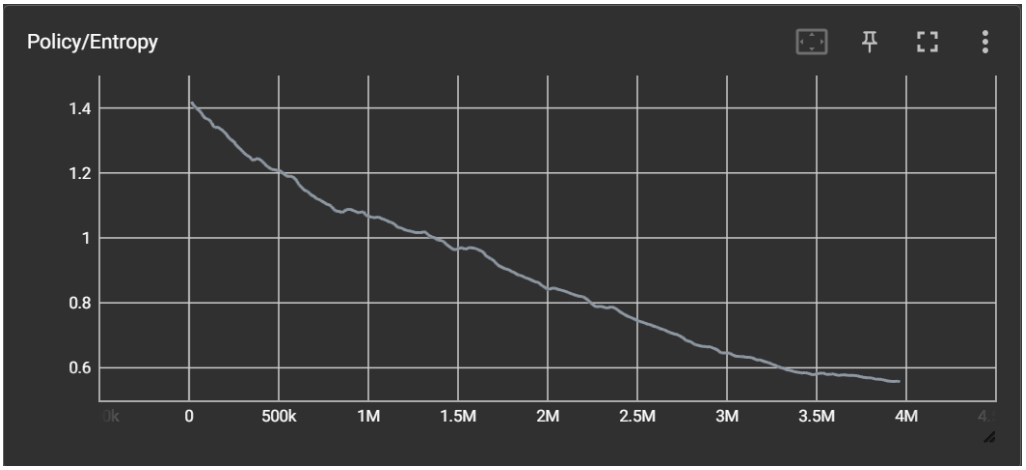

Figure 6: ***Reach a destination on an open plane*** *- Policy/Entropy graph*

Now that the AI had learnt to move and go to a destination, I wanted it to be able to avoid obstacles and other agents. After adding a negative reward and ending the training episode if any collision occurred, I started training the agents. After a while, I noticed that the AI loved hitting walls and surprisingly the reward values were still getting better. I realized that they were showcasing this behaviour to avoid the constant negative reward for every time step. To correct this behaviour, I reversed the reward per time step and made it positive so that the AI would want to survive for longer. This helped, and they started avoiding each other and the obstacles. However, it introduced another issue where the AI found an optimal path to go around the target so that the negative distance and positive time step balanced out for it to keep accumulating more rewards by maximizing the episode time instead of ending the session by hitting the destination at first and then and as seen in the resulting graph, it kept hitting into obstacles so that it would end the episode early without incurring any negative rewards thus increasing the rewards.

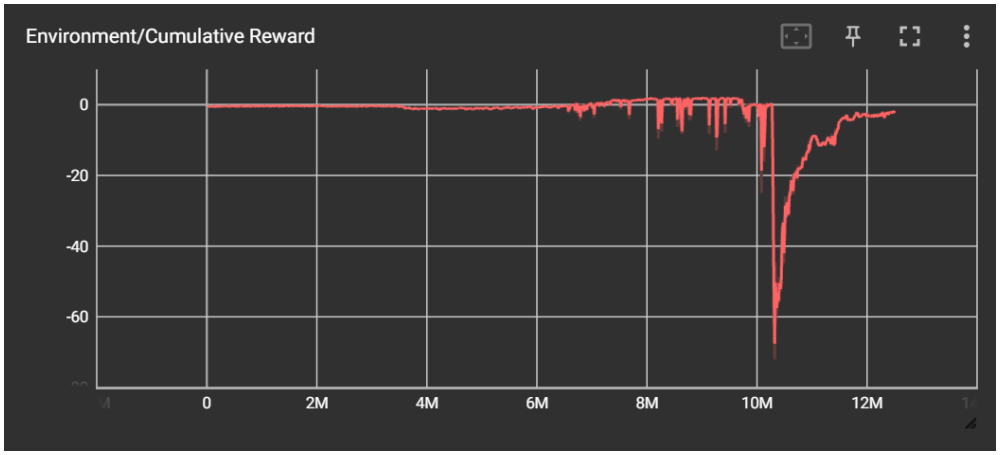

Figure 7: ***Reach a destination while avoiding obstacles on a plane*** *- Cumulative reward graph*

