School of Electronic Engineering
and Computer Science

Final Report

Programme of study:
BSc Computer Science

Project Title:
Exploration in Reinforcement learning

Supervisor:
Paulo Rauber

Student Name:
Ahmed Karim

Final Year
Undergraduate Project 2022/23

Date: Add submission date

Queen Mary
University of London

Abstract

This project considers the problem of exploration in reinforcement learning(RL). RL has shown promising results in recent years and has been able to solve complex problems in high dimensional non-linear environments, however, it suffers from two key problems which are sample efficiency and generalization across unseen tasks. The aim of this project is to investigate more efficient ways the agent can explore its environment which is vital when an agent is deployed in a continuous state action space.

RL has had tremendous progress in recent years, especially with the introduction of deep learning into the field of RL. Examples include beating the world champion in the game of Go (Silver et al., 2016) and solving a plethora of Atari games (Mnih et al., 2013), including hard exploration games such as Montezuma revenge (AP Badia et al., 2020b). Deep learning models are remarkably good at modelling non-linear functions and generalization. They are also known for their low sample efficiency, often requiring huge amounts of computational resources to solve complex tasks. This of course is not ideal if we want RL agents to be deployed into the real world such as robotics where it must deal with continuous state action spaces and is far more complex than any Atari game would be.

In this project, I emphasise that for reinforcement learning to be applied to the real world such as robotics, it is vital that the issues of sample efficiency and generalization are solved. I suggest using model-based reinforcement learning which has recently shown promising advances in both these problems. I also use curiosity driven exploration which has shown good results when dealing with sparse reward settings. Sparse rewards is a very important issue to tackle in reinforcement learning since many aspects of real life do not have immediate rewards to them so there must be other ways humans learn from their environment.

Contents

Chapter 1: **Introduction**

1.1 Background

Traditional RL tends to mean model free RL which tries to find the solution to a problem purely from trial and error without building a model of the environment. Model-free RL has shown excellent performance in simulated environments such as video games but they suffer from high sample complexity so they require too many interactions with the environment before they can get meaningful behaviour. There is also the issue of generalizability with model free methods as they fail to generalize when they come across unfamiliar states. This contrasts with model-based methods which first builds a model of the environment the agent is in through interacting with it and then uses this internal model to derive a policy (or planning using the learned model) (Schrittwieser et al., 2020). Since the agent can understand the dynamics of its environment, it should be able to generalize to unseen tasks. To get a better understanding, a contextual scenario can be considered where a human may have never seen a car accident, or been in one, but despite this, they know that driving a car into a brick wall is a bad idea due to their understanding of the physics of the world and the ability to plan in their head what the consequences of their actions would be. This model is built into the brain gradually with other inputs through interactions with the environment. In this sense model-based RL also has model-free elements to it. This is in essence why building an internal model can prove to be beneficial. It will allow agents to act safely in the real world if executed correctly.

Model based methods can be broken down into two stages. The first stage is learning the model of the environment and the second stage is using the internal model to derive a policy. To build a model of the world the agent needs to gather sufficient information by interacting with the real environment and this is where efficient exploration is needed. The agent must be able to seek out novel states in the environment that give it the most information to build an accurate internal model. Efficient exploration is also important for the second stage too where the agent needs to derive a good policy from the internal model as it needs to know which states to seek out to get the most reward and information about the environment.

The most successful model-based RL algorithm applied to continuous state action spaces is DreamerV2 (Hanfer et al., 2020) . Typically, the benefit of model-based RL is the ability to plan its actions as it can see the outcome in its internal model and so knows whether to execute these actions or not. In DreamerV2 however, it was not just that it was planning but that it was doing model free RL in imagination. This has a lot of advantages since it can train a model free agent without having to interact with the real world which results in massive sample efficiency gains as presented by the results. DreamerV1 (Hanfer et al., 2019) has been applied to robotics (Wu et al., 2022) where it has been able to train a quadruped robot to do various movements such as walk, stand etc all in a small amount of time. As the correctness of the model cannot be guaranteed in complex tasks since models are subject to abstraction and estimation. Using model free techniques to let the agent learn beyond the inaccuracies of the model would be the next step. A key question would be what aspects of the environment can be modelled and which aspects should be dealt with model free learning.

1.2 Problem Statement

In reinforcement learning, when agents can explore environments efficiently they learn more effectively. That is, when the agent explores efficiently, it will require fewer interactions with the environment to master the task at hand. This is especially important in environments where the agent does not have access to a lot of rewards to learn from, as rewards are the main learning signal for reinforcement learning agents. One way to then encourage (efficient) exploration is to provide the agent with intrinsic motivation, in addition to extrinsic motivation (rewards from the environment). An example of intrinsic motivation is 'episodic memory', where we give the agent a memory in which it stores all states it has visited in each episode, which it can then use to prioritize visiting states it has not visited in that episode. As such, the agent has an additional motivation for taking certain actions and visiting certain states, rather than being fully focused on rewards. Another form of intrinsic motivation is 'curiosity' where an agent will be rewarded for discovering a novel state, hence encouraging exploration in unknown areas which can potentially lead to more extrinsic rewards too down the line.

Using curiosity to encourage exploration is an intuitively appealing idea. However practically it has its limitations, as what happens when we deal with randomness in an event, the agent could forever be curious without any progress. This is due to the agent receiving an intrinsic reward every time it comes across this random event (e.g Switching TV channels) and so it has no incentive to get out of its current situation. However in what environments and settings would curiosity be beneficial. Sparse reward settings could prove to be a bad match for curiosity. We explore different environments using curiosity driven exploration to understand where curiosity should be applied and where it should be avoided.

Furthermore, it would be interesting to see the effect curiosity will have when combining it with a model-based RL algorithm such as DreamerV2.

Finally seeing a combination of DQN and RND applied to Mario environment would give us insights into how curiosity and the different ways you can measure the difference between networks will affect the performance.

1.3 Aim

The aim of the project is to gain a better understanding of efficient exploration in MBRL and curiosity-driven algorithms. This will be approached in several ways. First, we will assess the curiosity algorithm in various environments to understand its limitations and reasons for failure. These environments will range from those with dense rewards to those with sparse rewards.

Another focus will be on adding intrinsic rewards and noisy layers to the DreamerV2 agent and examining the effects of different exploration strategies other than epsilon-greedy on the agent's rewards. We want to see whether the addition of intrinsic rewards will result in an increased performance and this needs to be tested on environments with and without sparse rewards (though this is limited due to computational constraints). Furthermore, we aim to explore the combination of curiosity-driven methods with other techniques in various environments. For example, we plan to combine DQN with RND and observe its impact on the Mario agent's rewards. We will also investigate if different

ways of calculating error between the two networks would affect the agent's performance.

Additionally, we will explore the potential benefits of compressing states in episodic memory when applying DQN to Pong. This will allow us to assess whether such compression would have a positive effect on the agent's performance. If the performance is promising and does not result in a decrease, we will move on to add this to the Mario environment which is more complex. (Again, due to computational constraints I have to test it on easier environments first to see if its beneficial to try in the first place).

In summary, the project aims to delve deeper into efficient exploration in MBRL and curiosity-driven algorithms by examining their performance in various environments and in combination with other techniques. Through these investigations, we hope to generate valuable insights and potential directions for future research.

1.4 Research Questions

- Why does curiosity fail or underperform compared to other algorithms in certain environments?
- Will environments with dense reward settings prove to be counter productive when applying curiosity techniques?
- Will rewards increase if we leverage model-based RL methods in conjunction with curiosity driven exploration?
- Understanding the role of Episodic memory and can we instead of storing state by state, store groups of similar states together?
- How does the choice of exploration strategy, such as epsilon-greedy or others, impact the performance of model-based algorithms? (If we had more computational power we could test these on various different environments)
- Can incorporating noisy layers in the DreamerV2 agent improve its exploration and adaptability to various environments?
- How does the combination of different curiosity-driven methods, like ICM and RND, influence the performance of reinforcement learning agents in complex tasks?
- How does the choice of error calculation method between the two networks (e.g., in the DQN+RND combination) affect the agent's performance?
- Are there specific environmental characteristics or task properties that make curiosity-driven exploration particularly effective or ineffective?

Chapter 2: **Literature Review**

2.1 Review

2.1.1 Reinforcement learning

Reinforcement learning (RL) as explained by Sutton, Barto (2018) is learning to map situations to actions to try and maximize a numerical reward signal. RL is a type of machine learning that enables an agent to learn from its environment by interacting with it. In RL, the agent is not explicitly told what actions to take to achieve a goal, but instead learns to make decisions based on the feedback it receives from the environment in the form of a numerical reward signal. The goal of the agent is to learn to take actions that will maximize its long-term cumulative reward. This is achieved through a trial-and-error process where the agent explores the environment, taking different actions and observing the resulting rewards.

The agent's behaviour is guided by a policy, which is a mapping of states to actions. The policy is updated over time as the agent learns from its experiences. The agent uses a value function to evaluate the expected future reward associated with being in a particular state and taking a particular action. The value function is used to guide the agent's exploration and to update the policy.

RL agents have two primary components: perception and action (David Abel thesis). Perception involves processing information from the environment, which includes sensory inputs, such as visual or audio data, or numerical data, such as stock prices or sensor readings. Action involves selecting a course of action based on the agent's current state and the policy it has learned.

RL has applications in a wide range of fields, including robotics, game playing, and decision making. RL algorithms have been used to teach robots to navigate through complex environments, to develop strategies for playing games like chess and Go, and to optimize resource allocation in industries like finance and energy.

Overall, RL provides a powerful framework for building agents that can learn to make decisions based on feedback from their environment, making it a valuable tool for tackling complex problems where traditional programming approaches may not be effective.

2.1.2 States and Rewards

In reinforcement learning, the environment is the external system with which the agent interacts, while a state refers to the current situation or condition of the environment at a particular time. A state can be thought of as a snapshot of the environment, which captures all the relevant information needed for the agent to make decisions. This includes factors such as the position of the agent, the presence of obstacles, the time of day, the weather conditions, and so on.

Rewards are the feedback mechanism used by the environment to guide the behaviour of the agent. They are numerical values that indicate how well the agent is performing in achieving its goals. The agent's primary objective is to maximize the total reward it receives over time, which is why it needs to learn to associate actions with rewards.

Overall, the concept of states and rewards is essential in reinforcement learning, as they provide the foundation for the agent to learn from its environment and improve its decision-making capabilities over time.

2.1.3 Model Based Reinforcement Learning – DreamerV2

Reinforcement Learning (RL) has shown impressive results in various fields such as game playing, robotics, and autonomous driving. However, traditional RL approaches have limitations when it comes to sample complexity and generalization to new tasks. Model-free RL algorithms, such as Deep Q-Networks have achieved impressive performance in Atari games (Mnih et al., 2013), but they require a large number of interactions with the environment, which is not feasible for more complex real-world continuous state-action spaces.

Model-based RL (MBRL) has proven to be more data-efficient and generalizable when learning control tasks. However, it is challenging to use MBRL in domains with high-dimensional complex observations, such as images. The DreamerV2 (Hanfer et. al., 2021) algorithm addresses these challenges by solving long-horizon tasks from images purely from latent imagination.

The DreamerV2 algorithm builds upon the Recurrent State-Space Model (RSSM) used in DreamerV1(Hanfer et. al., 2019). However, it uses categorical variables instead of Gaussian variables, allowing the model to reason about the world in a discrete way, leading to more accurate future predictions. Additionally, DreamerV2 incorporates KL balancing, which allows predictions to move more quickly towards the representations, which is essential for good planning. DreamerV2 does not plan the way usual planning algorithms work but instead uses model-free RL very quickly because it operates in "imagination."

