UPPSALA
UNIVERSITET

An Evaluation of the Unity Machine Learning Agents Toolkit in Dense and Sparse Reward Video Game Environments

Faculty of Arts

Department of Game Design

Authors: Jari Hanski, Biçak, Kaan Baris

Bachelor Thesis in Game Design, 15 hp

Program: Bachelor's Programme in Game Design and Programming

Supervisor: Ulf Benjaminsson

Examiner: Masaki Hayashi

May 2021

Abstract

In computer games, one use case for artificial intelligence is used to create interesting problems for the player. To do this new techniques such as reinforcement learning allows game developers to create artificial intelligence agents with human-like or superhuman abilities. The Unity ML-agents toolkit is a plugin that provides game developers with access to reinforcement algorithms without expertise in machine learning. In this paper, we compare reinforcement learning methods and provide empirical training data from two different environments. First, we describe the chosen reinforcement methods and then explain the design of both training environments. We compared the benefits in both dense and sparse rewards environments. The reinforcement learning methods were evaluated by comparing the training speed and cumulative rewards of the agents. The goal was to evaluate how much the combination of extrinsic and intrinsic rewards accelerated the training process in the sparse rewards environment. We hope this study helps game developers utilize reinforcement learning more effectively, saving time during the training process by choosing the most fitting training method for their video game environment. The results show that when training reinforcement agents in sparse rewards environments the agents trained faster with the combination of extrinsic and intrinsic rewards. And when training an agent in a sparse reward environment with only extrinsic rewards the agent failed to learn to complete the task.

Keywords: Unity, ML-Agents, Reinforcement learning, Sparse rewards environment, Artificial Intelligence

Contents

1. Introduction

Artificial intelligence is becoming more important in our daily life. It has been used to automate tasks, help to make better decisions, and solve complex problems. The new artificial intelligence methods have been developed to meet this new demand. These methods started as a solution to a single narrow problem but now the goal has shifted to create autonomous intelligent behavior. One interesting use case for these new methods is to create intelligent agents in video games. There have also been attempts to create a general artificial intelligence framework by training artificial intelligence to play video games (Perez-Liebana, 2016).

Reinforcement learning is a promising machine learning method that has been rising in popularity in the video game industry in recent years. The reinforcement learning tools such as the Unity machine learning toolkit have become more available for game developers. Reinforcement Learning has been used to train artificial intelligence to play games such as Chess and Go (Silver, et. Al, 2018), and StarCraft (Vinyals, et. al., 2019). The machine learning toolkit makes it possible to train artificial intelligence for games using powerful and modern reinforcement learning algorithms. With multiple reinforcement methods to choose from, it is not trivial to know which method is suitable for what kind of game. Also, the training can take a long time, and it is difficult to evaluate if the training is going to be successful until the training has been concluded.

The purpose of this study is to evaluate the different reinforcement algorithms in the sparse rewards environment. The main reinforcement learning algorithm used in the study rewards the agent for completing tasks in the environment. Because the sparse reward environment only rewards the agent when it has completed the task the training might take a long time or fail. Therefore the main reinforcement algorithm is combined with assistive methods such as curiosity and behavior cloning to help the agent to learn quicker. Assistive reinforcement methods function by helping the agent to discover the rewards from the environment by providing the agent rewards for being curious or following a demonstration from a player. The study aims to answer how much does the assistive reinforcement learning methods increase the training speed of reinforcement agents. There exist some comparisons of the methods included in the ML-agents toolkit. For example, Goulart et. al. (2019) compared differences between reinforcement algorithms in unity ML-agents toolkit by teaching agent to play Bomberman (1985). But they focused on measuring how well the agent played against another agent trained with a different method. Also, the example environments included in the ML-agents toolkit provide a limited comparison between some of the methods. But there is a need for a broader evaluation of all the available assistive reinforcement methods in the ML-agents toolkit in the same environment.

There are multiple training tools for evaluating reinforcement learning algorithms (Juliani, et. al. 2020). These are usually based on existing games or game engines. DeepMind Lab is built from the Quake III engine and can be used for studying tasks in robotics and animal psychology. The shortcomings are that the engine is based on old technology, and it is limited to a first-person perspective. Another training environment is Project Malmo (Malmo) which is based on the Minecraft game. It has been used in research focusing on hierarchical control and multi-agent communication. Limitations are related to the underlying Minecraft engine. This includes pixelated graphics and a simple physics system. It is also only able to simulate scenarios that

are possible in Minecraft. The Unity Machine Learning Toolkit (ML-Agents) is an open-source add-on package to the Unity game engine. It enables the developers to use game environments to train artificial intelligence for games. These so-called agents can be trained using reinforcement learning to perform tasks in the game. Unity ML-agents toolkit was our choice for this study because it makes reinforcement learning more accessible to game developers.

We compared different algorithms in the ML-Agents by training reinforcement learning agents. The training was done in both dense and sparse rewards environments to compare the benefits in both environments. In the dense reward environment, the agent gets rewarded often by the training environment. While sparse reward environment has a limited number of rewards or they receiving rewards requires completing complex tasks. The reinforcement learning methods were evaluated by comparing the training speed of the agents in the same environment. The goal was to evaluate how much the combination of extrinsic and intrinsic rewards accelerated the training process. The results from this study could help game developers utilize reinforcement learning more effectively, saving time during the training process by choosing the most fitting training method for their video game environment.

The result from this study shows the assistive reinforcement learning methods are beneficial in sparse rewards environments. But did not have a significant benefit in the dense rewards environment. The reinforcement agent failed to complete the task in a sparse rewards environment with only extrinsic rewards and succeeded with the combination of the extrinsic and intrinsic rewards. This indicates that the assistive reinforcement learning methods are beneficial when the rewards in the game are sparse.

2 Reinforcement Learning Methods

2.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) solves problems by trial and error. To achieve this the agent is placed in a learning environment for a predetermined time. The training time is divided into training episodes with defined duration in time steps *t*. In each step, the agent observes the state of the environment (S_t) through its sensors. The agent selects its next action (A_t) and modifies the environment state (S_{t+1}). The agent will receive a reward (R) from the environment when it achieves its goals. The goal of the agent is to maximize the cumulative rewards for each training episode.

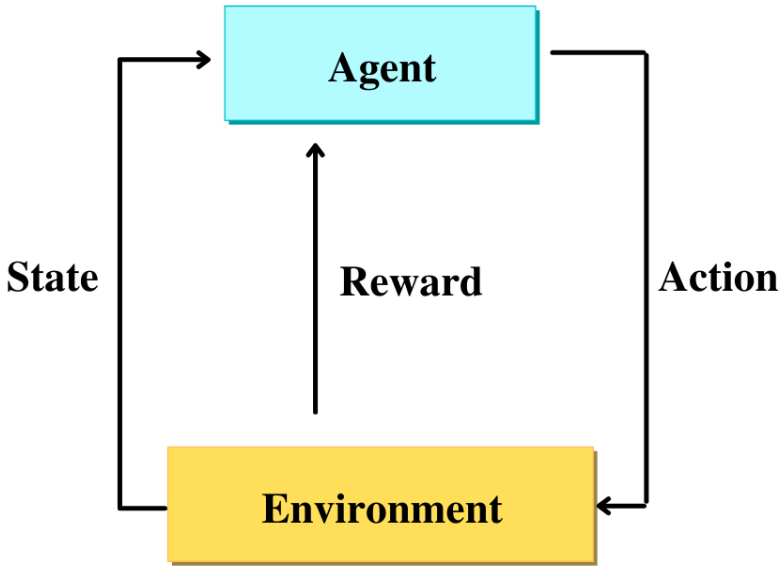


Figure 2: The reinforcement learning cycle.

A learning environment is where the agent can observe its surroundings and perform actions. The environment needs to be designed so that the agent can collect enough information relevant to its goal. The environment must also contain enough negative or positive rewards to the agent so that it will learn from its exploration of the action space. The agent should be able to predict how its actions affect the state of the environment so it can learn to understand its learning progress.

The agent's actions allow it to affect the environment in a certain way. By trying actions randomly, the agent aims to find a behavior (policy) that gives it the highest reward. The policy is the agent's behavior function which tells the agent which action to take in the state S. The agent learns by receiving positive or negative rewards as feedback for its actions. The rewards the agent receives can be sparse or frequent and can differ in size. The agent can receive negative rewards for example performing tasks slowly or colliding with a wall and positive rewards for collecting coins or scoring a goal in a football match. The agents can receive

extrinsic or *intrinsic* rewards during their training. Barto et al. (2004) refer to *intrinsic rewards* as an agent's *intrinsic* motivation for example curiosity. He also refers to extrinsic *reward* as something that is tied to a specific task in the learning environment, for example reaching the end of the level in a video game.

Policy gradient (Peters and Bagnell, 2010) methods rely on optimizing parameterized policies to maximize expected cumulative reward. Thus, the goal is to maximize the reward function therefore the gradient ascent optimization algorithm is used.