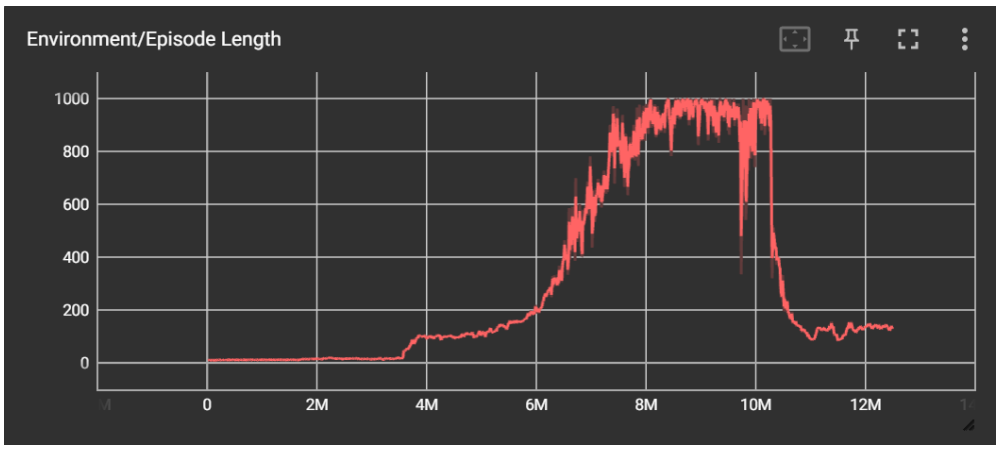

Figure 8: ***Reach a destination while avoiding obstacles on a plane*** *- Episode Length graph*

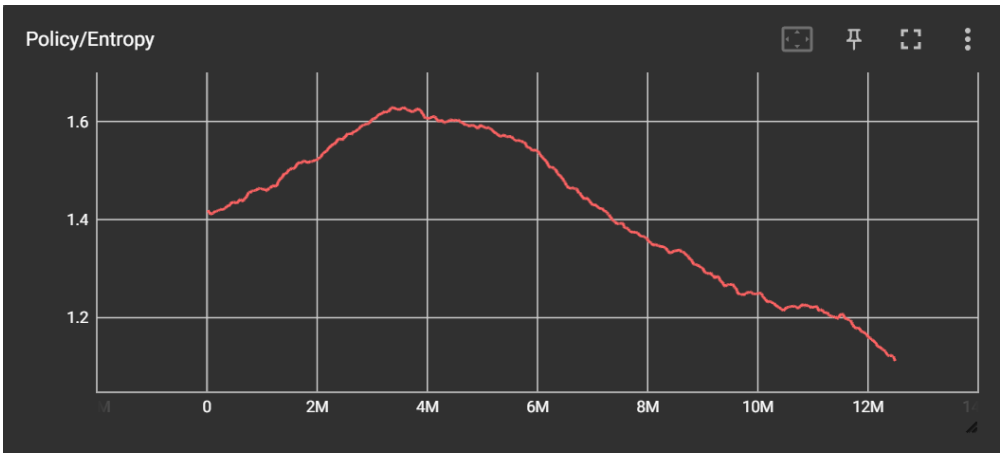

Figure 9: ***Reach a destination while avoiding obstacles on a plane*** *- Policy/Entropy graph*

My next step was to add checkpoints which the agent could follow a path and collect all the checkpoints. A small reward of +0.2 was added for each checkpoint collected and the

checkpoint was added to a list that doesn't allow collecting the same checkpoint. Another observation vector was added which kept track of the number of checkpoints collected. They were trained with a simple straight pattern with a destination at the end at first which was easy to follow if the AI was closer to the first checkpoint. On changing the training to work with random start locations, they performed considerably worse as it was hard to know the locations of the checkpoints. To give the agent more information about its environment, I added a Ray Perception sensor which detects the tag "Checkpoints", and I added the tag to the checkpoints. The agent performed better than before by finding closer checkpoints and starting to collect them. However, it failed to give a low standard deviation since it couldn't follow the checkpoints in each pattern. I proceeded to create a harder pattern which involved branching paths and needed the agent to take a U-Turn from the end and take another route. A reward of +1 was given if the agent collected all the available checkpoints. Firstly, training the agent from the same start location gives great results with them following the pattern, taking a U-Turn, and collecting the remaining checkpoints. However, when random start locations were introduced, again the agent found it difficult. The agent found an optimal by balancing negative and positive rewards and went around in an ellipse, maximizing the episode time as seen in the resulting training graphs below.