The use of latent states is another significant contribution of DreamerV2. During training, an encoder converts each image into a stochastic representation that is incorporated into the recurrent state of the model. From each state, a decoder reconstructs the corresponding image to learn general representations. A Gated Recurrent Unit (GRU) is used to learn the latent state transitions. Given a latent state and action, the agent tries to predict the reward received and what the game would look like after taking the action, which is trained using Back Propagation Through Time (BPTT).

MBRL has achieved better performance than model-free methods while being able to generalize across unseen tasks better and with much fewer samples. (Ha, Schemidhourr,2018) demonstrated the possibility of training an agent to perform tasks entirely in a simulated latent space world. This is essential as it is computationally expensive to train agents directly from pixels, and there must be a way to compress the pixel representation into a latent one and use it for training. However, VAEs used to train the model have their limitations since they may encode parts of the observations that are not relevant to the task at hand.

In summary, DreamerV2 is a model-based RL algorithm that operates in the latent space, making it more data-efficient and generalizable. It utilizes categorical variables, KL balancing, and a GRU to learn latent state transitions. The use of latent states and imagination allows for better generalization and faster training of RL agents. However, there are still challenges to overcome, such as encoding relevant information from observations into the latent space, which requires further research.

2.1.3.2 MuZero V DreamerV2

In addition to DreamerV2, another prominent model-based RL algorithm is MuZero (Schrittwieser et al., 2020), which has demonstrated impressive performance across

various domains, including board games like Go, Chess, and Shogi, as well as Atari games. MuZero extends the ideas of AlphaZero by learning a model of the environment instead of assuming perfect knowledge of the game dynamics. It does so by learning a latent dynamics model, which predicts the next latent state, immediate reward, and policy distribution given the current latent state and action. This allows the algorithm to plan efficiently using Monte Carlo Tree Search (MCTS), making it applicable to a wide range of environments.

One of the primary advantages of MuZero over DreamerV2 is its ability to handle partially observable environments and environments with complex dynamics. DreamerV2, while being a powerful algorithm for continuous control tasks, has certain limitations when it comes to handling partially observable environments. The primary reason for this is that DreamerV2 assumes that the environment's dynamics can be effectively modelled and learned using a deterministic or probabilistic world model. In partially observable environments, the agent does not have access to the complete state information, making it more challenging to learn an accurate world model.

MuZero's use of MCTS for planning allows it to explore and exploit the environment more effectively, leading to better policies. However, MuZero can be computationally expensive due to its reliance on MCTS, which may limit its applicability in real-time settings or resource-constrained problems.

On the other hand, DreamerV2 focuses on learning a world model using latent variables and generates imagined trajectories to train a policy and value function in a model-free manner. This approach leads to substantial sample efficiency improvements, as the agent can learn from imagined experiences without interacting with the real environment. Furthermore, DreamerV2 has demonstrated success in continuous control tasks, such as robotics, where sample efficiency and real-time performance are critical.

In this report, we have chosen to explore DreamerV2 due to its balance of performance, sample efficiency, and applicability to continuous control tasks. While MuZero has shown remarkable results in certain domains, its reliance on MCTS may not be suitable for all environments, particularly those with strict computational requirements or continuous state-action spaces. DreamerV2's ability to leverage both model-based and model-free techniques makes it a promising avenue for further research and development in the field of reinforcement learning.

2.2 Exploration in Reinforcement Learning

2.2.1 Efficient Exploration

In reinforcement learning, when agents can explore environments efficiently they learn more effectively. That is, when the agent explores efficiently, it will require fewer interactions with the environment to master the task at hand. This is especially important in environments where the agent does not have access to a lot of rewards to learn from, as rewards are the main learning signal for reinforcement learning agents. One way to then encourage (efficient) exploration is to provide the agent with intrinsic motivation, in addition to extrinsic motivation (rewards from the environment). An example of intrinsic motivation is 'episodic memory', where we give the agent a memory in which it stores all states it has visited in each episode, which it can then use to explore more novel states as these novel states would not be present in the episodic memory. As such, the agent

has an additional motivation for taking certain actions and visiting certain states, rather than being fully focused on rewards.

Using episodic memory to encourage exploration is an intuitively appealing idea. However, practically it has its limitations, as it becomes difficult to compare and keep track of states in high dimensional complex environments with large numbers of possible states. In such settings, we need to make different considerations than in tabular settings, where our episodic memory could simply be a table where we look up if a certain state has been visited or not. Random exploration techniques such as epsilon greedy are fine in tabular environments since the agent can visit each state multiple times. Though intuitively, we can see why this would fail in complex continuous state spaces. When an agent traverses through a continuous state space, it will never (negligible probability) come across the exact same state twice. Recent works (Badia et al., 2020a) have presented methods to measure similarity between states and explore the environment based on how novel the states it comes across are. This is typically referred to as exploration via curiosity. A forward model would be used to predict the next state and depending how far the prediction was off the actual state, the agent would be rewarded. This led to agents that were able to solve Montezuma's revenge amongst other hard exploration games. Random Network Distillation techniques (Burda et al., 2018) have also been used to avoid the issue of stochasticity of the environment which resulted in the agent being in a forever loop of being curious, without any meaningful exploration although this is only a temporary solution since the real world is filled with random events we cannot predict.

2.2.2 Curiosity Driven Exploration

Curiosity-driven exploration is a concept in reinforcement learning that enhances an agent's exploration in its environment by prioritizing actions that lead to the acquisition of new and diverse knowledge. This approach allows agents to learn faster and more efficiently by discovering novel strategies and behaviours, instead of relying solely on external rewards.

As previously mentioned, traditional RL methods often rely on random exploration, which can be inefficient, especially in complex environments. Curiosity-driven exploration provides a more targeted approach by using intrinsic motivation or novelty-based techniques to direct exploration. Intrinsic motivation arises from within an agent, encouraging exploration based on its curiosity and desire to learn. Novelty-based exploration prioritizes actions that lead to novel or previously unexplored areas of the state space.

Two popular curiosity-driven exploration techniques are Intrinsic Curiosity Module (ICM) and Random Network Distillation (RND). ICM learns a forward and inverse dynamics model of the environment, with curiosity-driven rewards derived from the error in the forward model's prediction. Mathematically, the intrinsic reward can be defined as:

$$Reward\_ICM = \eta * ||F(s, a) - s'||^2$$

where F(s, a) is the forward model's prediction of the next state given the current state s and action a, s' is the actual next state, and η is a scaling factor.

RND uses a fixed random neural network as a target and trains a separate predictor network to mimic the target network's output. The intrinsic reward is based on the prediction error between the target and predictor networks:

$$Reward\_RND = \kappa * ||T(s) - P(s)||^2$$

where T(s) is the target network's output, P(s) is the predictor network's output, and κ is a scaling factor.

Curiosity-driven exploration is particularly useful in environments where external rewards are sparse or difficult to define. By prioritizing exploration and learning over immediate rewards, agents can discover new strategies and behaviours that lead to more significant rewards in the future.

In conclusion, curiosity-driven exploration, as implemented in techniques like ICM and RND, is a promising approach in reinforcement learning. By encouraging agents to explore uncertain, novel, or intrinsically motivating areas of the state space, curiosity-driven exploration can improve the efficiency and effectiveness of RL methods, especially in complex and challenging environments.

2.2.3 Exploring in Latent States

Exploring directly from pixel inputs is computationally expensive and recent work has been done to try and solve this issue. VAEs (Ha, Schmidhuber 2018) have been used to compress the pixel input into a low dimensional latent vector. The compressed representation can be used to reconstruct the original image but the issue with VAEs however is that it may encode irrelevant information from the observation or leave out important information. Other methods include random features and inverse dynamic features but these each have their own trade-offs either including irrelevant features or failing to be sufficient. Being able to compress pixel inputs is critical. Mapping features that are completely not under the control of the agent is not needed. An example from (Pathak et al., 2017) portrays this well. Imagine an agent observing movement of tree leaves in a breeze. It's very hard to model the breeze, predicting pixel changes for each leave would be impossible and so the prediction error in pixel space would always be high leaving the agent forever curious. The agent needs to understand that there are some parts of the environment that it simply cannot control or predict. The intrinsic curiosity model tries to solve this issue. The problem however is that the feature encoders could get rid of features that are not directly influenced by the agent and this will result less than optimal exploration strategy.

2.2.4 Episodic Curiosity through reachability

In reinforcement learning, agents are typically rewarded for actions that lead to positive outcomes, while receiving no or negative rewards for actions that lead to negative outcomes. However, in cases where the agent is far from its objective, there may not be a well-defined reward signal to guide the agent, leading it to potentially wander aimlessly. To address this, researchers have proposed rewarding agents for exploring new parts of the environment, an approach known as curiosity-driven exploration.

One such approach is episodic curiosity through reachability, as described in Savinov et al. (2018). To determine if it has visited a particular state before, the agent stores snapshots of previously visited states in memory. However, directly comparing these snapshots to the current observation can be challenging, as the agent may encounter the same place from different perspectives. Instead, the agent uses a neural network to predict the next state based on the current observation. If the prediction is accurate, the state is deemed familiar, and the agent receives little to no reward. Conversely, if the prediction is inaccurate, the state is considered novel, and the agent receives an intrinsic reward for exploring it.

Episodic curiosity is preferable to surprised curiosity, which can lead the agent to become stuck in a state of perpetual curiosity, such as when faced with a television constantly changing channels at random. By incorporating an episodic memory, the agent can eventually exhaust all possibilities, leading to continued exploration.

In this approach, novelty is determined through reachability. The agent begins with an empty memory and, at each time step, compares the current observation to those stored in memory to assess novelty. If the current observation is within a certain number of steps of any observation in memory, it is deemed familiar, and no reward is given. However, if the current observation requires more steps to reach than the designated threshold, the agent receives an intrinsic reward for exploring the novel state.

2.2.5 Gaussian

Gaussian exploration is another approach to encourage exploration in reinforcement learning, allowing agents to probe their environment more effectively, which can lead to faster learning and better performance. Gaussian exploration is particularly useful in continuous action spaces, where adding Gaussian noise to the selected action encourages exploration by perturbing the actions with a degree of randomness.

Traditional RL methods often rely on random exploration, which can be inefficient, especially in complex environments. Gaussian exploration provides a more targeted approach by adding noise to the actions chosen by the policy. This encourages the agent to explore nearby actions, potentially discovering more effective strategies and behaviours.

Gaussian exploration is particularly useful in environments where the optimal actions are continuous and can be discovered by exploring the action space in a smooth, local manner. By adding Gaussian noise to the policy actions, agents can discover new strategies and behaviours that lead to improved performance.

In conclusion, Gaussian exploration, as implemented by adding Gaussian noise to policy actions, is a promising approach in reinforcement learning. By encouraging agents to explore the action space in a smooth, local manner, Gaussian exploration can improve the efficiency and effectiveness of RL methods, especially in complex and challenging environments with continuous action spaces.

2.2.6 Noisy Linear Layers

Noisy Linear Layers is an approach in reinforcement learning that enhances an agent's exploration by adding noise directly to the weights of the network during training. This allows agents to explore more efficiently and adaptively by incorporating exploration directly into the learning process. The exploration strategy naturally adjusts as the agent learns the environment, instead of relying solely on external rewards or fixed exploration rates such as epsilon greedy.