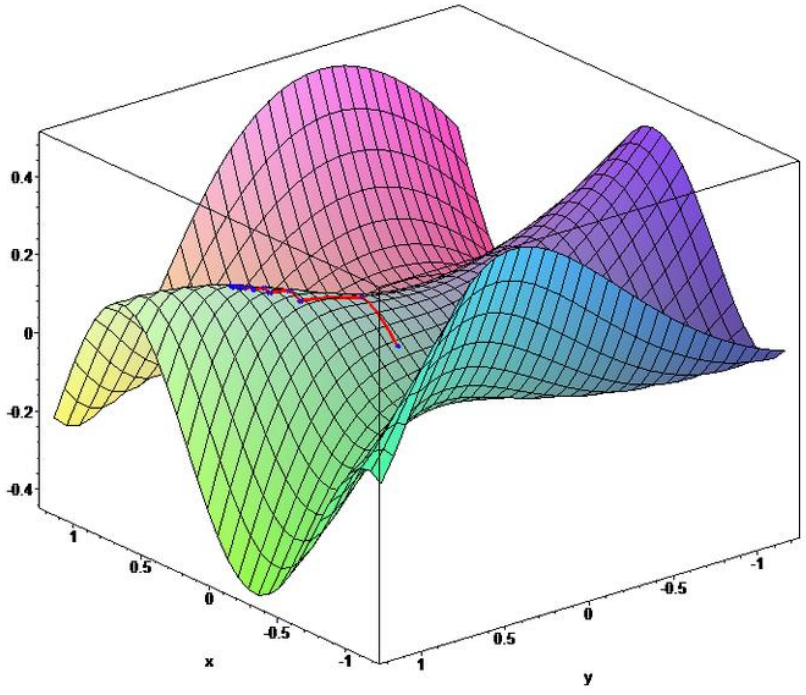


Figure 3: Gradient ascent (retrieved from Wikipedia.org, public domain)

The policy gradient loss can be expressed as:

$$L^{PG} = \hat{E}_t\left[log\ log\ \pi_\theta(a_t \mid s_t)\hat{A}_t\right]$$

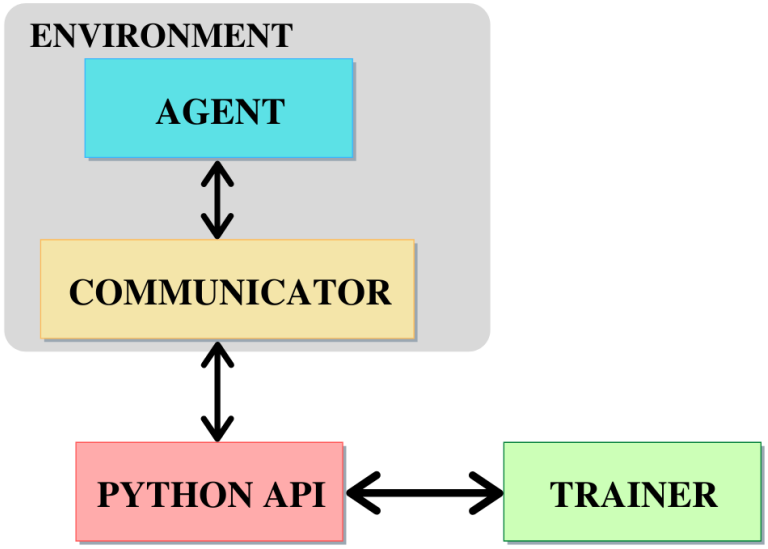
Where the π_θ is a neural network that takes the observed state from the environment as input and suggests actions to take as an output. And $\hat{A}_t$ is the advantage function which tries to predict what the relative value of the selected action in the current state is.

The advantage function consists of two components: the discounted rewards and the baseline estimate. The discounted rewards are the sum of the rewards the agent got during each timestep of the current episode. The rewards are discounted meaning the agent values more about rewards it will receive now than many timesteps later. The value function tries to give an estimate of the discounted sum of rewards from this time step forward. In other words, it is trying to guess the final return of the current episode starting from the current state. If the advantage result was better than the average return the probability of the agent choosing the action is increased. Likewise, if the advantage was worse than average the likelihood of choosing the action is lowered when encountering the same state. The problem of continuing running gradient ascent on a single batch of collected observations is that the parameters will

get updated far outside of the range where the data was collected. Because of this, the advantage function becomes unable to predict the expected return and this effectively destroys the model.

2.2 The Unity ML-Agents toolkit

The Unity Machine Learning Toolkit is an open-source add-on package to the Unity game engine (Juliani, et. al., 2018). It enables the developers to train behaviors using reinforcement learning by utilizing the open-source PyTorch reinforcement learning library. PyTorch enables training AI agents using CPU and GPU and the resulting Neural Network (NN) behavior model can be used to control the agents in the Unity engine. The ML-agents toolkit consists of three main components: the agent, the environment, and available actions for the agent. Agents share their experiences during the training. Their policy must access player inputs, scripts, and neural networks through Python. For this purpose, the agent uses Communicator to connect to the trainers through the Python API.



2.3 Hyperparameter configuration

Hyperparameter configuration file is used to configure the parameters for the reinforcement learning algorithms. The hyperparameters are based on the Pyramids demo from the Unity ML-agents toolkit. We changed hyperparameters based on trial and error until the training was successful in our environments. After finding the suitable hyperparameters the same hyperparameters were used for each of the training methods in the same environment. The only exceptions were the additional unique parameters that are required for GAIL, curiosity, behavior cloning, and RND training methods.

See appendix A for the parameters used in the city environment.

Figure 4: Training environment in Unity game engine.

A learning agent is created by placing a game object in the training environment with the required Agent script component. The Agent script is included in the ML-agents toolkit and allows developers to extend it by creating a new script inheriting from the Agent script to assign rewards and observations to the agent.

The Agent script has a dependency on a Behavior Parameters script. Behavior Parameters script behaves like the brain of the agent and communicates observations and rewards to the trainer through the Python API.

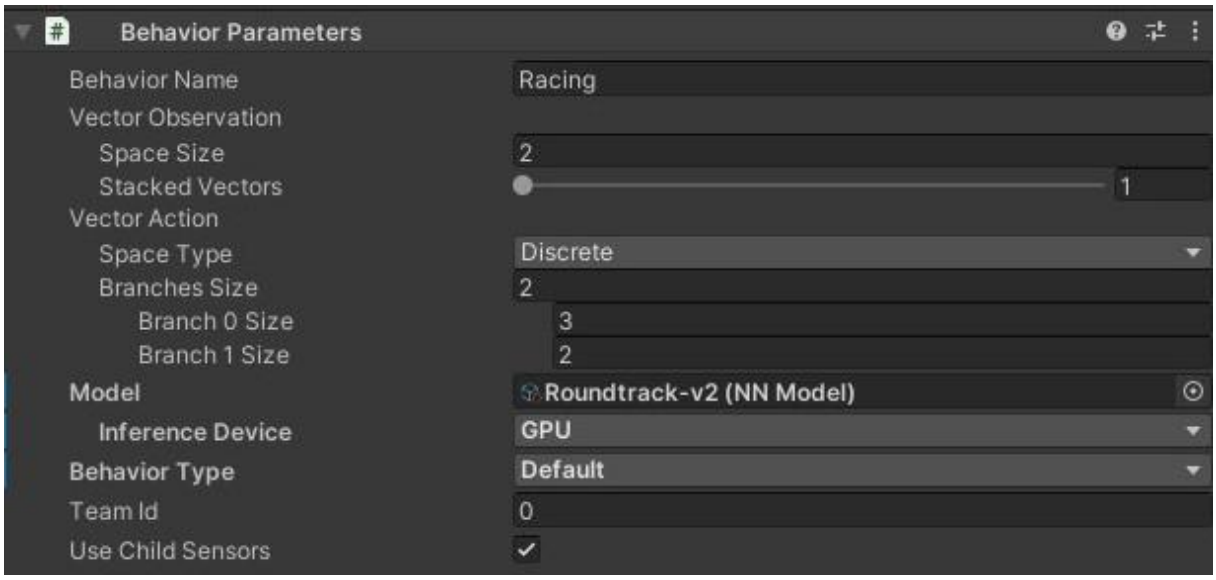


Figure 5: The Behavior Parameters script in Unity Editor

- **Behavior Name:** Defines which set of hyperparameters this brain uses.
- **Vector Observations:** The size of an array in which the observations of the environment are stored.
- **Vector Action:** Float (continuous) or integer (discrete) array containing actions for the agent.
- **Model:** The trained model used by the agent when not training.
- **Behavior Type:** Three possible options available: Default, Heuristic, and Inferencing. The Default option will train the agent when it is connected to the Python API and fall back to Inferencing otherwise. Inferencing means that the agent is using the trained model to choose its actions. The heuristic is used for testing agents by giving them actions by pressing keyboard keys for example.

The Decision Requester component is used to control the rate of requested new actions from the reinforcement learning model and feed the actions to the agent component. Agents collect observations from the game scene, take actions, and then receive rewards for them. The agents can observe the environment by using ray casts and supplied vectors.

2.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) reinforcement algorithm (Schulman et. al. (2017) is the default core algorithm used in the ML-agents toolkit. PPO is an easy-to-use algorithm and relatively stable because it is protected from destructive large policy updates from gradient ascent by constraining policy update maximum change to a smaller range of $[1 - \varepsilon, 1 + \varepsilon]$. The objective function penalizes the policy changes which move the r_t (θ) away from 1. The main objective function in PPO is:

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

- ε is a hyperparameter, (usually 0.1 or 0.2).
- $\hat{E}_t$ is the expectation over timesteps.
- r_t is the probability ratio of new and old policies.
- $\hat{A}_t$ is the estimated advantage at time *t.*
- θ is the policy parameter.

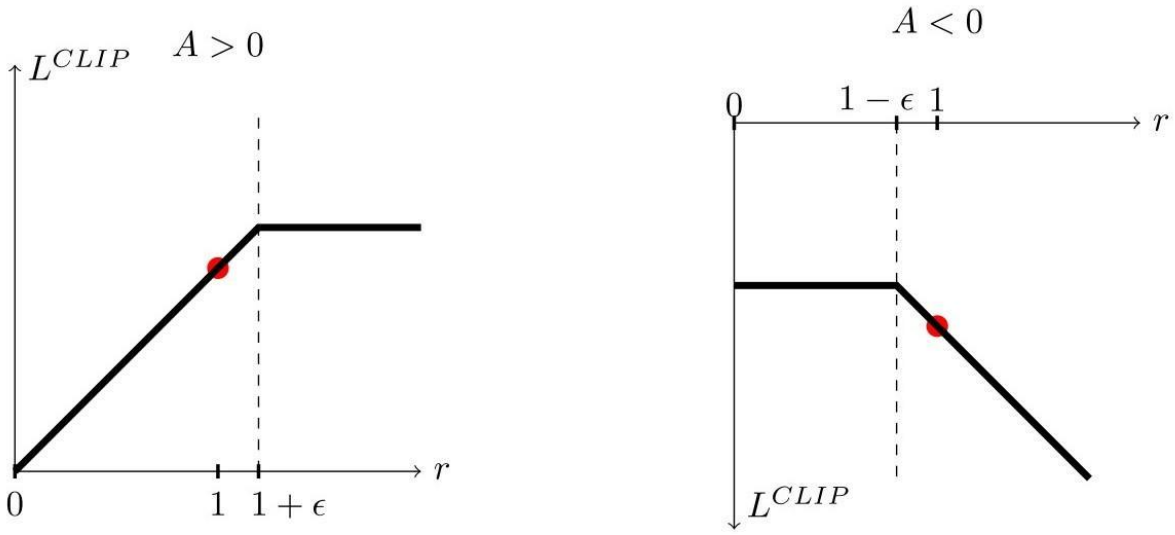


Figure 6: L^{CLIP} plotted as a function of probability ratio *r*, for positive advantage on the left and negative advantage on the right (Schulman, et. al., 2017)

Because the clipped objective function makes the PPO an easy-to-use and relatively stable reinforcement algorithm and is suitable for training agents for games.