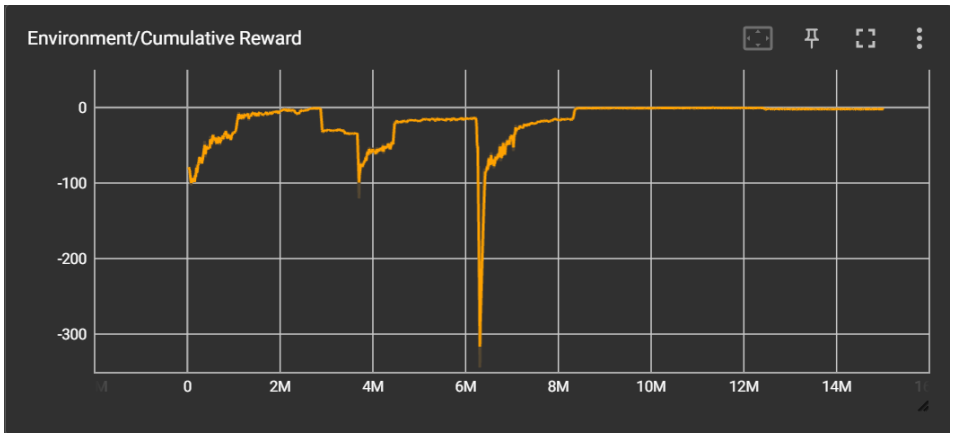

Figure 10: ***Follow a pattern to the destination*** *- Cumulative reward graph*

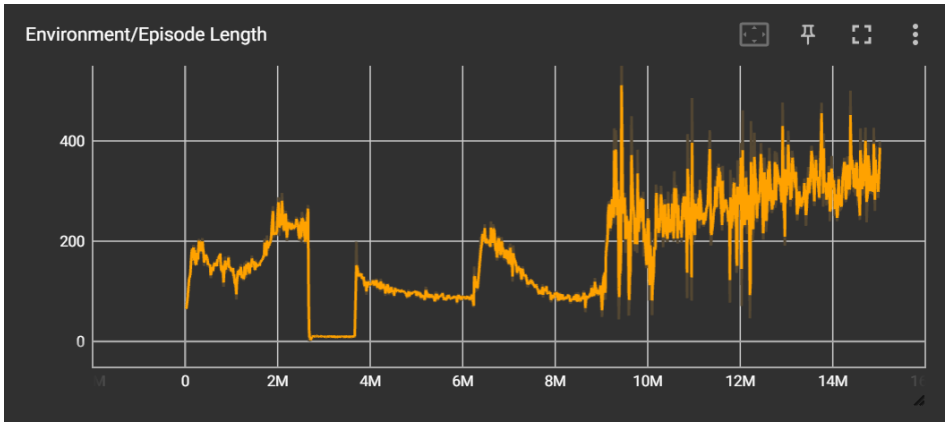

Figure 11: ***Follow a pattern to the destination*** *- Episode Length graph*

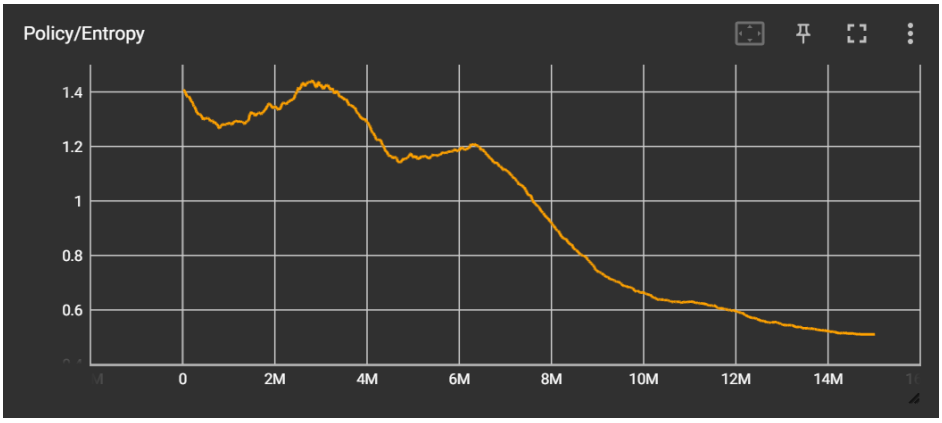

Figure 12: ***Follow a pattern to the destination*** *- Policy/Entropy graph*

Since the agent found optimal paths on an open plane, I decided to convert it to a more constrained environment like a crossroad using a 3d modelling software called “blender”, with the sides as obstacles. The agent was limited to fewer paths that it could take to destinations that lie on them. The rewards were structured such that the agent got a very small positive reward for every time step it was alive, a very small negative reward multiplied by the distance between the agent and the destination so that the agent will try to minimize the negative reward by reducing the distance and added the distance variable to the observation vector. A reward of +1 for reaching the destination and -1 if it hits any obstacles. Training the agent with these policies made it very tough for the AI to find the destination especially due to the layout in which it tried finding the shortest distance which would need the agent to sometimes go further away from the target to turn towards the destination. If the target faced the destination or the shortest route didn’t include going away from the destination, the agent performed better by easily reaching the destination. To make the agent able to go to random destinations, I decided to include another Ray Perception sensor which performed the task of finding the direction of the destination by ignoring the obstacles. Rewards were added based on the dot product between the direction of the car agent and the destination based on how close the target is. The other sensor would account for the distance only when the destination was in view. Training with these changes made the agent better at finding the direction of the target but didn’t change much. I realized that this was more of a path-finding issue therefore, I brought back the checkpoint system around corners and path changes along with adding a third ray sensor which only detects