Traditional RL methods like epsilon-greedy often rely on random exploration, which can be inefficient, especially in complex environments. Noisy Linear Layers provide a more targeted approach by using a stochastic component directly in the network's weights. This way, exploration is inherently tied to the learning process and adjusted adaptively based on the agent's understanding of the environment.

Noisy Linear Layers work by adding noise to the weights of the layers in the network. Instead of having fixed weights, the weights are perturbed by noise during each forward pass. This noise is drawn from a distribution (e.g., Gaussian) and added to the weights. The parameters of the noise distribution are learned during training, allowing the network to adapt the level of exploration based on its experience.

To use Noisy Linear Layers with epsilon-greedy exploration, you can replace the standard linear layers in your network with noisy ones. The noise will introduce exploration into the action selection process, making the epsilon-greedy exploration rate

less critical. The agent will automatically balance exploration and exploitation through the learned noise parameters.

Chapter 3: **Additional Sections**

3.1 Project Execution and Aims

The primary aim of this project is to evaluate and experiment with curiosity-driven algorithms to identify their limitations. For instance, we plan to test their performance in sparse reward settings versus dense reward settings and explore different hyperparameters in diverse environments. However, due to computational constraints, we will have to settle for simple environments. One example of a complex environment given that we had enough resources would be a scenario where the agent must navigate around obstacles to reach a goal and then train two agents, one with a standard RL algorithm and the other with an ICM-based algorithm. We would then record videos of each agent navigating the environment and compare their performances. We would expect the ICM-based agent to navigate the environment more efficiently and with fewer collisions. However of course as mentioned earlier, simpler environments will suffice for now.

Moreover, I plan to take a more ambitious approach by combining curiosity-driven exploration algorithms such as RND with DreamerV2, which is a model-based RL algorithm. The objective is to determine how intrinsic rewards affect the performance of the DreamerV2 agent. For instance, I intend to investigate if leveraging the look-ahead ability of DreamerV2 could enable us to calculate cumulative novelty or cumulative intrinsic rewards. Practically, this means that we may find that going through familiar states initially may bring more novelty later. The focus will be on how these intrinsic rewards affect the total extrinsic rewards.

3.2 Software

3.2.1 PyTorch

Although the official implementation of DreamerV2 is written in TensorFlow, I have opted to use a re-implementation of DreamerV2 in PyTorch for this project due to my previous experience with the framework. Similarly, I have also re-implemented the ICM algorithm in PyTorch, albeit in its simplified version, to gain a better understanding of the algorithm. However, when combining these two algorithms, it would be prudent to use the official implementation of ICM to ensure optimal performance and prevent any potential bugs.

3.2.2 Open AI Gym

This is where most the environments being tested on will come from. It is the standard benchmarks used for testing reinforcement learning agents and is very well documented.

Chapter 4: **Presentation**

4.1 Source Code

There is much more to the code then provided here but I included some key elements. This project was a mix of my own implementations and current GitHub implementations.

```
class ReplayBuffer(object):
    def __init__(self, max_size, input_shape,  compression_factor=5):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.compression_factor = compression_factor
        self.state_memory = np.zeros((self.mem_size, *input_shape),
                                     dtype=np.float32)
        self.new_state_memory = np.zeros((self.mem_size, *input_shape),
                                         dtype=np.float32)

        self.action_memory = np.zeros(self.mem_size, dtype=np.int64)
        self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done
        self.mem_cntr += 1

        if self.mem_cntr % self.mem_size == 0:
            self.compress_states()

    def compress_states(self):
        compressed_size = self.mem_size // self.compression_factor
        for i in range(compressed_size):
            start_idx = i * self.compression_factor
            end_idx = start_idx + self.compression_factor
            self.state_memory[i] = np.mean(self.state_memory[start_idx:end_idx], axis=0)
            self.new_state_memory[i] = np.mean(self.new_state_memory[start_idx:end_idx], axis=0)
            self.action_memory[i] = np.random.choice(self.action_memory[start_idx:end_idx])
            self.reward_memory[i] = np.mean(self.reward_memory[start_idx:end_idx])
            self.terminal_memory[i] = np.any(self.terminal_memory[start_idx:end_idx])

        self.mem_cntr = compressed_size
```

```python
class DeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, name, input_dims, chkpt_dir):
        super(DeepQNetwork, self).__init__()
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name)

        self.conv1 = nn.Conv2d(input_dims[0], 32, 8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, 3, stride=1)

        fc_input_dims = self.calculate_conv_output_dims(input_dims)

        self.fc1 = nn.Linear(fc_input_dims, 512)
        self.fc2 = nn.Linear(512, n_actions)

        self.optimizer = optim.RMSprop(self.parameters(), lr=lr)

        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def calculate_conv_output_dims(self, input_dims):
        state = T.zeros(1, *input_dims)
        dims = self.conv1(state)
        dims = self.conv2(dims)
        dims = self.conv3(dims)
        return int(np.prod(dims.size()))

    def forward(self, state):
        conv1 = F.relu(self.conv1(state))
        conv2 = F.relu(self.conv2(conv1))
        conv3 = F.relu(self.conv3(conv2))
        # conv3 shape is BS x n_filters x H x W
        conv_state = conv3.view(conv3.size()[0], -1)
        # conv_state shape is BS x (n_filters * H * W)
        flat1 = F.relu(self.fc1(conv_state))
        actions = self.fc2(flat1)
```

```python
import numpy as np
import torch as T
from deep_q_network import DeepQNetwork
from replay_memory import ReplayBuffer

class DQNAgent(object):
    def __init__(self, gamma, epsilon, lr, n_actions, input_dims,
                 mem_size, batch_size, eps_min=0.01, eps_dec=5e-7,
                 replace=1000, algo=None, env_name=None, chkpt_dir='tmp/dqn', compression_factor=5):
        self.gamma = gamma
        self.epsilon = epsilon
        self.lr = lr
        self.n_actions = n_actions
        self.input_dims = input_dims
        self.batch_size = batch_size
        self.eps_min = eps_min
        self.eps_dec = eps_dec
        self.replace_target_cnt = replace
        self.algo = algo
        self.env_name = env_name
        self.chkpt_dir = chkpt_dir
        self.action_space = [i for i in range(n_actions)]
        self.learn_step_counter = 0

        self.memory = ReplayBuffer(mem_size, input_dims, n_actions, compression_factor)

        self.q_eval = DeepQNetwork(self.lr, self.n_actions,
                                   input_dims=self.input_dims,
                                   name=self.env_name+'_'+self.algo+'_q_eval',
                                   chkpt_dir=self.chkpt_dir)

        self.q_next = DeepQNetwork(self.lr, self.n_actions,
                                   input_dims=self.input_dims,
                                   name=self.env_name+'_'+self.algo+'_q_next',
                                   chkpt_dir=self.chkpt_dir)


    def choose_action(self, observation):
        if np.random.random() > self.epsilon:
            state = T.tensor([observation],dtype=T.float).to(self.q_eval.device)
```

```
def choose_action(self, observation):
    if np.random.random() > self.epsilon:
        state = T.tensor([observation],dtype=T.float).to(self.q_eval.device)
        actions = self.q_eval.forward(state)
        action = T.argmax(actions).item()
    else:
        action = np.random.choice(self.action_space)

    return action

def store_transition(self, state, action, reward, state_, done):
    self.memory.store_transition(state, action, reward, state_, done)

def sample_memory(self):
    state, action, reward, new_state, done = \
                                self.memory.sample_buffer(self.batch_size)

    states = T.tensor(state).to(self.q_eval.device)
    rewards = T.tensor(reward).to(self.q_eval.device)
    dones = T.tensor(done).to(self.q_eval.device)
    actions = T.tensor(action).to(self.q_eval.device)
    states_ = T.tensor(new_state).to(self.q_eval.device)

    return states, actions, rewards, states_, dones
```

```python
class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, sigma_init=0.017):
        super(NoisyLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.sigma_init = sigma_init

        self.weight = nn.Parameter(torch.Tensor(out_features, in_features))
        self.bias = nn.Parameter(torch.Tensor(out_features))
        self.weight_noise = nn.Parameter(torch.Tensor(out_features, in_features), requires_grad=False)
        self.bias_noise = nn.Parameter(torch.Tensor(out_features), requires_grad=False)

        self.reset_parameters()
        self.reset_noise()

    def forward(self, x):
        if self.training:
            weight = self.weight + self.weight_noise * self.sigma_init
            bias = self.bias + self.bias_noise * self.sigma_init
        else:
            weight = self.weight
            bias = self.bias
        return torch.nn.functional.linear(x, weight, bias)

    def reset_parameters(self):
        std = math.sqrt(3 / self.in_features)
        nn.init.uniform_(self.weight, -std, std)
        nn.init.uniform_(self.bias, -std, std)

    def reset_noise(self):
        noise = torch.randn_like(self.weight)
        self.weight_noise.copy_(noise)
        noise = torch.randn_like(self.bias)
        self.bias_noise.copy_(noise)


def _build_model(self): #This is for the Noisy Network Linear
    model = [NoisyLinear(self.deter_size + self.stoch_size, self.node_size)]
    model += [self.act_fn()]
    for i in range(1, self.layers):
        model += [NoisyLinear(self.node_size, self.node_size)]
        model += [self.act_fn()]

    if self.dist == 'one_hot':
        model += [NoisyLinear(self.node_size, self.action_size)]
    else:
        raise NotImplementedError
    return nn.Sequential(*model)



def forward(self, model_state):
    action_dist = self.get_action_dist(model_state)
    action = action_dist.sample()
    action = action + action_dist.probs - action_dist.probs.detach()
    return action, action_dist

def get_action_dist(self, modelstate):
    logits = self.model(modelstate)
    if self.dist == 'one_hot':
        return torch.distributions.OneHotCategorical(logits=logits)
    else:
        raise NotImplementedError
```

```python
def add_exploration(self, state: torch.Tensor, action: torch.Tensor, itr: int, mode='train'):
    if mode == 'train':
        expl_amount = self.train_noise
        expl_amount = expl_amount - itr/self.expl_decay
        expl_amount = max(self.expl_min, expl_amount)
    elif mode == 'eval':
        expl_amount = self.eval_noise
    else:
        raise NotImplementedError

    if self.expl_type == 'epsilon_greedy':
        if np.random.uniform(0, 1) < expl_amount:
            index = torch.randint(0, self.action_size, action.shape[:-1], device=action.device)
            action = torch.zeros_like(action)
            action[:, index] = 1
        return action


    elif self.expl_type == 'gaussian':
        if np.random.uniform(0, 1) < expl_amount:
            action = torch.normal(action, expl_amount)
        return action

    elif self.expl_type == 'ucb1': #This is not designed well for BREAKOUT as its meant to be used for Multi Armed band
        with torch.no_grad():
            N = self.action_visits + 1e-9  # action_visits should be an array of visit counts for each action, avoid di
            Q = self.action_values / N  # action_values should be an array of accumulated rewards for each action
            c = np.sqrt(2 * np.log(itr + 1))  # exploration constant
            UCB = Q + c * np.sqrt(np.log(itr + 1) / N)  # calculate upper confidence bounds
            action_index = torch.argmax(UCB)  # select action with the highest upper confidence bound
            action = torch.zeros_like(action)
            action[:, action_index] = 1
        return action
```

```python
elif self.expl_type == 'boltzmann':
    with torch.no_grad():
        action_scores = self.model(state)
        action_probs = torch.softmax(action_scores / self.tau, dim=-1)  # apply temperature and get actio
        action_index = torch.multinomial(action_probs, 1).squeeze(-1)  # sample action from the distribut
        action = torch.zeros_like(action_probs)
        action[torch.arange(action_probs.shape[0]), action_index] = 1  # one-hot encode action
    return action


elif self.expl_type == 'thompson':
    if np.random.uniform(0, 1) < expl_amount:
        with torch.no_grad():
            action_logits = self.model(state)
            action_distribution = torch.distributions.categorical.Categorical(logits=action_logits)
            action_index = action_distribution.sample()
            action = torch.zeros_like(action_logits)
            action[torch.arange(action_logits.shape[0]), action_index] = 1
    return action


raise NotImplementedError
```

```python
class RND(nn.Module):
    def __init__(self, obs_shape):
        super(RND, self).__init__()
        self.target_network = nn.Sequential(
            nn.Linear(200, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 64)
        )

        self.predictor_network = nn.Sequential(
            nn.Linear(200, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 64)
        )

        for param in self.target_network.parameters():
            param.requires_grad = False

        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                nn.init.constant_(m.bias, 0)

    def forward(self, state):
        target_output = self.target_network(state)
        predictor_output = self.predictor_network(state)
        return target_output, predictor_output


def rnd_intrinsic_reward(embed: torch.Tensor, rnd_model):
    target_output, predictor_output = rnd_model(embed)
    intrinsic_reward = (target_output - predictor_output).pow(2).sum(1) / 2
    return intrinsic_reward
```

```python
import numpy as np

class Agent():
    def __init__(self, lr, gamma, n_actions, n_states, eps_start, eps_end, eps_dec):
        self.lr = lr
        self.gamma = gamma
        self.n_actions = n_actions
        self.n_states = n_states
        self.epsilon = eps_start
        self.eps_min = eps_end
        self.eps_dec = eps_dec

        self.Q = {}

        self.init_Q()


    def init_Q(self):
        for state in range(self.n_states): #Every state has multiple actions
            for action in range(self.n_actions):
                self.Q[(state,action)] = 0.0

    def choose_action(self,state):
        if np.random.random() < self.epsilon:
            action = np.random.choice([i for i in range(self.n_actions)]) # if Epsilon is 0.1 then there is a 10%
        else:
            actions = np.array([self.Q[(state, a)] \
                                for a in range(self.n_actions)])
            action = np.argmax(actions)

        return action

    def reduce_epsilon(self): #decrements epsilon
        if self.epsilon > self.eps_min:
            self.epsilon = self.epsilon * self.eps_dec
        else:
            self.epsilon = self.eps_min

    def learn(self, state, action, reward, state ):
```

```
class RND(nn.Module):
    def __init__(self, input_shape):
        super(RND, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        conv_out_shape = self.get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_shape, 512),
            nn.ReLU(),
            nn.Linear(512, 64)
        )

    def get_conv_out(self, shape):
        out = self.conv(torch.zeros(1, *shape))
        return np.prod(out.shape)

    def forward(self, x):
        x = self.conv(x).view(x.shape[0], -1)
        x = self.fc(x)
        return x
```

```
def update_model(self, minibatch):
    """
    Calculate Mean Square Error (MSE) between actual values and expected values from Deep
    Then perform stochasitc gradient descent
    """
    states, actions, rewards, next_states, _ = minibatch

    states = tensor(states).to(device)
    next_states = tensor(next_states).to(device)
    actions = tensor(actions).to(device)
    rewards = tensor(rewards).to(device)

    # Calculate intrinsic rewards
    intrinsic_rewards = tensor([self.calculate_intrinsic_reward(state) for state in state
    total_rewards = rewards + intrinsic_rewards

    Q = self.model(states)
    target_Q = self.target_model(next_states)

    # Get Q-values of actions taken for each state
    Q = Q.gather(1, actions.unsqueeze(-1)).squeeze(-1)
    target_Q = target_Q.amax(1)

    # Perform SGD
    td_target = target_Q * self.gamma + total_rewards
    loss = self.loss_function(Q, td_target)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    return Q.mean().item(), loss.item()
```

```python
class DQNAgent:
  def __init__(self):
    self.env = make_mario("SuperMarioBros-1-1-v0", COMPLEX_MOVEMENT)



    self.n_episodes = hp.N_EPISODES
    self.alpha = hp.ALPHA
    self.epsilon_start = hp.EPSILON_START
    self.epsilon_final = hp.EPSILON_FINAL
    self.epsilon = hp.EPSILON_START
    self.gamma = hp.GAMMA
    self.decay = hp.DECAY
    self.memory_size = hp.MEMORY_SIZE
    self.batch_size = hp.BATCH_SIZE
    self.min_exp = hp.MIN_EXP
    self.target_update_freq = hp.TARGET_UPDATE_FREQ

    self.input_shape = self.env.observation_space.shape
    self.n_actions = self.env.action_space.n

    self.model = DQN(self.input_shape, self.n_actions).to(device)
    self.target_model = DQN(self.input_shape, self.n_actions).to(device)
    self.optimizer = optim.Adam(self.model.parameters(), lr=self.alpha)
    self.loss_function = nn.MSELoss()

    self.random_net = RND(self.input_shape).to(device)
    self.predictor_net = RND(self.input_shape).to(device)
    self.rnd_optimizer = optim.Adam(self.predictor_net.parameters(), lr=hp.RND_ALPHA)

    self.replay_memory = ExperienceReplay(self.memory_size)
    self.logger = Logger()
```