2.5 Assistive reinforcement learning methods

The assistive reinforcement learning methods are often used to supplement the Proximal Policy Optimization algorithm cause they help the agent to find rewards early in the training which reduces the training time. The training of agents is done in the Unity game engine while the python trainer is running in the background. During the training, the agents collect observations, perform actions, and receive rewards. The different agents are trained using the following assistive methods included in the ML-toolkit:

- Proximal Policy Optimization (PPO)
- Generative Adversarial Imitation Learning (GAIL)

- Random network distillation (RND)
- Curiosity-driven exploration
- Behavior Cloning (BC)

Barto et. al. (2004) discovered that by combining intrinsic rewards with extrinsic rewards they were able to train competent autonomous agents. Alternative machine learning methods were developed to address the shortcomings of machine learning algorithms at the time. Problems such as that the algorithms had to be carefully hand-tuned for a single problem and could not extend their abilities to another problem. Thus, these assistive methods are combined with the Proximal Policy Optimization algorithm when training the agents.

2.6 Generative Adversarial Imitation Learning (GAIL)

Generative Adversarial Imitation Learning (GAIL) is a model-free imitating algorithm that offers performance gains compared to earlier model-free methods in complex environments (Ho, Ermon, 2017). Because the method is model-free it must learn from the environment, and it needs more interaction with it than model-based methods. The method explores actions randomly to find which of the actions allows the policy to shift towards the expert's policy. GAIL allows agents to learn behavior by observing the expert's behavior without the knowledge of the reinforcement signal. The expert behavior is recorded in a demonstration file containing states and actions. The framework can extract behavior directly from the provided example.

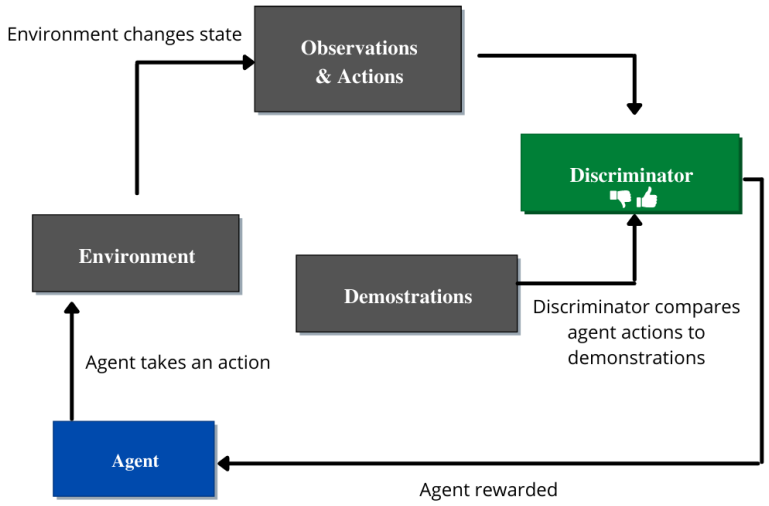


Figure 7: Generative Adversarial Imitation Learning

2.7 Curiosity-driven exploration by self-supervised prediction

Rewards are sparse or absent in real-world environments according to Pathak et. al. (2017). In these situations, curiosity can be used to reward agents to explore the environment and learn

new skills needed to reach their goals. With this method, the agent's ability to predict the consequences of its actions is defined as the agent's curiosity. Curiosity-driven exploration (Pathak, 2017) pushes agents to explore more effectively and allows for less interaction with the environment to reach the goal as well as being able to predict its consequences in unseen states. The benefit of curiosity-driven exploration is that it pushes the agent to explore new areas in the games.

2.8 Behavior Cloning (BC)

Behavior cloning (Hussein et al., 2017) is a solution to a need to mimic human behavior effectively by intelligent agents. The agents trained with behavior cloning allow them to behave like a human would in a similar situation. It has been used in fields such as self-driving cars, assistant robots, and human-computer interaction. Imitation learning uses information extracted from the behavior of the teacher and the environment. Limitations of imitation learning are that they require good quality demonstrations from experts and that the agent can never surpass the skill of human demonstration. The behavior cloning is excellent for agents which needs to mimic the demonstration as closely as possible. The downside is that the agent does not improvise when choosing actions.

2.9 Random network distillation (RND)

Random network distillation (Burda, 2018) introduces an exploration bonus to reinforcement learning methods. The bonus is the error of a neural network (NN) feature prediction of the observations from a fixed randomly initialized NN. The method is designed to tackle the sparse rewards problem by acting as a directed exploration method. The agent is rewarded for novel experiences of interacting with the environment or exploring new areas. Unlike most curiosity methods the RND method is not susceptible to getting stuck when observing random noise on the TV for example. This is due to the method using exploration bonus instead of absolute prediction error. The RND is a promising algorithm that is suitable for agents who need to explore complex environments with lots of noise in the collected observation data.

3 Relevant work in the field of Reinforcement Learning

A sparse reward task is defined as an environment where only a tiny number of states will reward the agent. A sparse reward function is easy to design and only requires defining the criteria for solving a problem. When using the sparse rewards, the agent is rewarded when the criteria are fulfilled. But in environments with spare rewards, finding the reward is challenging. Therefore, learning from the task in a reasonable time is difficult. The agent might never perform the correct sequence of actions to receive the reward. This prevents the training from progressing. While the sparse rewards are difficult to learn from recent work has demonstrated that sparse reward functions result in trained policies which can perform the desired action instead of getting stuck in local optima (Vecerik, et. al, 2017). This means that the agent is more likely to do what the designers want instead of trying to optimize for collecting rewards that are supposed to guide the agent. In our study, a city environment was designed to reflect a sparse reward environment.

The solution for this is *the Reward shaping* (Mataric, 1994) technique which is used to help with the random search in sparse reward environments. It requires creating a reward function that guides the reinforcement agent to perform the desired task. While the approach might help the agent to learn the task, Nair et. Al. (2018) found that manually shaping the reward function can lead to suboptimal results. Meaning agents get stuck in the local optima instead of finding the optimal behavior. Also, they tend to be specific to a single problem and require domain expertise from the designer (Mataric, 1994). In our study, the rewards shaping approach was used when we designed the racing track training environment to guide the agents towards the goal by using the checkpoints and giving the agent rewards when it is facing the checkpoint.

There are also methods to aid the reinforcement learning process by utilizing demonstrations recorded by humans. For example, Generative adversarial imitation learning (GAIL) (Ho & Ermon, 2016) allows the agent to extract policy directly from the data which results in significant improvements in training results in imitating complex behaviors in large environments. This improves the training results of learning the effective policy in games with sparse rewards. Another method is behavior cloning (Burda, 2018) which has been used to train more human-like Non-Player Characters (NPCs) for a first-person shooter. (Borovikov, et al., 2019). There have also been successes in using curiosity methods when training agents for games. An agent was successfully trained to play Super Mario Bros using curiosity-driven exploration (Pathak, 2017). Another example is Montezuma's revenge (1984) which is a notoriously hard computer game for reinforcement learning agents to learn. It has been used as a benchmark for reinforcement learning algorithms. It was too difficult for most algorithms until 2018 when OpenAI achieved a breakthrough and were able to train an agent using a novel Random Network Distillation (RND) algorithm (Burda, et.al., 2018) which achieved a new record high score. These assistive methods are used with PPO and evaluated in this study.

4 Method

4.1 Training Environments

Dense and sparse reward training environments were created to test the reinforcement methods. The dense reward environment is a racing track in which the agent is rewarded for completing sections of the track and penalized for colliding with the walls. The sparse reward environment is a city where the agent must pick up passengers and return them to one specific spot. The agent only receives a reward after completing the whole task.

Racing track environment

A 3D training environment was created in the Unity game engine as seen in figure 7. It has two different training tracks with unique shapes and lengths. The shape of the track was chosen arbitrarily. The main difference is that the cars are driving clockwise on another track and counterclockwise on the other. Two training tracks were also created to prevent overfitting the model (Zhang et. al. 2018) because overfitting can lead to the race car agents failing to complete tracks they have not seen before. Four cars controlled by the agents are allocated for each of the training tracks.

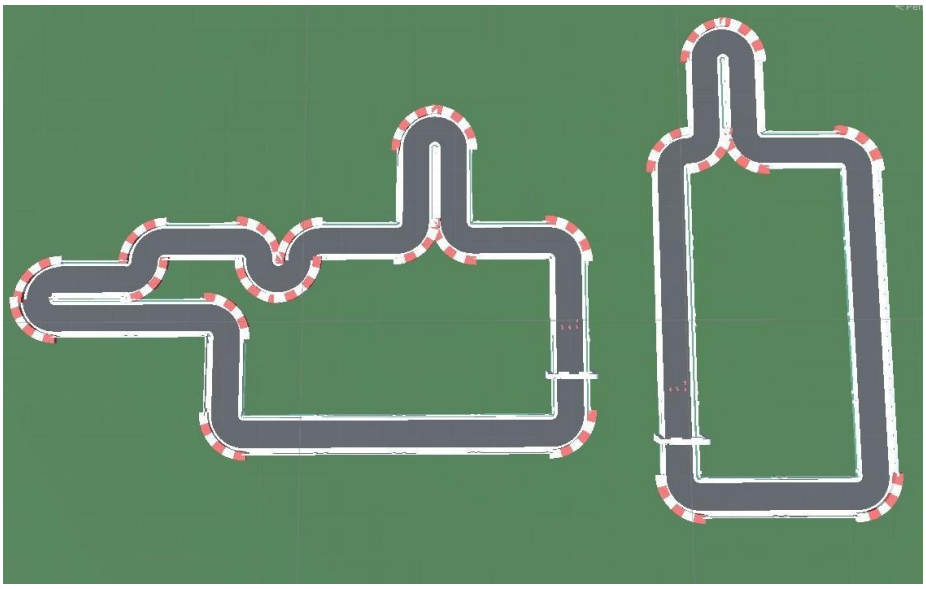


Figure 8: Two training tracks in the training environment

City environment

Another 3D environment was created with sparse rewards (Figure 9). The goal of the agent is to find a passenger in a city and return the passenger to the goal. The position of the passenger is random, and the goal position is static.