checkpoints. Additionally, I added the reward system to account for checkpoints and for the agent to move forward only. This worked by making the agents follow a path till the checkpoint was in view and then adding the rewards for the distance between. The only issue was that the driving looked scattered, and the agent went anywhere and randomly in the middle to try to get to the destination quickly. To “teach” the agent to drive on the correct sides of the road, I added a divider obstacle to the street model and added checkpoints for the agent to always follow. Instead of adding a small reward when passing through checkpoints, I added a direction for each checkpoint and the agent is rewarded positively if it passes from the same direction along with a small error margin and a reward of -1 if it triggered it from the wrong direction indicating that it was on the other side of the

street. This too didn't yield good results as the agent was unable to go to the destinations behind it or where it had to turn around to face the right direction. To try and solve this, I added a small negative reward multiplied by the distance from the checkpoint ahead to make the agent follow the checkpoints even though it led away from the destination. The hyperparameter "buffer_size" was also increased for training. This partly worked, but also created another issue where to stay "alive" for a longer time, the agent would go front and back in the same spot. To prevent this, I started a timer when the velocity of the agent was below a certain threshold and if it stayed below it for more than a certain time, a reward of -1 was awarded. This partly worked again as it was unable to do the same for random destinations and if it was spawned in random locations. Either the policy needed to be described better or the agent needed to be trained for longer. After attempting to solve for more than just one U-turn, I realized that it would be easier to just change the environment again and make it a bit simpler for navigation. In the graphs below, the first 3 sets are the result of training the agent without following lane or side rules. The other 3 sets are the result of training the agent with checkpoints with designated directions to pass through them.