Chapter 5: **Evaluation**

Approach:

Firstly, I decided to understand how reinforcement learning works by re-implementing the Q-Learning algorithm and then slowly increasing complexity by re-implementing the naïve DQN algorithm and testing it on various environments, most notably breakout. I wrote a naïve version without an experience replay which proved to be vital to get any meaningful performance. The results showing what the performance was like with and without experience replay is displayed below (and it was not using Conv layers).

Moving on from here, I thought there must be more efficient exploration methods when dealing with unfamiliar environments and that's where I came across curiosity driven exploration which was a very interesting concept as explained in the above sections.

First to really understand the algorithm ICM, I reimplemented the algorithm in PyTorch and got it working on the Cart Pole environment. The good thing was that ICM is not a complicated algorithm to implement yet has shown to be very effective in dealing with sparse reward environments as shown in the paper. Random Network Distillation is an even better curiosity algorithm which is more robust when it comes to stochastic events, though this is still a major issue in the research community.

Further to this I decided in order to get meaningful results and focus on the things that can enhance the algorithms performance, I went on GitHub to take a look at other implementations that were more complete then my own. Before I talk about the DreamerV2 algorithm, I want to talk about the experiments conducted which was using PPO combined with ICM in the MountainCartContinuous environment where I conducted tests with various different hyper parameters and hidden feature sizes of ICM. The results of this did show PPO_ICM performed better than PPO by itself. I ran this multiple times to confirm the results. But more on the results in the later sections.

Building upon the previous experiments, I delved into a PyTorch implementation of the DreamerV2 algorithm, which was originally developed using TensorFlow. I thoroughly examined the codebase and initiated modifications to improve the agent's performance. I began by fine-tuning various hyperparameters, such as the planning horizon length and the state representation used by the RSSM (discrete or continuous), among others, in an effort to optimize performance.

In addition to adjusting hyperparameters, I expanded the range of exploration methods available to the agent. Initially, the agent utilized only epsilon-greedy exploration. I incorporated several alternative exploration techniques, including Gaussian, Boltzmann, and others. One particularly impactful modification involved adding noise to the linear layers of the actor network. This single addition led to a significant increase in the agent's performance, highlighting the importance of diverse exploration strategies.

Furthermore, I explored the integration of curiosity-driven exploration into the DreamerV2 agent. While a comprehensive implementation of curiosity would require more time and potentially adjustments to the official implementation, I successfully incorporated an RND model to estimate intrinsic rewards. The RND model calculated these rewards based on the difference between predicted and actual encoded observations of the environment, which were then added to the total reward signal. This inclusion of curiosity-driven exploration provided another promising avenue for enhancing the performance of the DreamerV2 agent, which could be further investigated and refined in future work.

Next, I wanted to go back to testing the DQN algorithm but this time I did not use my own implementation since it was a simple version, but I used a more established implementation. First I tested DQN on the pong environment and after approximately 4-5 million steps the performance peaked. Of course, a decaying epsilon greedy was used as there is no need to explore the environment once peak performance is reached (although most the learning was done whilst there was very little exploration). With this DQN algorithm, states, actions, rewards that occurred are stored in an episodic memory which of course fills up and then deletes the oldest memory to make way for new ones. I wanted to see the affects of compressing the closest stores together into a mean, so it takes less space and keeps relevant information though the performance decreased drastically as a result, more on this later.

Finally, I wanted to see the performance of DQN on Mario and also combined the random network distillation curiosity algorithm with it (albeit a simpler version) to see the performance differences. This was interesting to see how the agent was learning to play Mario over time though this was by far the most computationally taxing, even more so then the DreamerV2 agent (because that was done on MinAtar). This of course meant I could not leave experiments going for as along as I would have liked because interesting results happens after around 15k episodes though I limited my experiments to around 4k episodes in Mario as adding RND meant I had to train more neural networks.

Algorithm Explanations:

Q-learning

Q-learning is a model-free, reinforcement learning algorithm used to find the optimal action-selection policy for any given Markov decision process (MDP). The Q-learning algorithm uses a table of Q-values to learn the value of taking a particular action in a given state, where the Q-value is defined as the expected sum of future rewards when a particular action is taken from a given state.

The Q-learning algorithm starts with an initial Q-table, where all Q-values are set to zero. The algorithm then repeatedly interacts with the environment by selecting an action based on the current state and the Q-table and receiving a reward and the next state from the environment. The Q-table is updated after each action, based on the reward received and the maximum Q-value of the next state. The update rule is as follows:

$Q(s,a) = Q(s,a) + \alpha [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

where $Q(s,a)$ is the Q-value of taking action a in state s, α is the learning rate, r is the reward received for taking action a in state s, γ is the discount factor (a parameter that determines the importance of future rewards relative to immediate rewards), $\max_{a'} Q(s',a')$ is the maximum Q-value of the next state s', and a' is the set of all possible actions in the next state.

The Q-learning algorithm continues to interact with the environment and update the Q-table until it converges. At convergence, the Q-table represents the optimal action-selection policy for the given MDP.

However of course as you can tell, if the number of states the agent interacts is infinite then there is no way to populate a finite table with all the possible states and actions and so a smarter approach has to be taken to address this. This is where implementing DQN

comes into fruition where we take advantage of Neural networks to predict what the value of an action is at a given state.

Deep-Q Learning

Deep Q Learning is an extension of the Q-learning algorithm that uses a deep neural network as a function approximator for the Q-value function. This allows it to handle high-dimensional state spaces, making it suitable for more complex environments such as Atari.

In Deep Q Learning, the Q-value function is represented as a neural network that takes the current state as input and outputs the Q-value for each possible action. The Deep Q Learning algorithm starts by initializing a neural network with random weights, then repeatedly interacts with the environment, selecting actions based on the current state and the Q-values predicted by the neural network. The algorithm then receives a reward and the next state from the environment and stores the experience (state, action, reward, next state) in a replay memory buffer.

The neural network is trained using minibatch gradient descent on random samples of experiences from the replay memory buffer. During training, the neural network is optimized to minimize the difference between the predicted Q-values and the target Q-values, which are computed using the Bellman equation:

$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$

where Q(s,a) is the predicted Q-value for taking action a in state s, r is the reward received for taking action a in state s, γ is the discount factor, $\max_{a'} Q(s',a')$ is the maximum predicted Q-value for the next state s' and all possible actions a'.