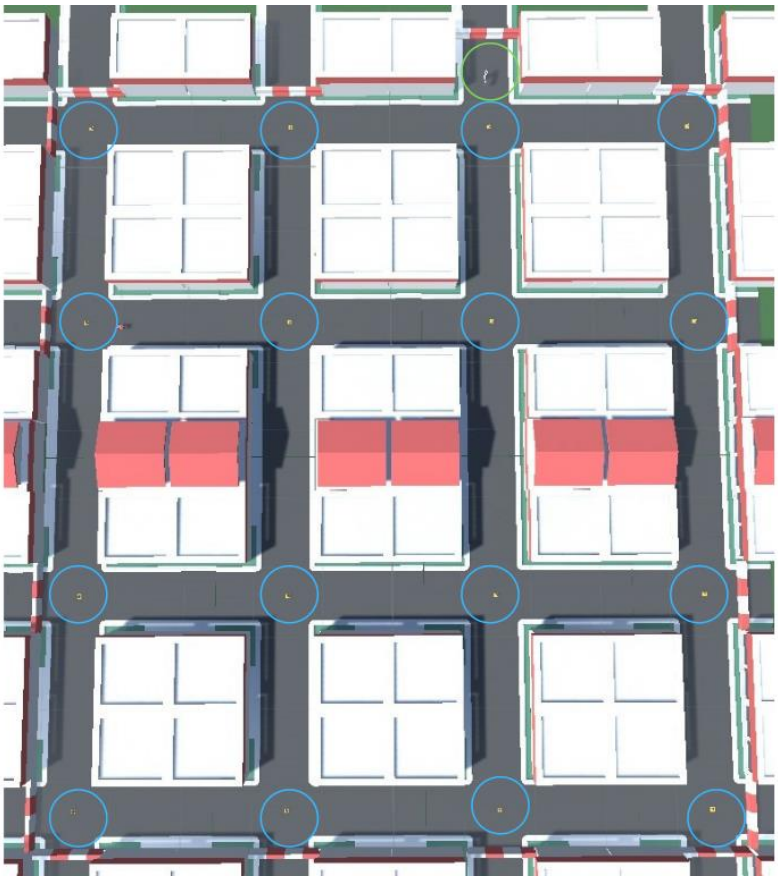


Figure 9: Taxi training environment. Blue circles are the possible positions for the passenger. The green circle is the destination of the passengers (goal).

4.2 Reward space

It is important to design a suitable reward function for the agent to learn the desired behavior. If the reward function design is flawed, it could lead to unwanted results or agents optimizing their behavior for the rewards instead of the intended goal.

Racing track environment

To reinforce wanted behavior the environment needs to contain extrinsic rewards for the agent to discover. Trigger boxes are placed at regular intervals along each track. When the agent reaches a trigger, it receives a small positive reward. The agent is rewarded a small reward every time step when it is facing a correct waypoint:

$$R_1 = dot(v_{agent}, v_{waypoint})$$

Where the $v_{agent}$is the forward direction of the agent and $v_{waypoint}$ is the direction to the next waypoint.

The agent is also rewarded for its high speed:

$$R_2 = (m_t/m_{max})$$

Where the m_t is the magnitude of the velocity vector and m_{max} is the maximum velocity magnitude.

The agent is given a small penalty for each time step to encourage completing the goal as quickly as possible:

$$R_3 = \frac{-1}{maxStep}$$

If the agent collides with a wall, it receives a big negative reward, and the training episode immediately ends.

City environment

The agent is given a big reward (+2) when it successfully delivers a passenger to the goal zone. The agent also gets small negative reward every time step:

$$R_3 = \frac{-1}{maxStep}$$

The agent is not rewarded for picking up the passengers or colliding with walls which makes it a sparse reward problem.

4.3 Action space

An agent is only able to pick one action out of the list of actions. The acceleration force is applied to the agent's rigid body when the agent chooses to accelerate or brake. The turning directions are mapped from the action input and are multiplied by the turning speed variable assigned to the agent script.

Racing track environment

The AI agents are controlling a racing car intending to reach the end of the track. The agent can perform the following two discrete actions:

Accelerating:

- Idle (0)
- Accelerate (1)

Turning:

- Drive straight (0)
- Turn left (1)
- Turn right (2)

City environment

The car in the city environment is only able to perform one action at a time:

- Break (0)
- Idle (1)
- Accelerate (2)
- Turn left (3)
- Turn right (4)

4.4 Agent Observations

It is important to decide which observations of the game state are relevant and helpful when training the AI agents. Providing too little relevant information prevents the agent from learning and providing too much irrelevant information provides no useful information for solving the problem. The observations are a way for agents to measure the state of the environment. The observations are collected by the agent's *VectorSensor* component which collects observations as a list of floats. For the best results, the observations are normalized to the range of [-1, 1] or [0, 1]. This helps PPO neural networks in finding solutions faster. The observations are added to the sensor every time the agent enters a new state and before it has taken an action. It is important to give the agent relevant information about its surroundings and omit the information not relevant to the goal. The information must be sufficient for the agent to find the best policy to complete the task and maximize the rewards.

In addition to the agent observations, the Demonstrations Recorder is used to record the demonstrations from the Unity engine for the use of GAIL and behavior cloning. The heuristic mode of the agent allows authors to control the agent by using the keyboard. The demonstrations contain actions, rewards, and observations of the agent while it was controlled by the player.

Observations in the racing track environment

To learn about the walls in the environment the agents use ray casts. The rays collide with the walls of the track and help the agent detect walls around them. An agent is ray-casting forward the direction of the car and to the sides of the car. The walls are identified with the "Wall" tag and identified by the agent.

The agent can observe the location of the next checkpoint and its direction and speed training environment. For this purpose, the agents are collecting the following information:

- Direction to the next waypoint. A dot product between the agent's forward vector and the normalized direction to the next waypoint is added to the agent's observations. This is used by the agent for determining if it is facing the right way.
- Speed of the agent. A normalized magnitude of the velocity vector is used by the agent to determine how fast it is traveling (current Magnitude – min. Magnitude/max. Magnitude – min. Magnitude).

Observations in the city environment

The agent in is equipped with *Ray-cast Detectors* and can detect following objects in the environment:

- Goal
- Passenger
- Wall

The agent also observed its velocity.

4.5 Reinforcement learning training methods

Multiple different combinations of learning methods were used in training, and the training time and cumulative rewards were analyzed and compared. The training time is measured in time steps and the training methods were evaluated based on the cumulative rewards the agents received and if the agent was able to solve the problem within the maximum timesteps. The training of the neural network was done in parallel to speed up the training process.

The following assistive methods were used with Proximal Policy Optimization (PPO):

1. Random network distillation
2. Behavior Cloning
3. Curiosity
4. Generative Adversarial Imitation Learning
5. Generative Adversarial Imitation Learning + Behavior Cloning + Curiosity

4.6 Collecting training data

Statistics were collected about the environment and the policy during training of the reinforcement learning agent.

Following important statistics are recorded:

Cumulative Reward: For every training episode the cumulative reward and the time step is recorded. The mean reward is expected to increase during the training and reach a similar level for each of the training methods. Thus, the mean cumulative reward per episode is compared.

Episode Length: The number of simulation times steps it took the agent to complete its goal.

Following policy statistics are also recorded:

Extrinsic Reward: How big a reward the agent received from the environment.

Value Estimate: A reward estimate for all the states visible to the agent. Should increase during training.

Curiosity Reward: A mean cumulative intrinsic reward per episode for agents using the curiosity method.

Curiosity Value Estimate: Agent's estimate of the curiosity reward it will receive.

GAIL Reward: A mean cumulative reward awarded to the agent by the discriminator neural network.

GAIL Value Estimate: An agent's prediction of the GAIL reward it receives.

Entropy: Randomness of the decisions of the agent. Entropy should decrease as the training progresses.

4.7 Analyzing the results

The agents were evaluated by the cumulative reward they received during the training process. The period of data collection was performed until the maximum steps were reached during the training or after the cumulative rewards stabilized. The earlier the cumulative reward stabilized the faster the training progressed.

Finding optimal values for the PyTorch hyperparameters is crucial for the training to be successful. Therefore, a visualizing tool called *TensorBoard* was used for observing the training progress by visualizing the rewards awarded to the agents. The TensorBoard displays charts of cumulative rewards plotted against elapsed time steps.

The racing track environment acts as a test environment where there should be no big benefits from using assistive reinforcement methods with proximal policy optimization. The city environment is the actual test environment where we should be able to expect an increase in training results compared to PPO alone.

5 Results and discussion

The training of reinforcement learning agents was performed in two different test environments. A racing track environment included dense rewards in form of checkpoints for the cars. The city environment was a sparse reward environment that did not contain any rewards other than when the agent reached the goal. The agents were trained in both environments using different reinforcement learning methods. The learning progress of agents was observed and evaluated from the data collected during the training. The cumulative rewards are plotted against the elapsed time-steps during the training. The time-step here means one learning cycle during which the agent takes one action and receives rewards.