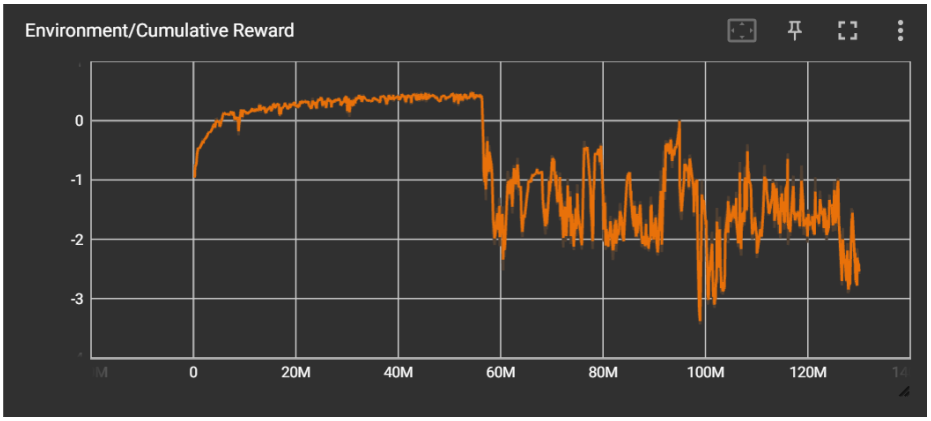

Figure 13: ***Crossroad without lane directions*** *- Cumulative reward graph*

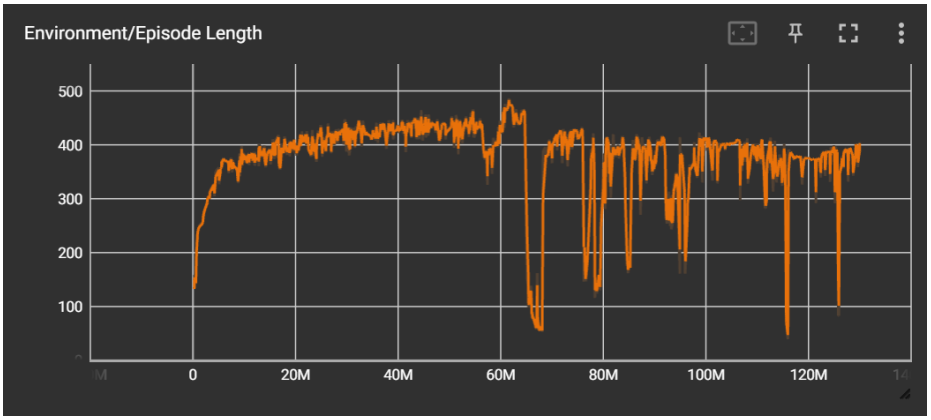

Figure 14: ***Crossroad without lane directions*** *- Episode Length graph*

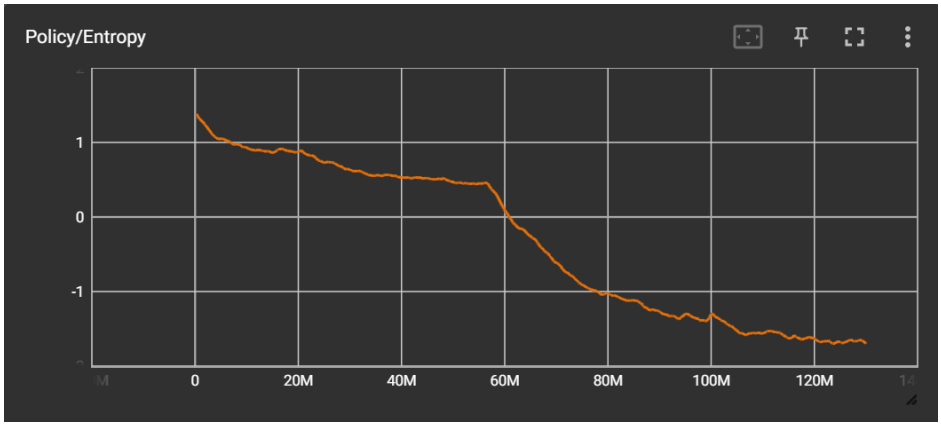

Figure 15: ***Crossroad without lane directions*** *- Policy/Entropy graph*

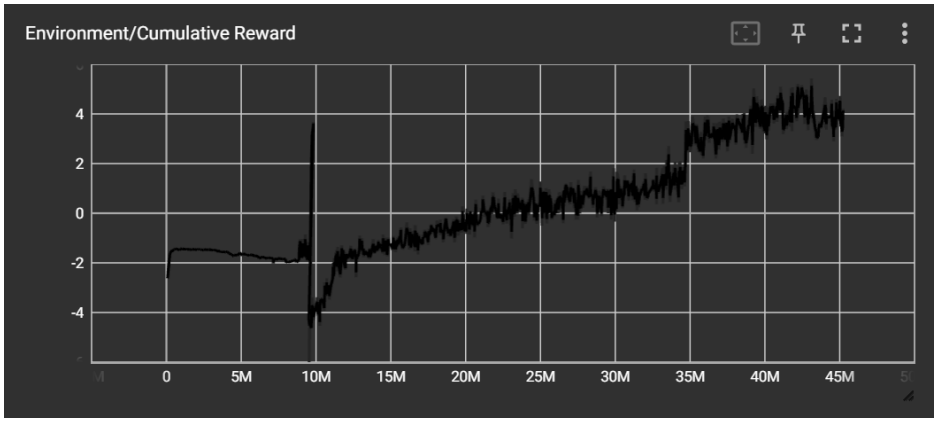

Figure 16: ***Crossroad with lane directions*** *- Cumulative reward graph*

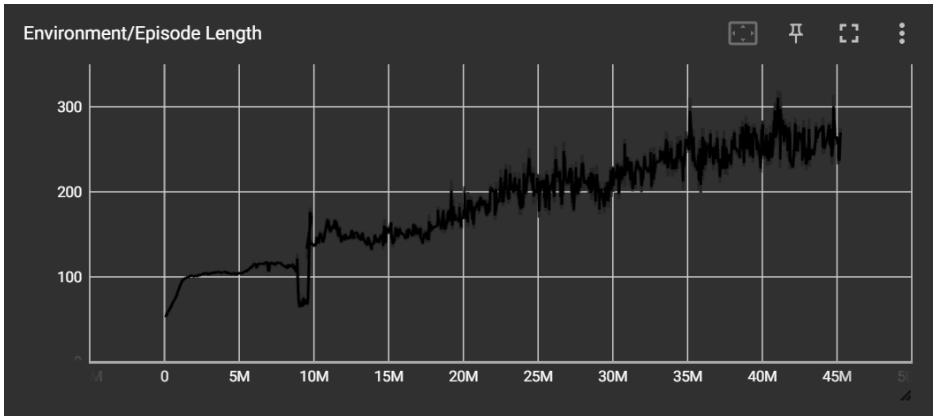

Figure 17: ***Crossroad with lane directions*** *- Episode Length graph*

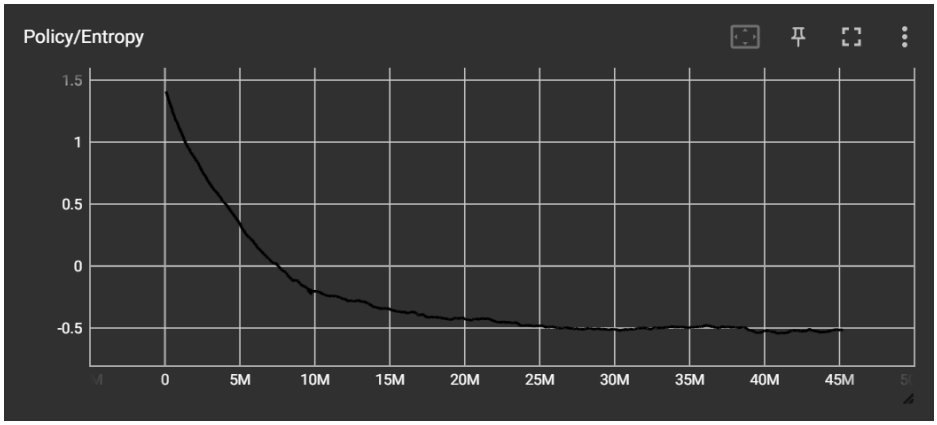

Figure 18: ***Crossroad with lane directions*** *- Policy/Entropy graph*

After repeatedly trying and failing to get the agent to follow lanes and switch between random destinations, I decided to simplify the environment and make an intersection that involved three one-way roads, one of which creates an intersection with a 2-lane street. I