Intrinsic Curiosity Module

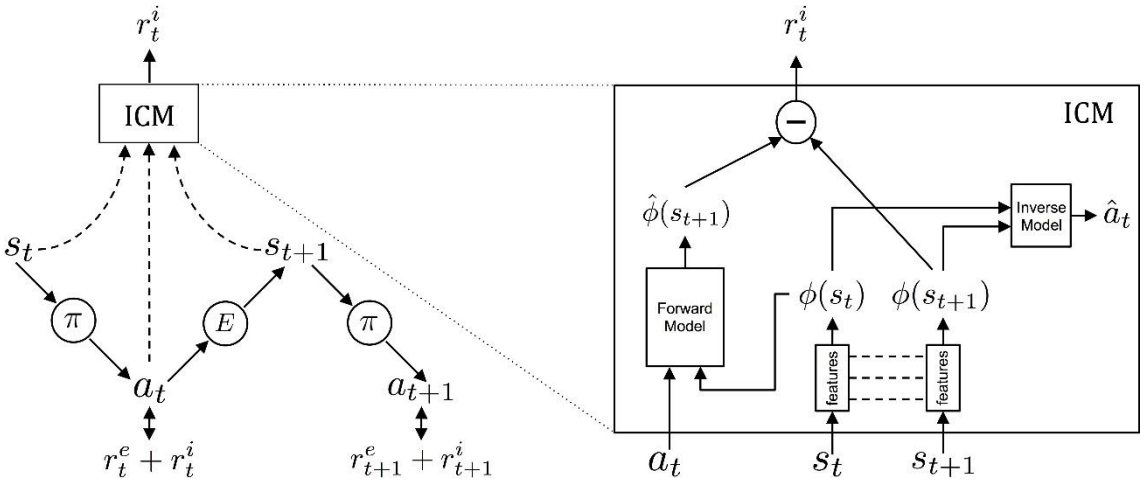
The idea behind ICM is that agents that are curious about their environment will explore more and learn faster, leading to better performance on a wide range of tasks. ICM works by introducing an additional reward signal in addition to the standard reward signal that an agent receives from the environment. The additional reward signal is based on the agent's ability to predict the consequences of its own actions. Specifically, the ICM consists of two neural networks: a forward model and an inverse model.

The forward model is trained to predict the next state that the agent will encounter given its current state and an action. The inverse model is trained to predict the action that the agent took to transition from one state to another. These predictions are compared to the actual outcomes of the agent's actions to compute a measure of prediction error.

The prediction error is then used as the additional reward signal for the agent. Agents that are able to predict the consequences of their actions more accurately will receive a higher extrinsic reward. It has been shown to be particularly effective in environments where there is a high degree of uncertainty or novelty, as it encourages agents to explore and learn about their environment in a more efficient and effective manner.

$\text{Reward\_ICM} = \eta * ||F(s, a) - s'||^2$

where F(s, a) is the forward model's prediction of the next state given the current state s and action a, s' is the actual next state, and η is a scaling factor.



Random Network Distillation

Random Network Distillation (RND) is another curiosity-driven exploration technique. It works by training two neural networks, a fixed random target network, and a separate predictor network. The target network has randomly initialized weights, while the predictor network learns to mimic the output of the target network. The idea is that the agent will explore the environment to reduce the prediction error between the target and predictor networks.

The intrinsic reward in RND is based on the prediction error between the target and predictor networks and can be defined as:

Reward_RND = κ * ||T(s) - P(s)||²

where T(s) is the target network's output for the current state s, P(s) is the predictor network's output for the current state s, and κ is a scaling factor.

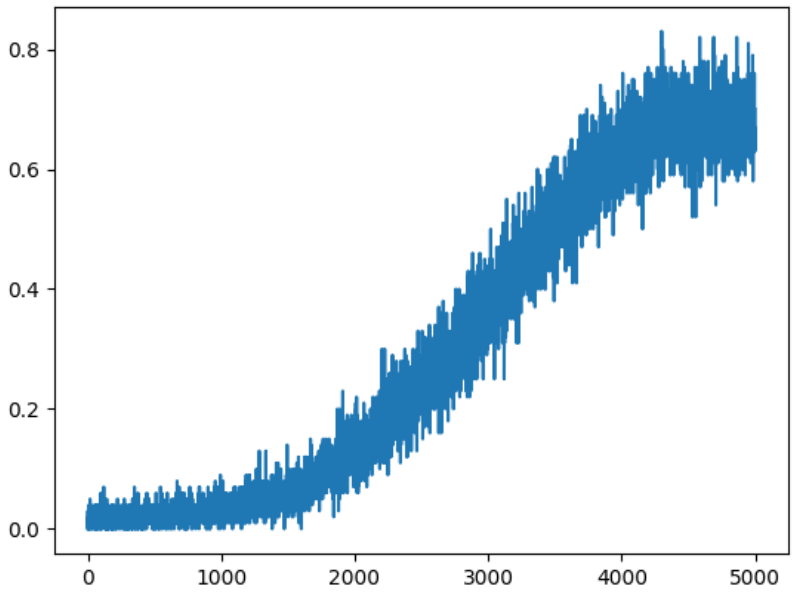
DreamerV2 explained in the literature review.