5.1 Training results from the racing track environment

The training data is represented in a line chart. Y-axis is the cumulative reward for the episode and the X-axis the timestep. The pale lines are raw measurements and the solid lines are smoothed using exponential moving average. Each line in the chart represents different training methods:

- Orange: PPO + RND
- Blue: PPO
- Red: PPO + Curiosity
- Light blue: PPO + Behavior Cloning
- Pink: GAIL + Curiosity
- Gray: PPO + GAIL

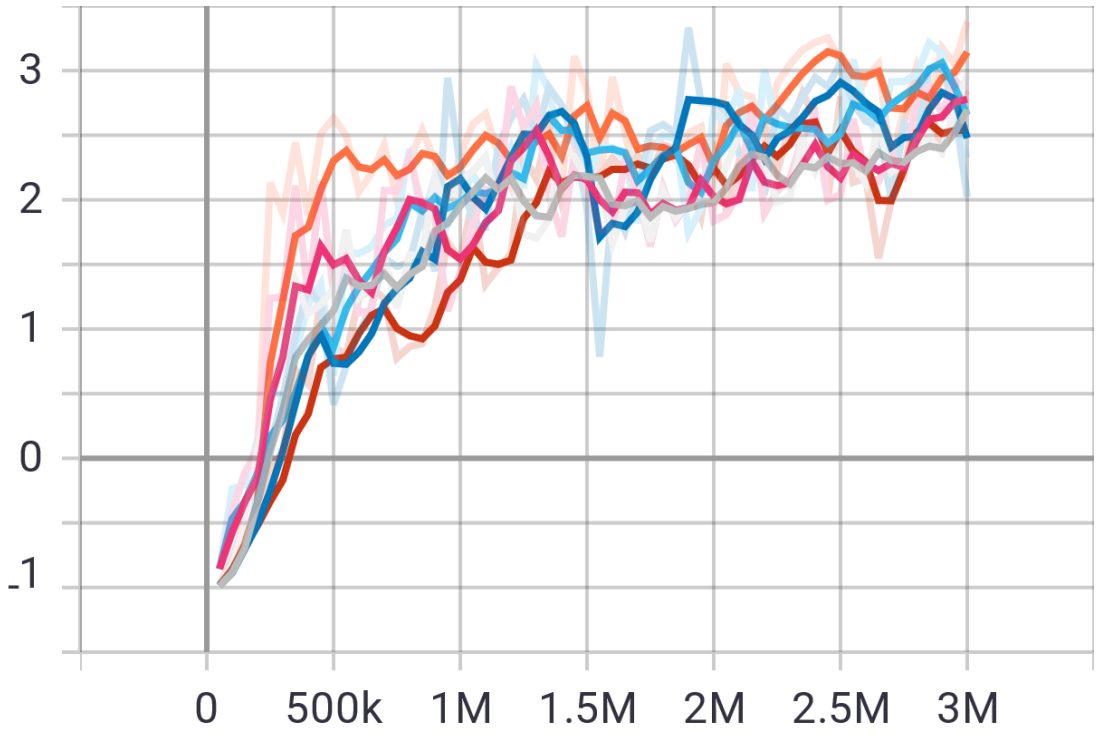


Figure 10: Racing track environment: Cumulative Reward

The PPO + RND agent performed the best and achieved a high cumulative reward in the shortest time. The slowest agent was the PPO + Curiosity. Each of the agents learned to perform the task similarly and the quickest agent was trained within 500k timesteps and the slowest was fully trained in 1.5 million timesteps. Apart from small performance differences, there are no large differences between the methods.

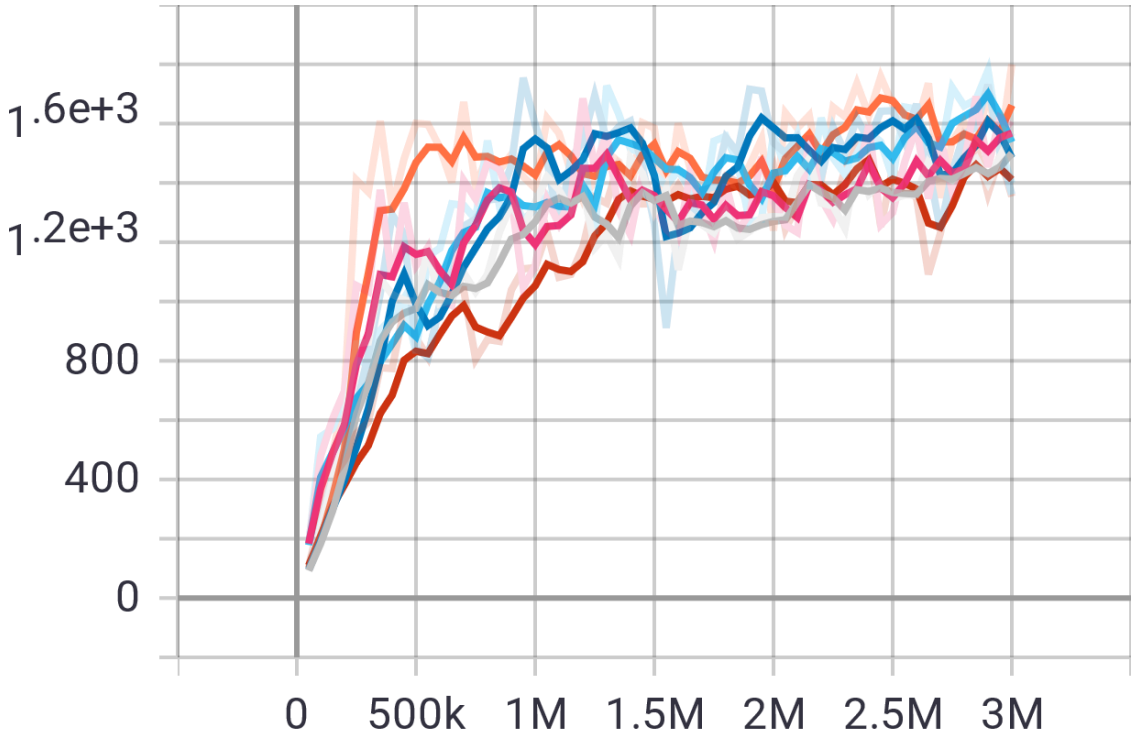


Figure 11: Racing track environment: Episode Length

The PPO + RND agent achieved the longest episode lengths earliest in the group meaning it was able to complete the whole track without colliding with the walls earlier than other agents. PPO + RND agent was fully trained after only 500k time-steps. PPO performed worst of all the agents and it receives the high cumulative rewards at around 1.5 million time-steps mark. At the end of the training, there are no big differences in the performance of each method and each of the agents performs the task in a similar time.

5.2 Training results from city track environment

Each line in the chart represents different training methods:

- Orange: PPO + GAIL + BC + Curiosity
- Blue: PPO + GAIL + Curiosity
- Red: PPO
- Light blue: PPO + Curiosity
- Pink: PPO + RND

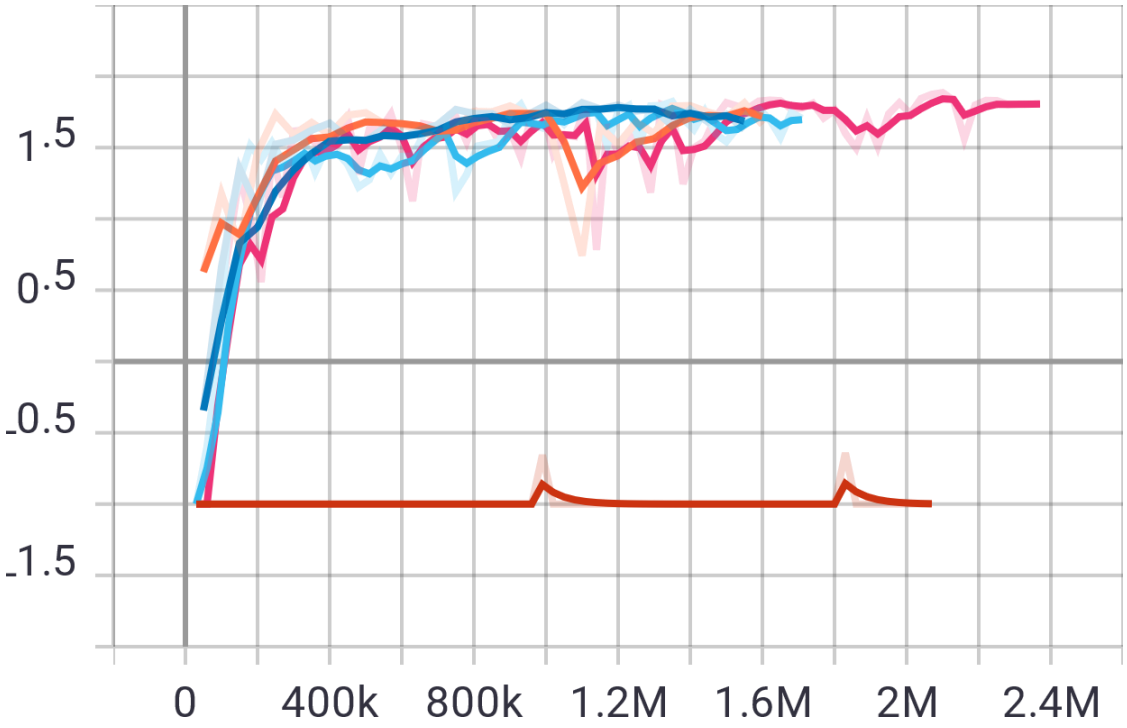


Figure 12: City environment: Cumulative Reward.

As seen in figure 12 the training was performed using different methods. The training was stopped after around 1.6 to 2 million timesteps when the cumulative rewards had flatlined.

In the city environment, the red agent's (PPO) line is flat and received constant negative rewards meaning it failed to complete the goal and training failed. When PPO was combined with assistive methods they were able to learn the task successfully. There was a small dip in rewards around 1.2 million steps for agents using the curiosity (Orange) and RND method (Pink). This is normal behavior for the curious agent when they start starts to explore a new state which is not linked to extrinsic rewards. This causes the agents to ignore the extrinsic rewards in the favor of the rewards it receives from being curious. The agents quickly reached a cumulative reward of 1.5 within 400k time-steps which is relatively quick when the task was so complex.