used a similar setup of checkpoints marking the different paths with the direction dictating the driving direction with a modification of removing the extra sensors and leaving one to detect the checkpoints and obstacles. The reward structure was similar to before where the closest checkpoint was factored till the direction of the agent and destination matched. On training, the agent was able to take simple turns and even reach the destinations if they were simple to achieve. However, I wanted it to be able to take U-Turns and alternate between the destinations randomly to create a traffic-like setting. I removed the observation vector which kept track of the distance between the agent and the destination. The reward structure was kept the same except for one important change related to the checkpoints. Information about the checkpoints captured from the ray sensor was compared to a stored value to check if it was closer to the destination than the previous checkpoint and was set at the closest checkpoint the agent had to travel to. Distance-based rewards were given on whether the agent was facing toward the destination else the closest checkpoint. Some of the hyperparameter configurations that were changed included an increase to the "batch_size", "num_epoch" and "time_horizon" parameters. On training, the agent learnt how to take a U-Turn and go to more complex single destinations. The "time_horizon" parameter helped the agent to make decisions based on a longer time frame. To expand this capability, I added another vector observation which considers the score which was incremented based on the number of destinations reached. After training with randomized destinations, the results were considerably better as the agent learnt to travel between multiple checkpoints and achieved a simple traffic system.

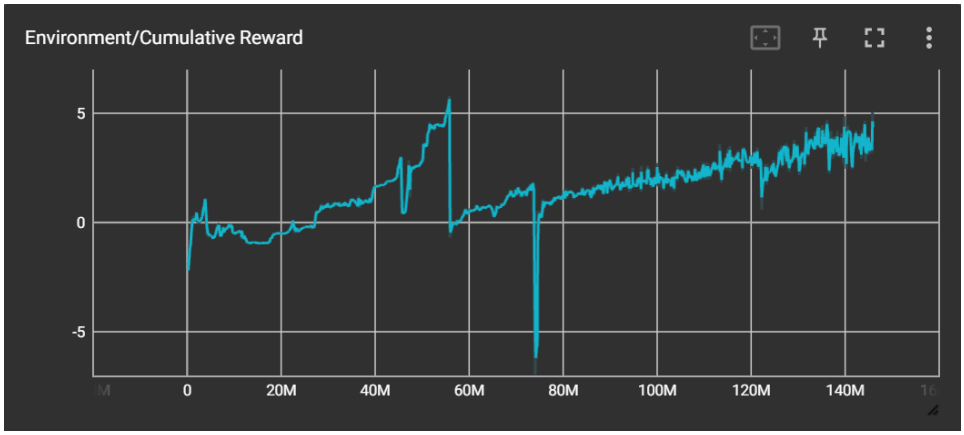

Figure 19: ***Intersection with random destinations*** *- Cumulative reward graph*