Results and Explanations

Experiment 1 – Q Learning applied to Frozen Lake Environment

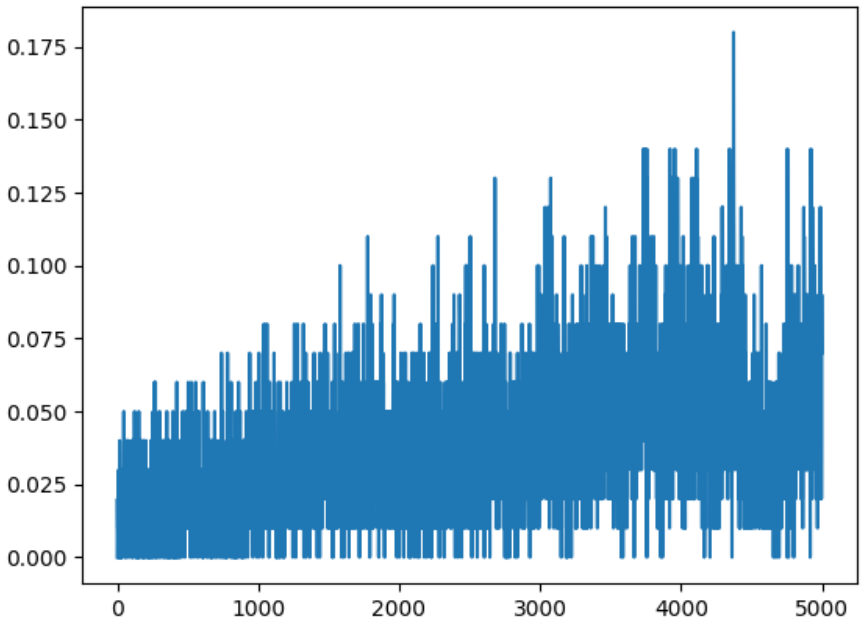
Gamma=0.9

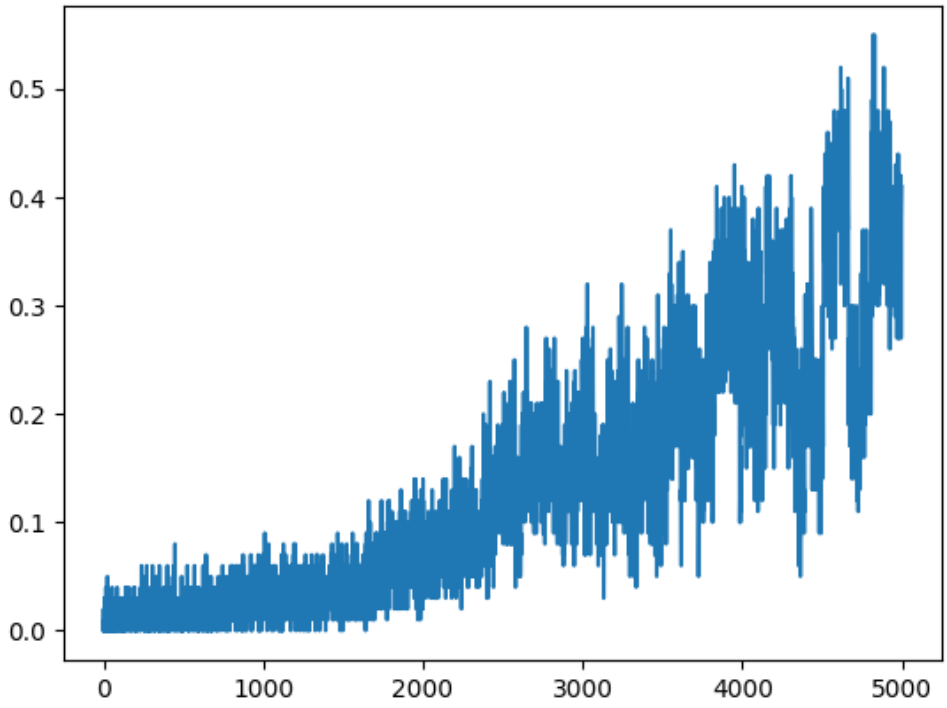
lr = 0.001



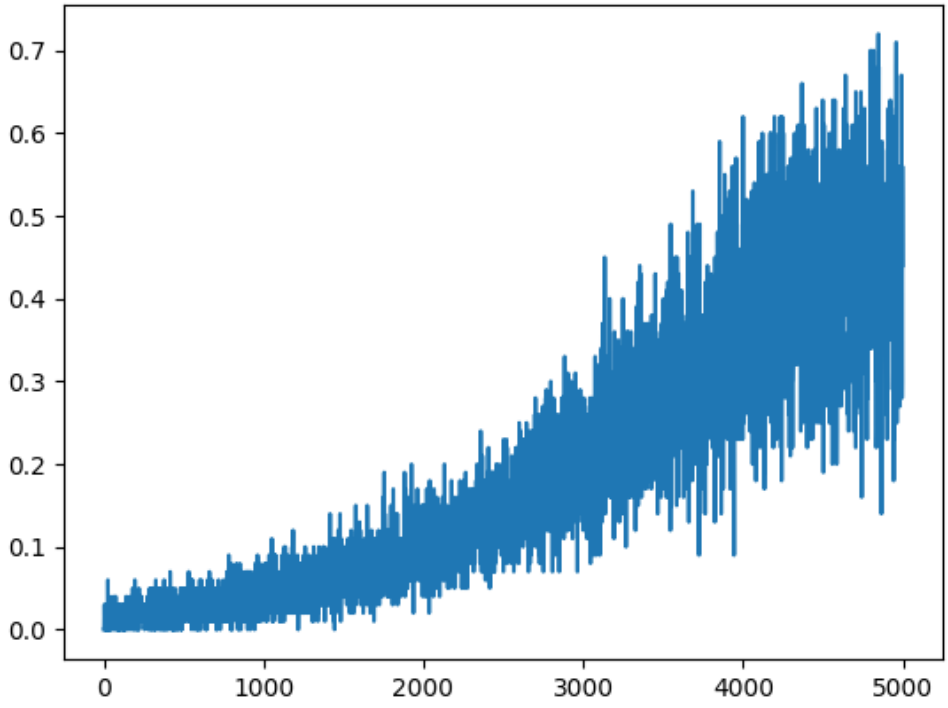
lr=0.001

Gamma = 0.1

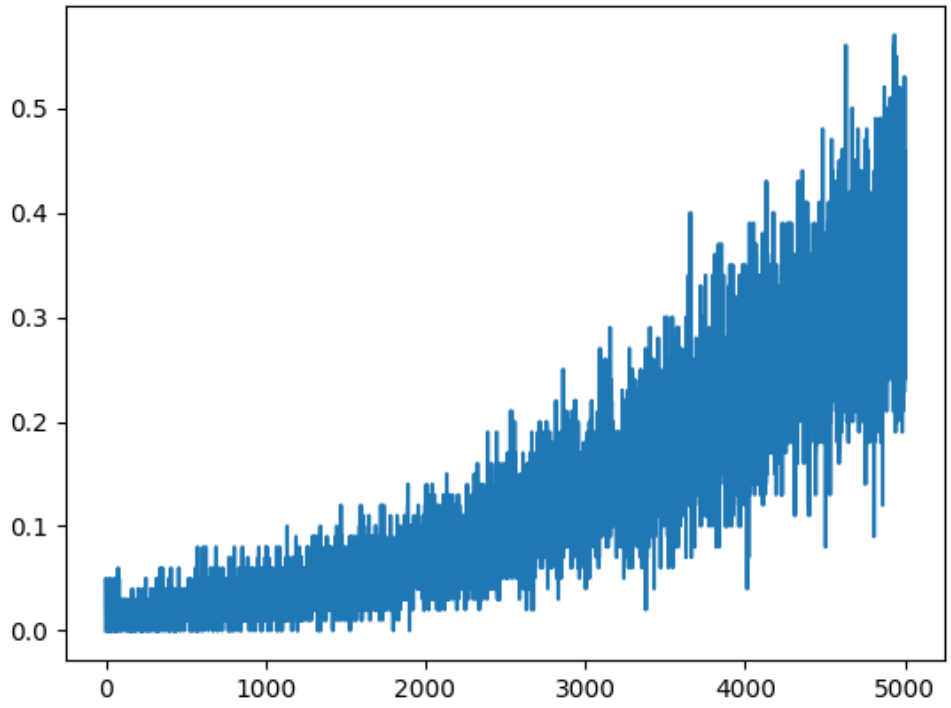




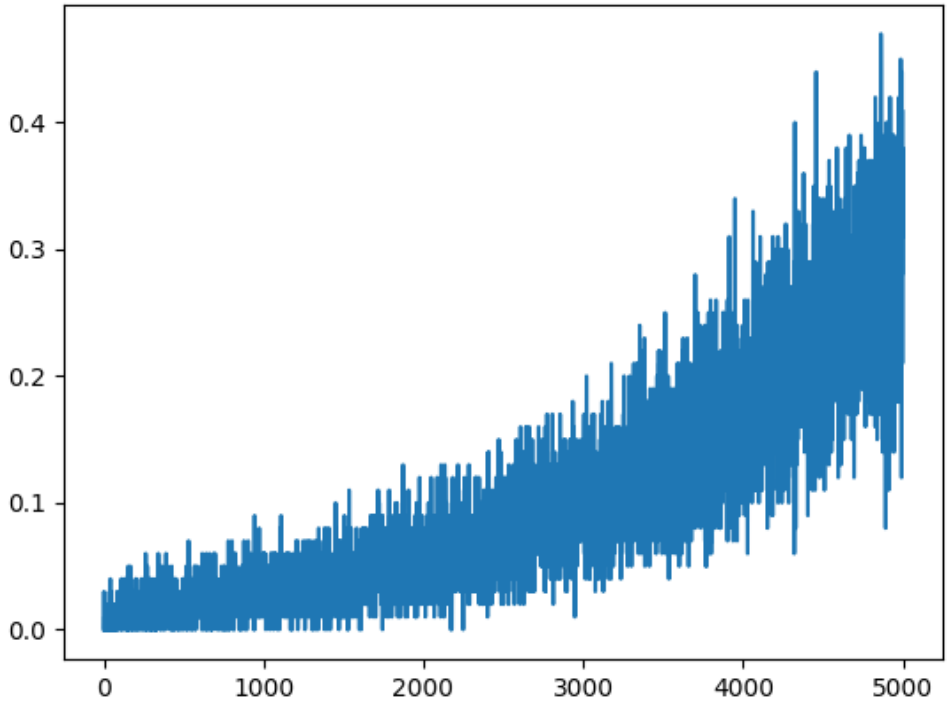
gamma = 0.5, lr = 0.001



gamma = 0.9, lr = 0.1



Gamma = 0.9, lr = 0.3



Gamma = 0.9, lr=0.5

1. Gamma (discount factor): This hyperparameter determines the extent to which future rewards are considered when updating Q-values. A high gamma (closer to 1) means that the agent will prioritize long-term rewards, while a low gamma (closer to 0) means that the agent will focus more on immediate rewards.

    - Gamma = 0.9: With a gamma value of 0.9, the agent is considering both immediate and future rewards when making decisions. This leads to more consistent performance, as the agent is more likely to discover optimal policies that consider the long-term consequences of its actions.

    - Gamma = 0.5: Decreasing the gamma value to 0.5 reduces the agent's focus on future rewards, which can lead to more variance in results and suboptimal performance, as the agent may not fully consider the long-term implications of its actions.

    - Gamma = 0.1: With a gamma value of 0.1, the agent is mostly focused on immediate rewards, neglecting the long-term consequences of its actions. This can result in poor performance, as the agent fails to discover optimal policies that balance short-term and long-term rewards.

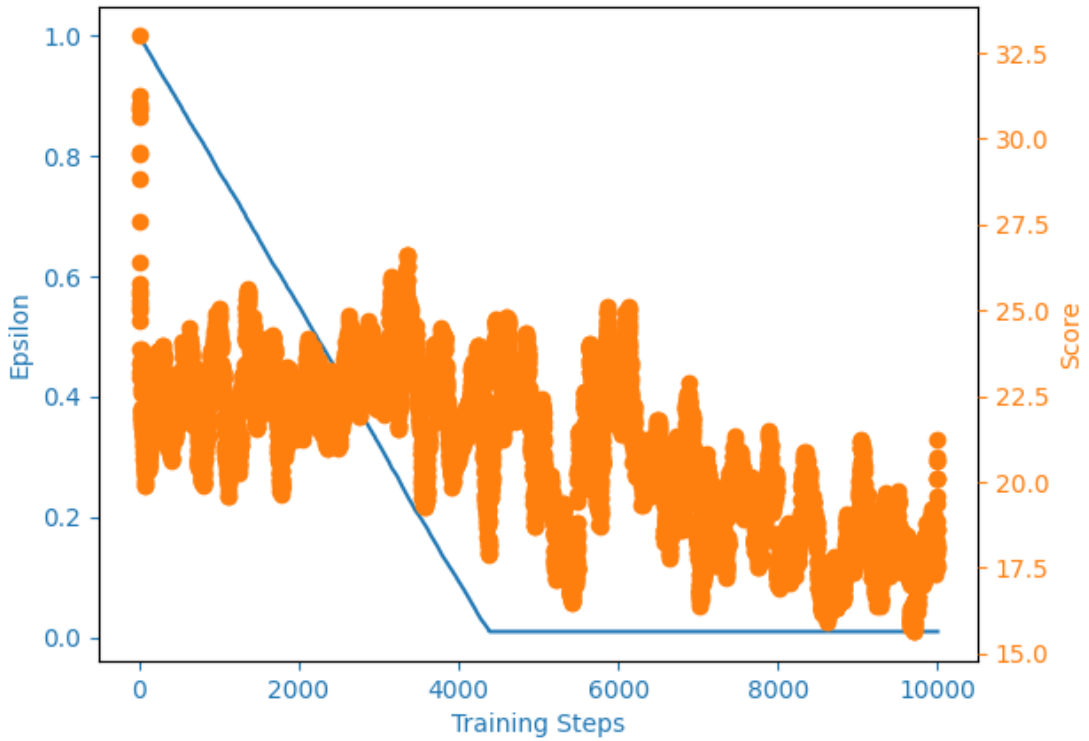
2. Learning rate (lr): This hyperparameter controls the rate at which Q-values are updated. A high learning rate (closer to 1) means that the agent will make large updates to its Q-values, while a low learning rate (closer to 0) means that the agent will make smaller updates.

    - lr = 0.001: With a low learning rate, the agent updates its Q-values slowly, which can lead to more stable and consistent learning. This is likely why the best performance was observed with gamma = 0.9 and lr = 0.001.

    - lr = 0.5: A higher learning rate can lead to faster initial learning but may cause the agent's performance to oscillate and be less consistent. This is likely why the performance was approximately half when compared to the best results when using lr = 0.5.

In conclusion, the best performance was achieved when using gamma = 0.9 and lr = 0.001, as this combination of hyperparameters allowed the agent to balance the consideration of long-term rewards and learn at a stable and consistent rate. Lower gamma values resulted in worse performance due to the agent's focus on immediate rewards, while a higher learning rate led to less consistent learning.

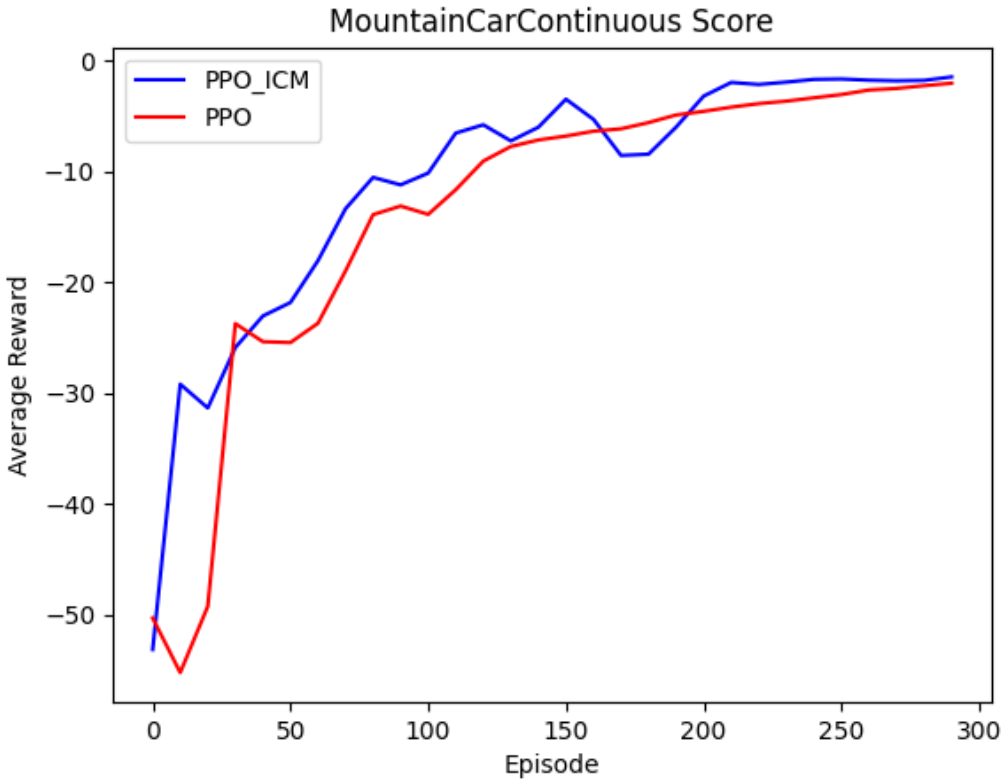
Experiment 2 – Naïve DQN applied to Cart Pole Environment