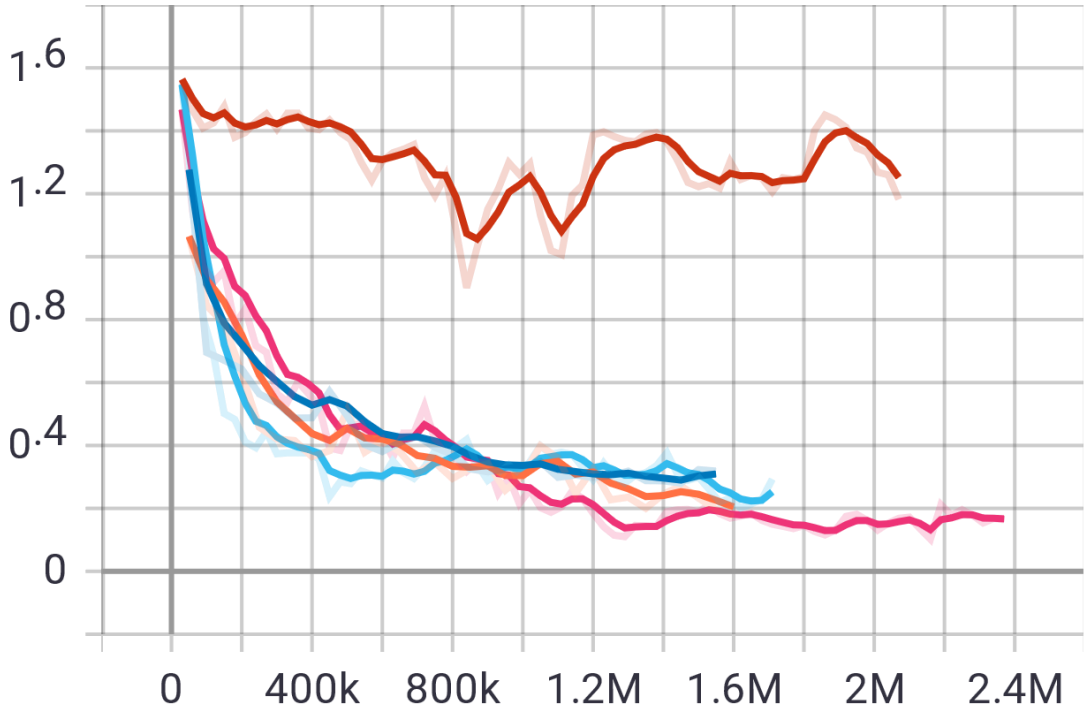


Figure 13: The Entropy of the policies

As seen in figure 14 the entropy of the policy stayed high. This means that the red agent never encountered the reward and continued to explore the environment randomly. The curiosity and imitation methods made it possible for the agent to find the reward and begin to optimize for the solution. This means that the agents began to make decisions based on the learned model instead of random actions.

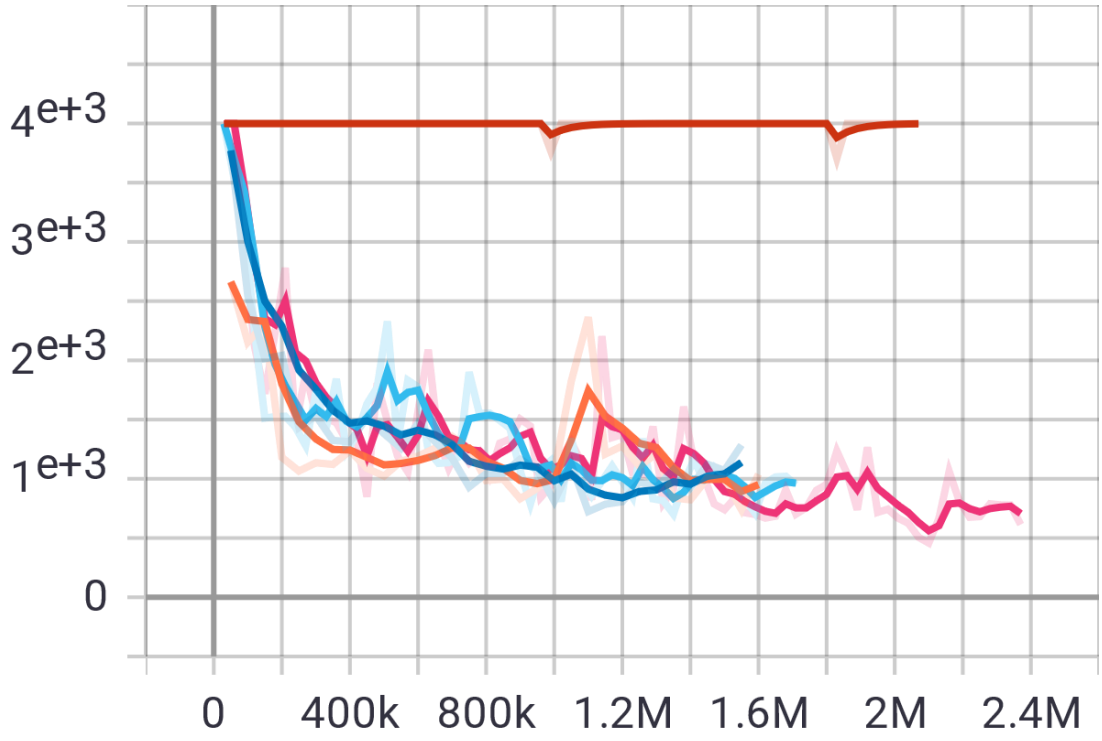


Figure 14: Racing track environment: **Episode Length**

As seen in figure 15 the episode length got shorter when most of the agents learned to complete the task faster. The red agent spent the maximum allowed episode length on trying to find the rewards. This is another sign that the red agent never was able to complete its goal and was looking for a reward randomly.

5.3 Discussion

The reinforcement methods behaved as expected in the dense rewards environment. There were no significant differences in the assistive methods compared to using PPO alone. This might be because the rewards were easy to find, and the reward shaping was working as intended. Also, the track only had one way forward and was surrounded by walls so there was a high chance that the agent found the first checkpoint by just exploring randomly.

The real differences can be seen in the city environment where random exploration does not work anymore. The agents seem to benefit from the intrinsic rewards to guide them towards the rewarding actions. The most important result is that the agent without assistive methods completely failed to complete the task of picking up passengers and delivering them. The agent's rewards did not improve during the training and it received a constant cumulative reward of -1. It sometimes got lucky and returned a passenger to the goal but was not able to do it consistently.

Another interesting thing is that the agents in sparse rewards environments trained as fast as the agents in the dense rewards environment. They both began to receive high cumulative rewards around 400k time-steps. The environments are not directly comparable because of different observations, actions, and hyperparameters. But we were expecting training to take longer in the sparse rewards environment.

5.4 Additional training data

Appendix B contains the most important training data for the racing track environment.

Appendix C contains the most important training data for the city environment.

6. Conclusion

In this paper, we compared different reinforcement learning methods in the machine learning toolkit for the Unity game engine. The reinforcement learning methods were evaluated in two different environments. One environment with sparse rewards and another with dense rewards. The goal was to provide empirical data to allow game designers to choose a suitable reinforcement algorithm for their game.

The results suggest that agents trained faster in a sparse rewards environment when they received a combination of extrinsic and intrinsic rewards. When an agent received only extrinsic rewards in the same environment it completely failed to learn the task. Using the proximal policy optimization reinforcement learning method alone was sufficient in the dense reward environment where the agent could find rewards easily but was not sufficient in the sparse reward environment.

When looking at the performance differences of the evaluated assistive methods the random network distillation (RND) method performed slightly better than other methods in the dense rewards environment. There was not much difference in the training speed when comparing the assistive methods in sparse rewards environments. When at least one of the assistive methods was used with proximal policy optimization (PPO) the training results improved significantly in sparse rewards.

Curiosity-driven exploration and RND are interesting methods for game developers because they require no pre-recorded demonstrations. They are also able to explore the game environment effectively. This allows game developers to save time because there is no need to record demonstrations. It also eliminates possible problems related to suboptimal demonstrations with generative adversarial imitation learning and behavior cloning. Thus, we recommend using RND when sparse rewards are used, and the environment is complex or large. Imitation learning methods are recommended when the agents should behave similarly to the player.

Since most of the assistive methods performed similarly in our sparse rewards environment it would be interesting to study them in a more complex environment. For example, in our city environment, all the streets were exact copies of each other. This meant the reinforcement agent could not learn to identify which street it currently was on. Adding some uniqueness on each of the streets could have helped the methods perform better. We might also see bigger differences in the results when giving the agent more complex tasks. Further study could also be done on the evaluation of the agents trained with different methods through the lens of how humanlike their behavior is or how unfair they feel to play against.

References

Barto, Andrew G., Satinder Singh, and Nuttapong Chentanez. "Intrinsically motivated learning of hierarchical collections of skills." Proceedings of the 3rd International Conference on Development and Learning. 2004.

Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018). Exploration by random network distillation. ArXiv, 1–17.

Borovikov, Igor, et al. "Towards interactive training of non-player characters in video games." arXiv preprint arXiv:1906.00535 (2019).

Goulart, Ícaro, Aline Paes, and Esteban Clua. "Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning." Joint International Conference on Entertainment Computing and Serious Games. Springer, Cham, 2019.

Hussein, A., Gaber, M. M., Elyan, E., Jayne, C., & Gaber, M. M. (2016). A Imitation Learning: A Survey of Learning Methods A:2 A. Hussein et al. ACM Computing Surveys, V, 1–35.

Ho, J., & Ermon, S. (2016). Generative adversarial imitation learning. *arXiv preprint arXiv:1606.03476*.

Juliani, A., Berges, V. P., Teng, E., Cohen, A., Harper, J., Elion, C., ... & Lange, D. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

Mataric, Maja J. "Reward functions for accelerated learning." Machine learning proceedings 1994. Morgan Kaufmann, 1994. 181-189.

Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018, May). Overcoming exploration in reinforcement learning with demonstrations. In 2018 IEEE International Conference on Robotics and Automation (ICRA) (pp. 6292-6299). IEEE.

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. 34th International Conference on Machine Learning, ICML 2017, 6, 4261–4270.

Perez-Liebana, Diego, et al. "General video game ai: Competition, challenges and opportunities." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 30. No. 1. 2016.