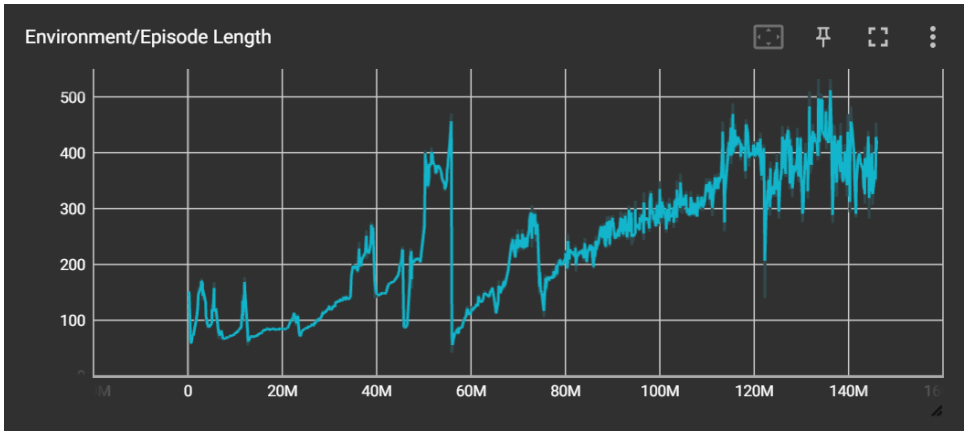

Figure 20: ***Intersection with random destinations*** *- Episode Length graph*

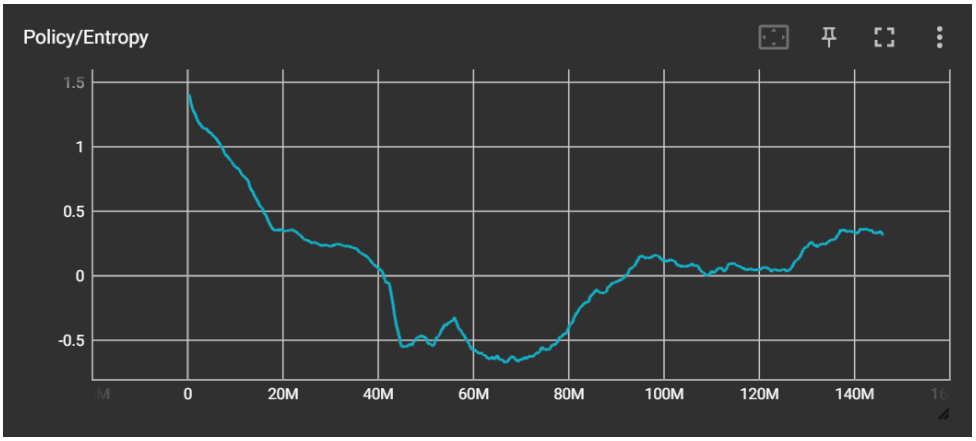

Figure 21: ***Intersection with random destinations*** *- Policy/Entropy graph*

The above training sessions explore the use of the ml-agents PPO algorithm to train a self-driving car for a simulated environment, and the processes and parameters involved that affected the training process. Although the results are not close to how traffic would behave in the real world. It still is a stepping-stone toward the capabilities and possibilities of problem-solving by the algorithm.

References

[1] IBM Developer. (n.d.). Models for machine learning. [online] Available at: https://developer.ibm.com/articles/cc-models-machine-learning/ [Accessed: 12 November 2023]

[2] Bourg, D.M. and Seemann, G. (2004) AI for game developers. 1st edn. Sebastopol, CA: O'Reilly

[3] IBM Developer. (2017). Train a software agent to behave rationally with reinforcement learning. [online] Available at: https://developer.ibm.com/articles/cc-reinforcement-learning-train-software-agent/ [Accessed: 18 December 2023]

[4] IBM Developer. (n.d.). Models for machine learning. [online] Available at: https://developer.ibm.com/articles/cc-models-machine-learning/#reinforcement-learning [Accessed: 18 December 2023]

[5] GitHub. (n.d.). unity-ml-agents/docs/Training-ML-Agents.md at master · miyamotok0105/unity-ml-agents. [online] Available at: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-ML-Agents.md [Accessed: 6 November 2023]

[6] openai.com. (n.d.). Proximal Policy Optimization. [online] Available at: https://openai.com/research/openai-baselines-ppo [Accessed: 12 November 2023]

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Openai, O. (2017). Proximal Policy Optimization Algorithms. [online] Available at: https://arxiv.org/pdf/1707.06347.pdf [Accessed: 26 December 2023]

[8] Technologies, U. (n.d.). Autonomous Vehicle Training with Simulated Environments | Unity. [online] unity.com. Available at: https://unity.com/how-to/simulated-environments-for-autonomous-vehicle-training [Accessed: 22 october 2023]

[9] Yusef Savid, Mahmoudi, R., Rytis Maskeliunas and Robertas Damaševičius (2023). Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework. Information, 14(5), pp.290–290. doi:https://doi.org/10.3390/info14050290

[10] 牛井 (2022). Unity ML-Agents (Beta). [online] GitHub. Available at: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-PPO.md [Accessed: 6 November 2023]