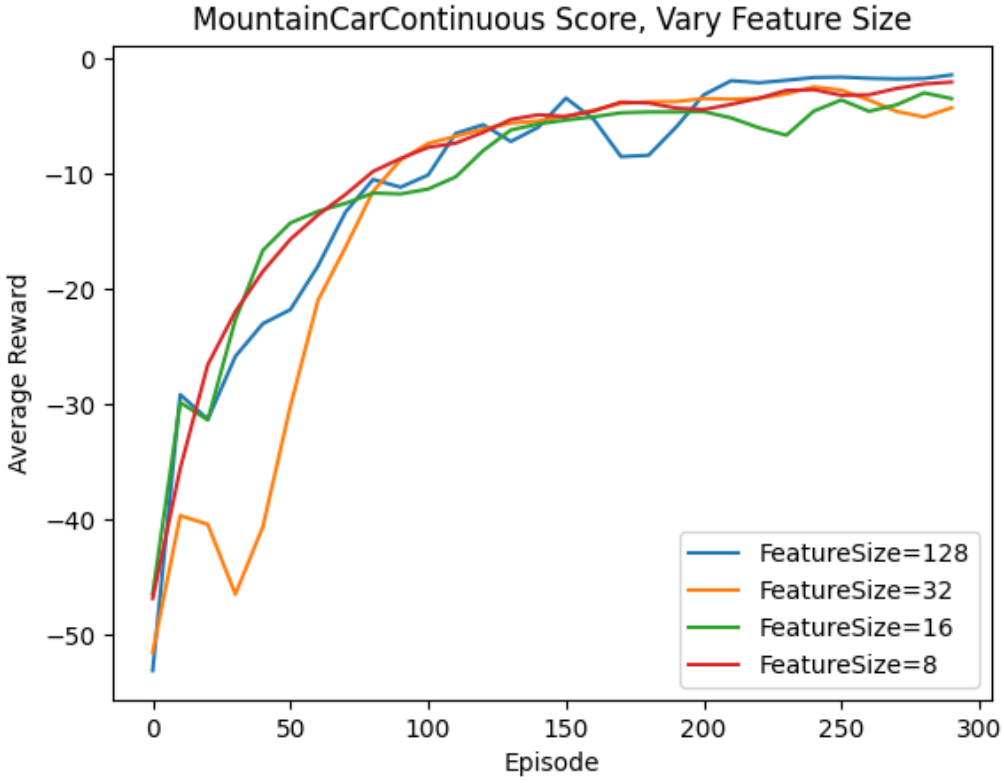


Lack of episodic memory: Without a replay buffer, the Naive DQN is learning from consecutive experiences, which can be highly correlated. This can lead to several issues:

- High correlation: When learning from consecutive experiences, the agent might overfit to recent experiences and forget valuable knowledge it has acquired earlier. This can cause the performance to fluctuate and even slightly decrease over time, as you observed.
- Unstable learning: Without the replay buffer, the Naive DQN is more likely to suffer from unstable learning due to the high correlation between consecutive experiences. This can result in the agent not fully converging to an optimal policy and exhibiting fluctuating performance.

Experiment 3 – PPO and ICM


MountainCarContinuous Score
PPO_ICM
PPO
Average Reward
Episode
0
−10
−20
−30
−40
−50
0
50
100
150
200
250
300


MountainCarContinuous Score, Vary Feature Size
FeatureSize=128
FeatureSize=32
FeatureSize=16
FeatureSize=8
Average Reward
Episode
0
−10
−20
−30
−40
−50
0
50
100
150
200
250
300

1. PPO + ICM vs. PPO: The performance improvement when using PPO_ICM compared to PPO alone can be explained by the added exploration driven by intrinsic motivation. PPO is a policy optimization algorithm that focuses on improving the policy through gradient-based updates, but it does not explicitly encourage exploration. ICM, on the other hand, provides an intrinsic reward signal based on the agent's ability to predict the consequences of its actions. By combining PPO and ICM, the agent not only optimizes its policy but also explores its environment more effectively, leading to better overall performance.

2. Effect of feature size in ICM model: The varying performance based on the feature size of the ICM model can be attributed to the capacity of the model to represent the environment's complexity. When the feature size is too small (e.g., 8), the ICM model may struggle to capture all relevant information, resulting in a suboptimal intrinsic reward signal. However, in some cases, a smaller feature size may act as a form of regularization, preventing the model from overfitting and leading to better generalization. This could explain why the performance with a feature size of 8 was relatively good.

On the other hand, when the feature size is too large (e.g., 128), the model may become overly complex, leading to slower learning and a higher chance of overfitting. However, a larger feature size can also provide the model with more capacity to represent the environment, which could explain the slightly better performance compared to the feature size of 8.

Surprisingly, a feature size of 32 led to the worst performance in your experiments. This could be due to a combination of factors, such as the balance between model capacity and regularization or the particular initialization of the model weights. It is worth noting that the choice of hyperparameters, including feature size, can be sensitive to the specific environment and problem, so the results may not generalize to other settings. Mountain Cart is a very simple environment which may explain why higher feature sizes did not yield better performance.

Experiment 4 – Dreamer V2 Exploration Methods and other hyper parameters – Main Experiments from here:

These are the initial parameters:

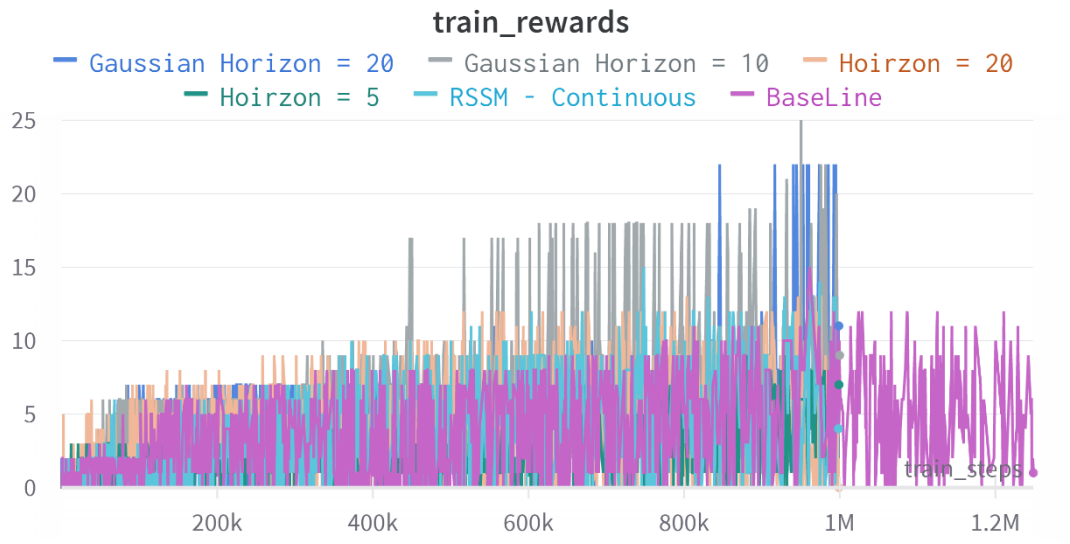
critic	
activation	"torch.nn.modules.activation.ELU"
dist	"normal"
layers	3
node_size	100
discount	
activation	"torch.nn.modules.activation.ELU"
dist	"binary"
layers	3
node_size	100
use	true
discount_	0.99
embedding_size	200
env	"breakout"
eval_episode	4
eval_render	true

expl	
eval_noise	0
expl_decay	7000
expl_min	0.05
expl_type	"epsilon_greedy"
train_noise	0.4
gif_dir	"results"
grad_clip	100
horizon	10
kl	
free_nats	0
kl_balance_scale	0.8
use_free_nats	false
use_kl_balance	true
lambda_	0.95
loss_scale	
discount	5

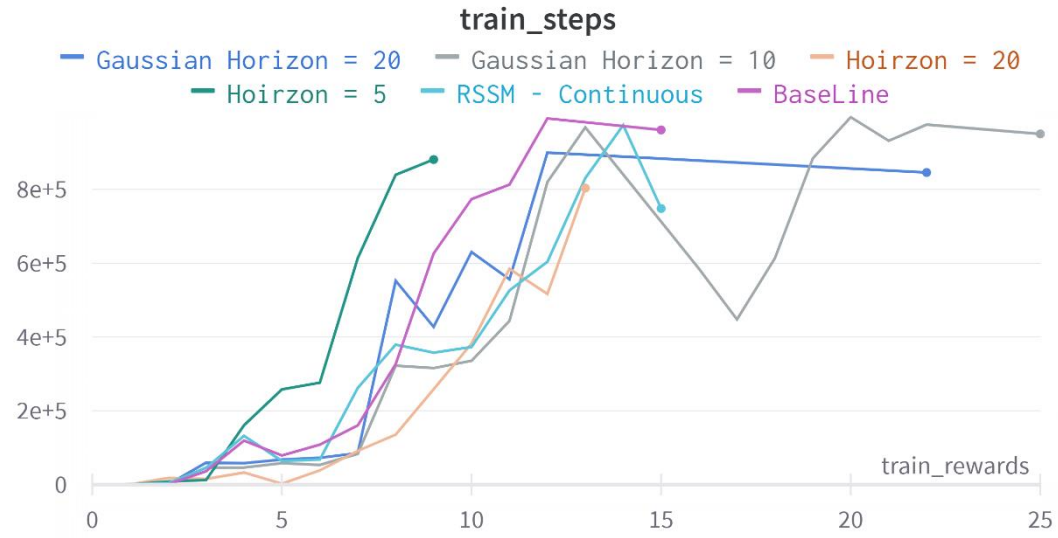
kl	0.1
reward	1
lr	
actor	0.00004
critic	0.0001
model	0.0002
model_dir	"results\\breakout_0_pomdp\\models"
obs_decoder	
activation	"torch.nn.modules.activation.ELU"
depth	16
dist	"normal"
kernel	3
layers	3
node_size	100
obs_dtype	"builtins.bool"
obs_encoder	
activation	"torch.nn.modules.activation.ELU"
depth	16
dist	null
kernel	3
layers	3
node_size	100
obs_shape (3 collapsed)	
pixel	true
reward	
activation	"torch.nn.modules.activation.ELU"
dist	"normal"
layers	3
node_size	100
rssm_info	
category_size	20
class_size	20
deter_size	200
min_std	0.1

stoch_size	20
rssm_node_size	200
rssm_type	"discrete"
save_every	100000
seed_steps	4000
seq_len	50
slow_target_fraction	1
slow_target_update	100
train_every	50
train_steps	5000000
use_slow_target	true