Peters, J., & Bagnell, J. A. (2010). Policy Gradient Methods. *Scholarpedia*, *5*(11), 3698.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140-1144.

Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., ... & Riedmiller, M. (2017). Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. arXiv preprint arXiv:1707.08817.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature, 575(7782), 350-354.

Zhang, C., Vinyals, O., Munos, R., & Bengio, S. (2018). A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.

Glossary

Assistive reinforcement learning methods	Methods which are used with PPO and are supporting the learning. Such as BC, GAIL, curiosity, and RND
BC	Behavior Cloning. An imitation learning technique.
Dense rewards environment	Reinforcement training environment where rewards are frequent and easy to encounter.
Extrinsic rewards	Rewards given to agent by PPO
GAIL	Generative Adversarial Imitation Learning
Intrinsic rewards	Rewards given to agent by Curiosity, GAIL or RND
RND	Random network distillation
Sparse rewards environment	Reinforcement training environment where rewards are rare and difficult to encounter.
Reinforcement learning agent	Reinforcement learning agent observes the environment and takes actions and learns from rewards.

Appendix A: Hyperparameters used in the city environment.

Hyperparameters	Parameter value
batch_size	128
buffer_size	2048
learning_rate	0.0003
beta	0.01
epsilon	0.2
lambd	0.95
num_epoch	3
learning_rate_schedule	linear

Reward Signal: extrinsic (PPO)	Parameter value
gamma	**0.9**
strength	**1.0**
num_layers	**2**
vis_encode_type	**simple**

Reward Signal: Curiosity	Parameter value
gamma	0.99
strength	0.02
hidden_units	256

Reward Signal: RND	Parameter value
gamma	0.99
strength	0.01
hidden_units	64
num_layers	3
learning_rate	0.0001

Reward Signal: BC	Parameter value
steps	150000
strength	0.5

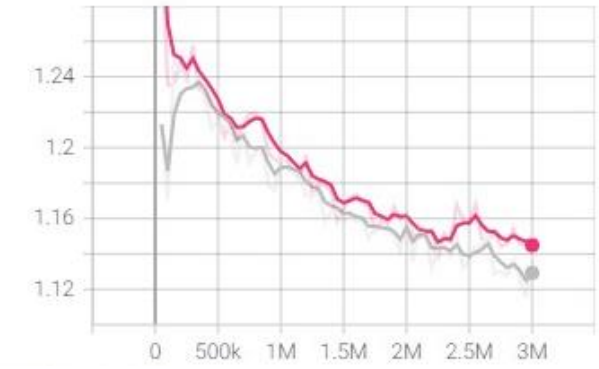
Network Settings	Parameter value
normalize	**false**
hidden_units	**512**
num_layers	**2**

Reward Signal: GAIL	Parameter value
gamma	0.99
strength	0.1

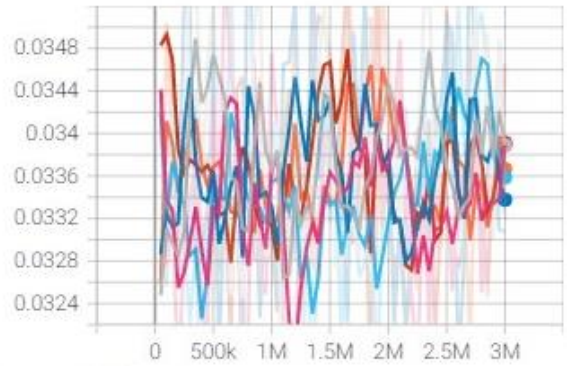
Appendix B: The results from the racing track environment

- Orange: PPO + RND
- Blue: PPO
- Red: PPO + Curiosity
- Light blue: PPO + Behavior Cloning
- Pink: GAIL + Cur

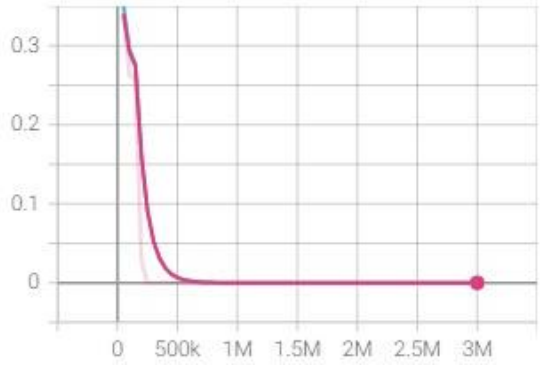
GAIL Loss
tag: Losses/GAIL Loss



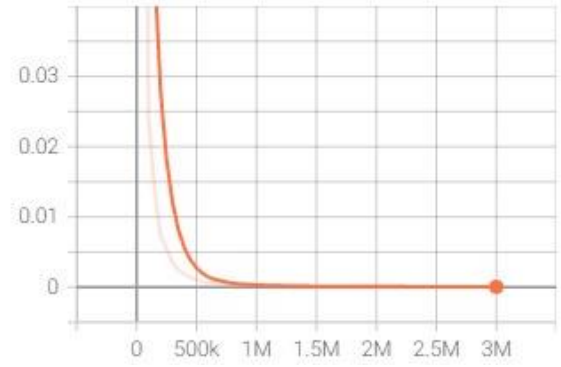
Policy Loss
tag: Losses/Policy Loss



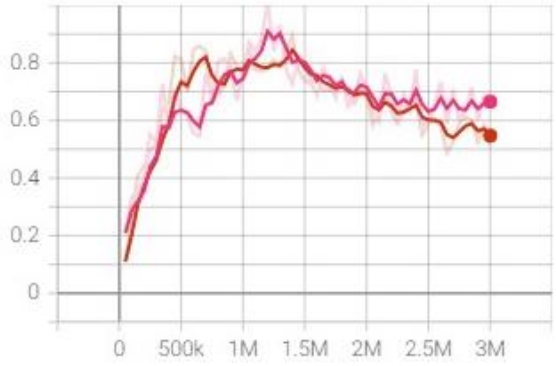
Pretraining Loss
tag: Losses/Pretraining Loss



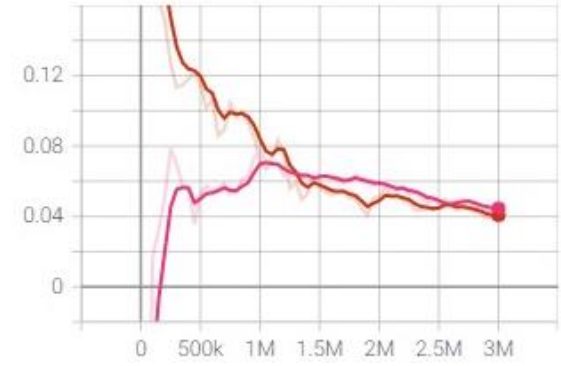
RND Loss
tag: Losses/RND Loss



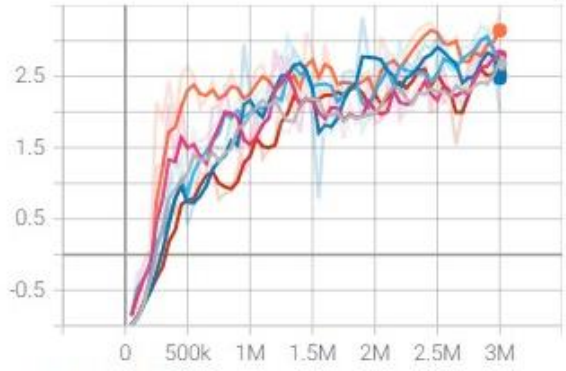
Curiosity Reward
tag: Policy/Curiosity Reward



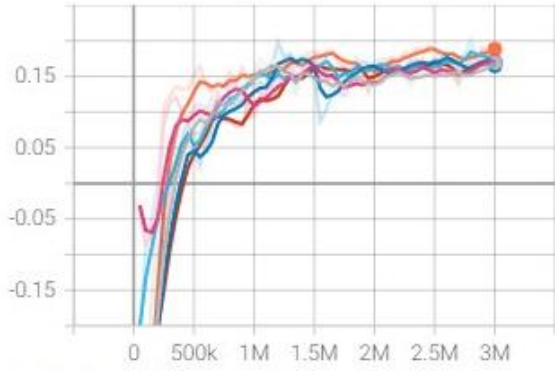
Curiosity Value Estimate
tag: Policy/Curiosity Value Estimate



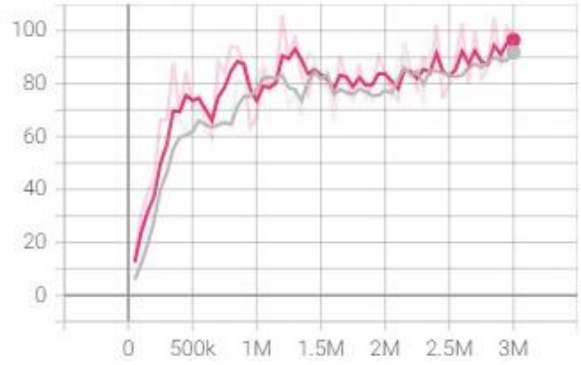
Extrinsic Reward
tag: Policy/Extrinsic Reward



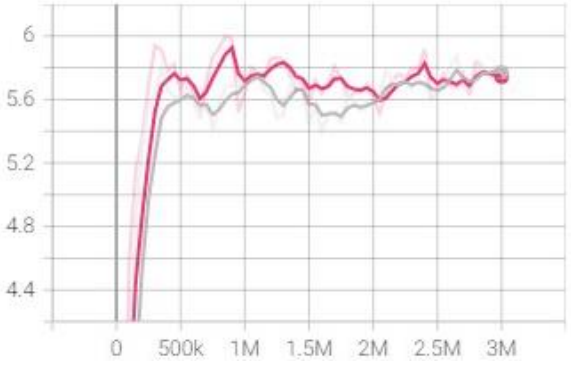
Extrinsic Value Estimate
tag: Policy/Extrinsic Value Estimate



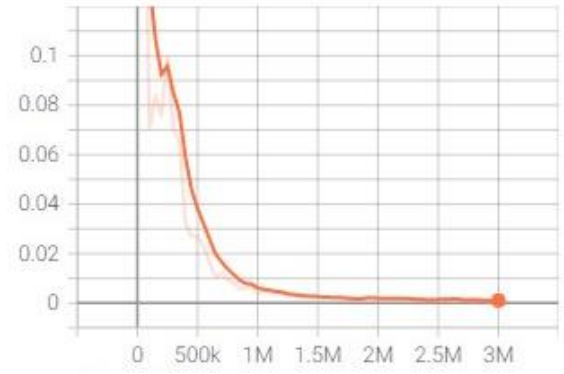
Gail Reward
tag: Policy/Gail Reward



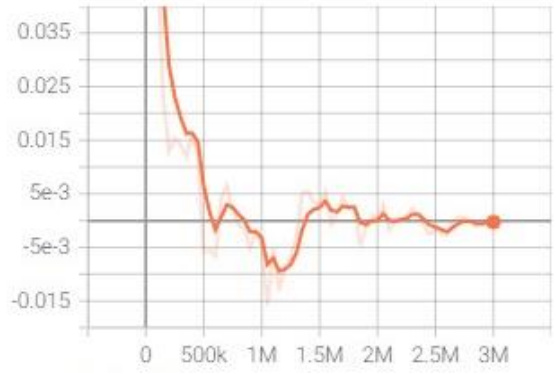
Gail Value Estimate
tag: Policy/Gail Value Estimate



Rnd Reward
tag: Policy/Rnd Reward

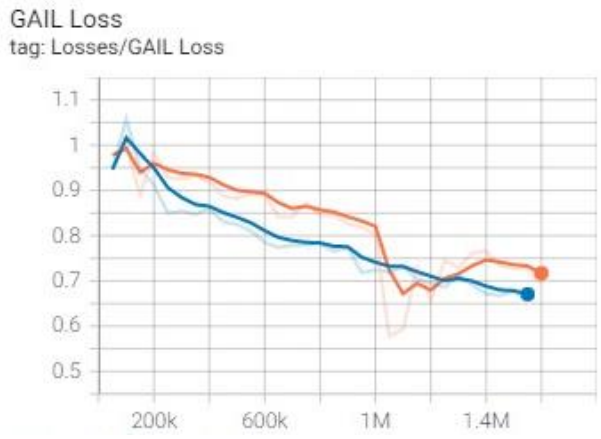


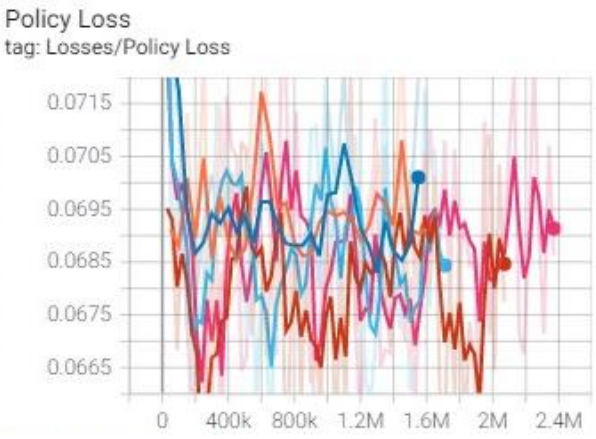
Rnd Value Estimate
tag: Policy/Rnd Value Estimate

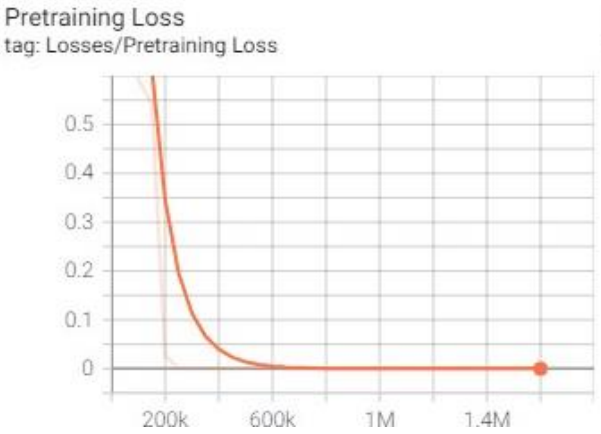


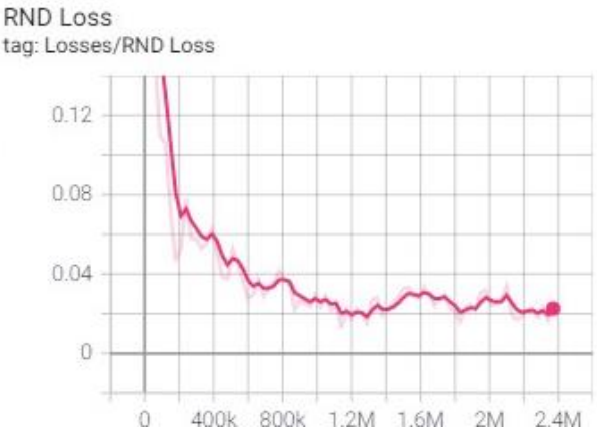
Appendix C: The results from the city environment

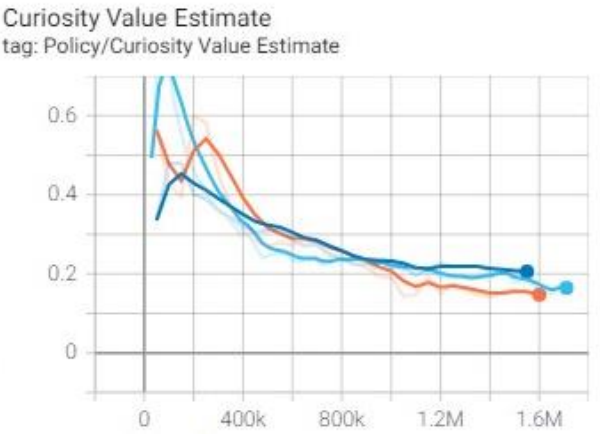
- Orange: PPO + GAIL + BC + Curiosity
- Blue: PPO + GAIL + Curiosity
- Red: PPO
- Light blue: PPO + Curiosity
- Pink: PPO + RND



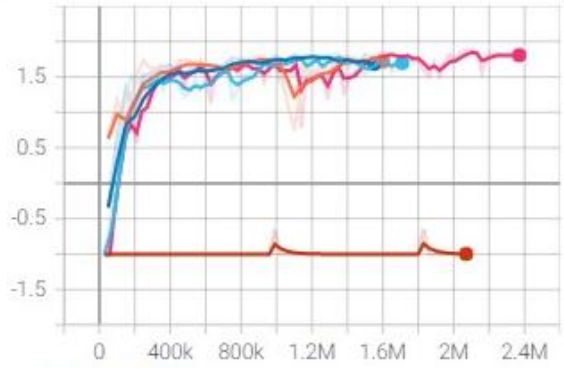




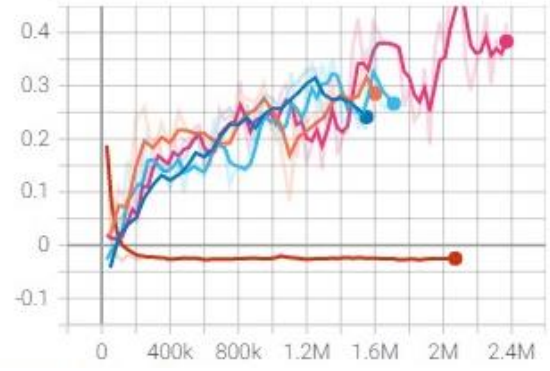




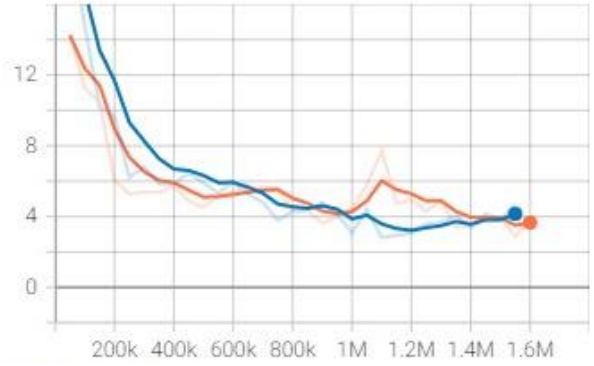
Extrinsic Reward
tag: Policy/Extrinsic Reward



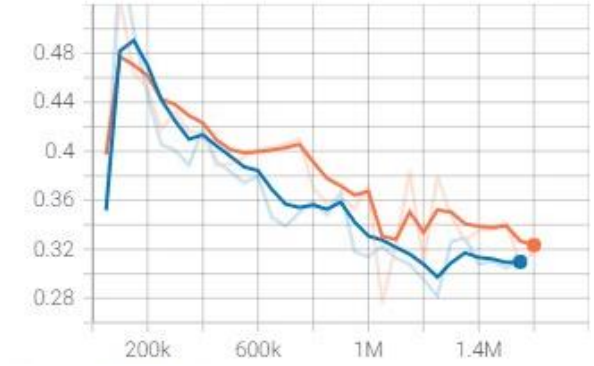
Extrinsic Value Estimate
tag: Policy/Extrinsic Value Estimate



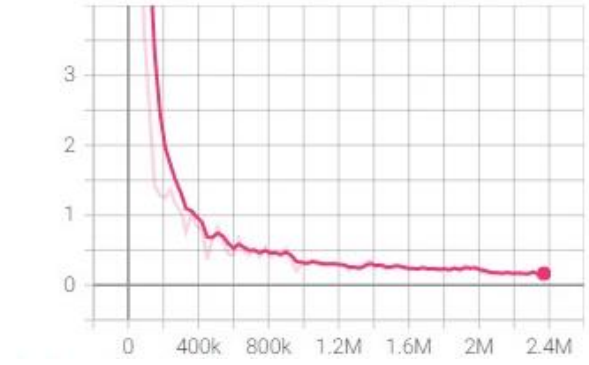
Gail Reward
tag: Policy/Gail Reward



Gail Value Estimate
tag: Policy/Gail Value Estimate



Rnd Reward
tag: Policy/Rnd Reward



Rnd Value Estimate
tag: Policy/Rnd Value Estimate