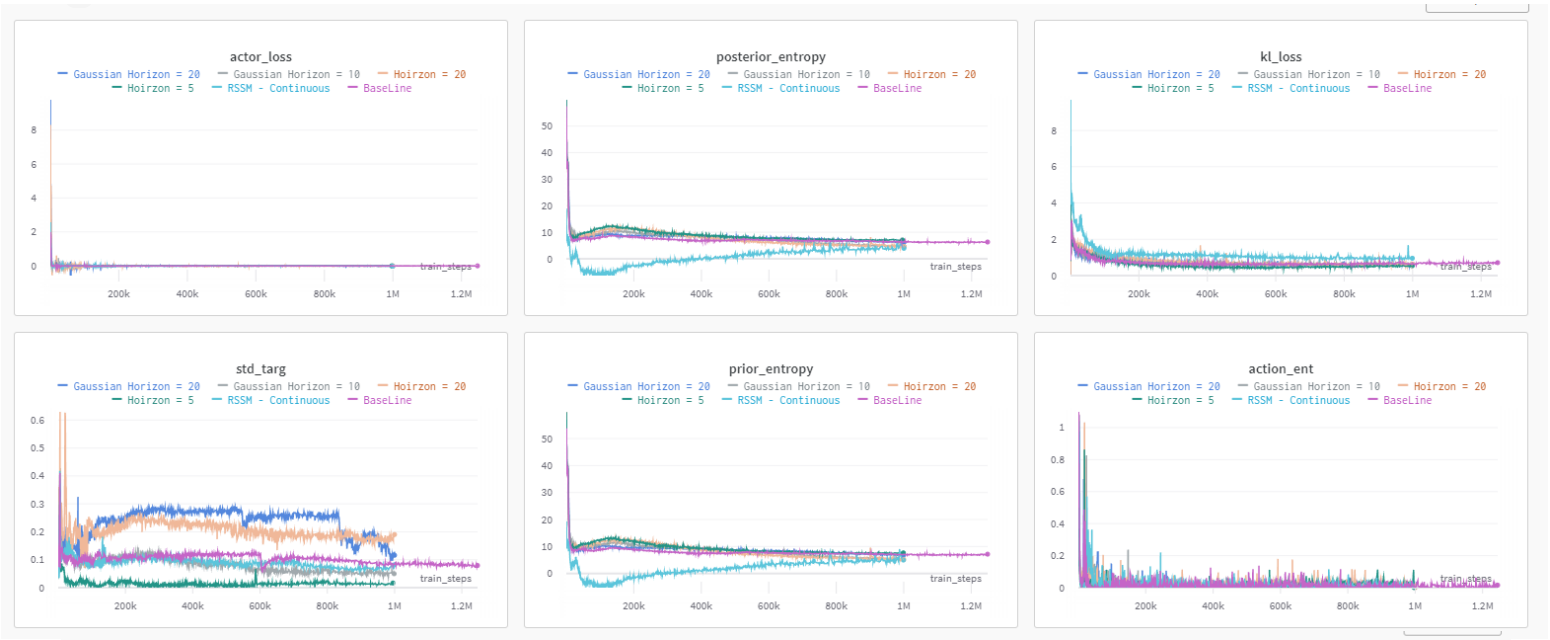
Base Line result is set using these parameters (Note train steps is actually set to 1 million). I will not include all the graphs in this section as it would take too much space but put the most important ones such as the rewards received.

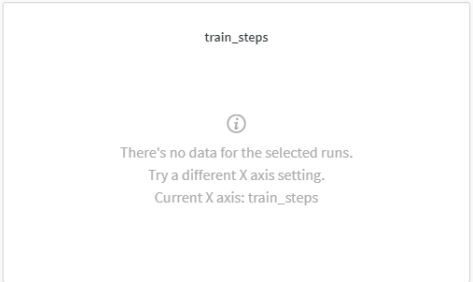


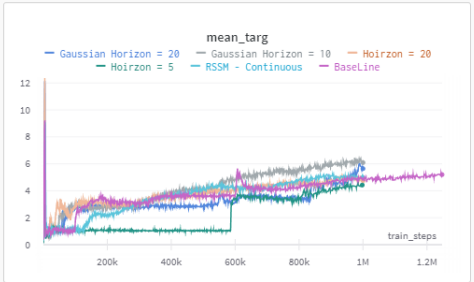
Given that train rewards over time clearly fluctuates, its not exactly quite clear which is why I also provide a graph where the y axis is train steps and x-axis is train rewards which is not ideal since its typically required to put the independent variable on the x-axis.

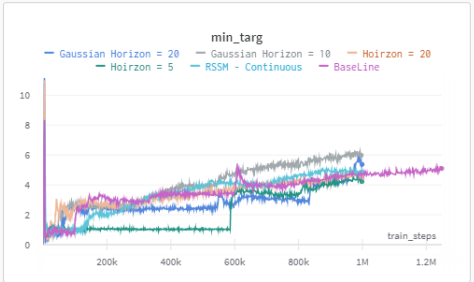


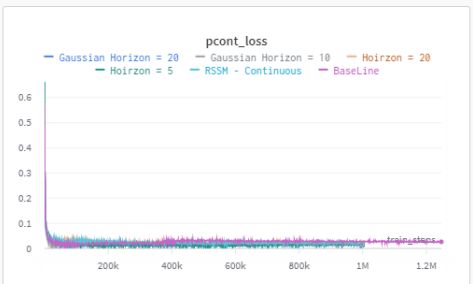
Other metrics:

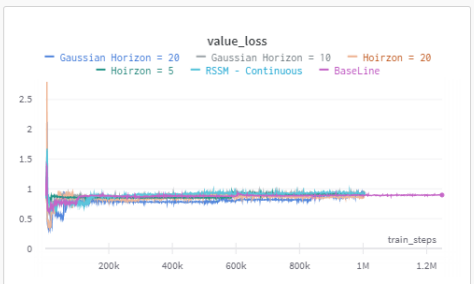


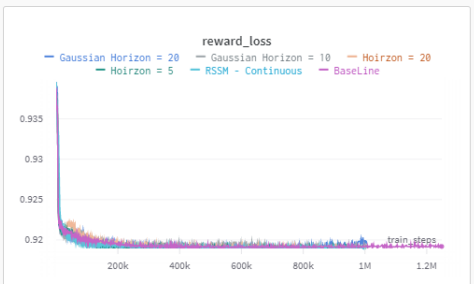












Explanation of these results:

In this experiment, the baseline performance was established using standard epsilon-greedy exploration and a horizon of 10. The horizon represents how many timesteps the agent can look ahead, and it was set to 10 as a default. The goal was to investigate the effect of increasing and decreasing the horizon on the agent's performance, as well as to explore the impact of using a Gaussian exploration strategy instead of epsilon greedy.

Decreasing the horizon to 5 timesteps significantly reduced performance, likely due to the agent's diminished ability to predict the consequences of its actions. On the other hand, increasing the horizon to 20 timesteps resulted in a slight decrease in performance, although not significant enough to draw a firm conclusion. These results were consistent regardless of whether the epsilon-greedy or Gaussian exploration strategy was used.

Using a Gaussian exploration strategy, rather than epsilon-greedy, led to a substantial improvement in performance compared to the baseline. The Gaussian strategy's robustness was further demonstrated by its good performance when the horizon was set to 20. There could be several reasons for the Gaussian strategy's superior performance, such as its ability to better balance exploration and exploitation.

Another factor investigated in the experiment was the impact of using a continuous state representation (RSSM) instead of the default discrete one. The results showed a decrease in performance when using a continuous RSSM, but this is likely due to the specific environment used (Breakout), which does not have continuous state-action spaces. It would be interesting to explore the effects of using continuous state representation in environments with continuous state-action spaces, but due to time and computational constraints, this was not possible for this project.

Additionally, it was observed that higher horizons led to longer runtime, although this should not be confused with sample efficiency, as all runs used the same number of training steps (except for the baseline, which unintentionally used 200k more steps).

In summary, this experiment provided valuable insights into the effects of varying the horizon and exploration strategy on the agent's performance. Decreasing the horizon hindered performance, while using a Gaussian exploration strategy yielded significant improvements over the baseline. Further research could investigate the impact of continuous state representations in environments with continuous state-action spaces and explore additional ways to optimize agent performance.

The rest of figures showed metrics such as:

1. **model_loss**: The loss for the world model, which is used to predict the next state, given the current state and action.

2. **kl_loss**: The KL divergence loss, which measures the difference between the prior and posterior distributions. This is to ensure that the learned model can generalize well.

3. **reward_loss**: The loss in predicting rewards by the world model.

4. **obs_loss**: The loss in predicting the next observation (state) by the world model.

5. **value_loss**: The loss in estimating state values by the critic network.

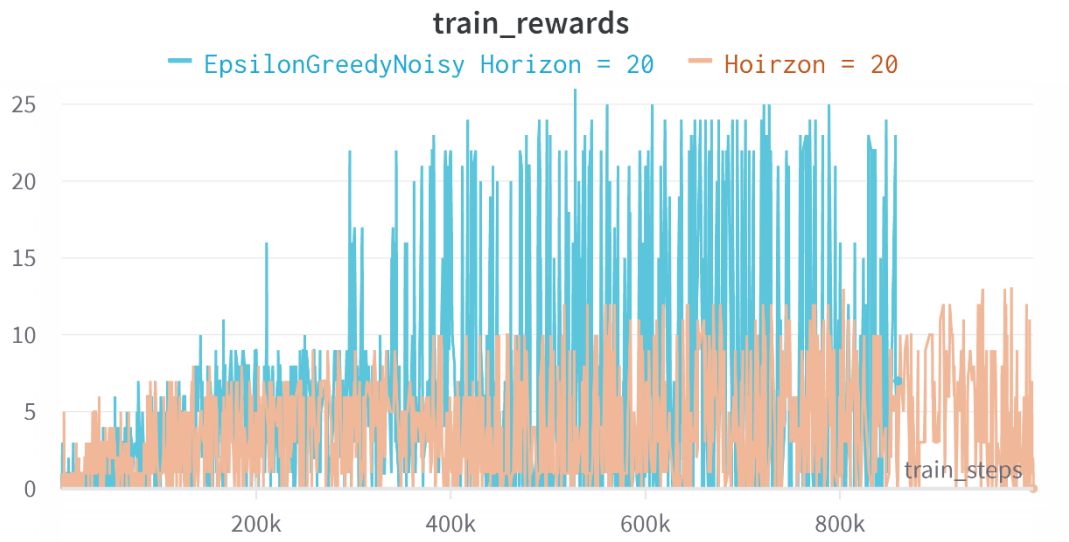
6. **actor_loss**: The loss in policy optimization for the actor network.

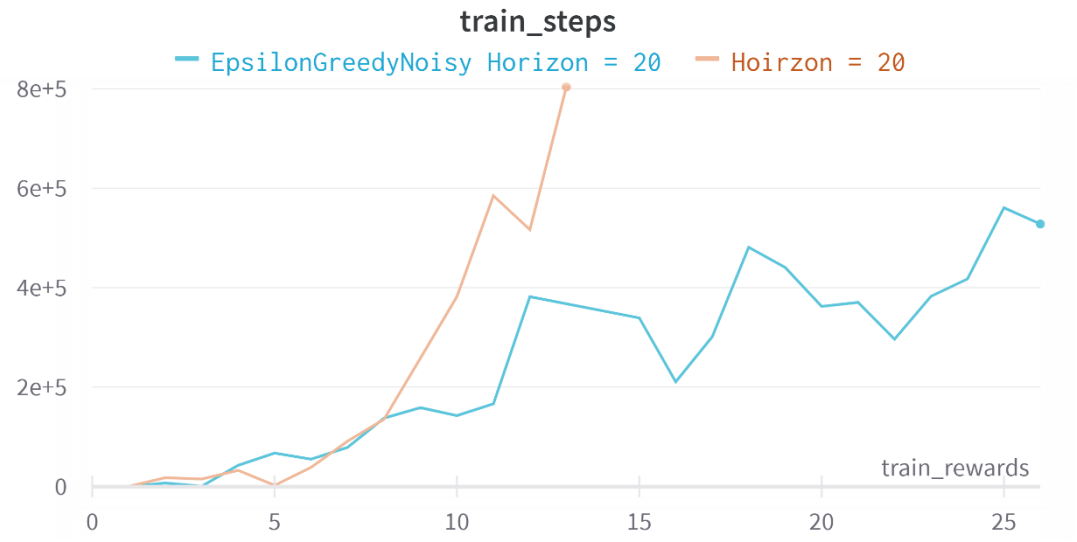
7. **prior_entropy**: The entropy of the prior distribution, which represents the model's uncertainty about the next state before observing the actual outcome.

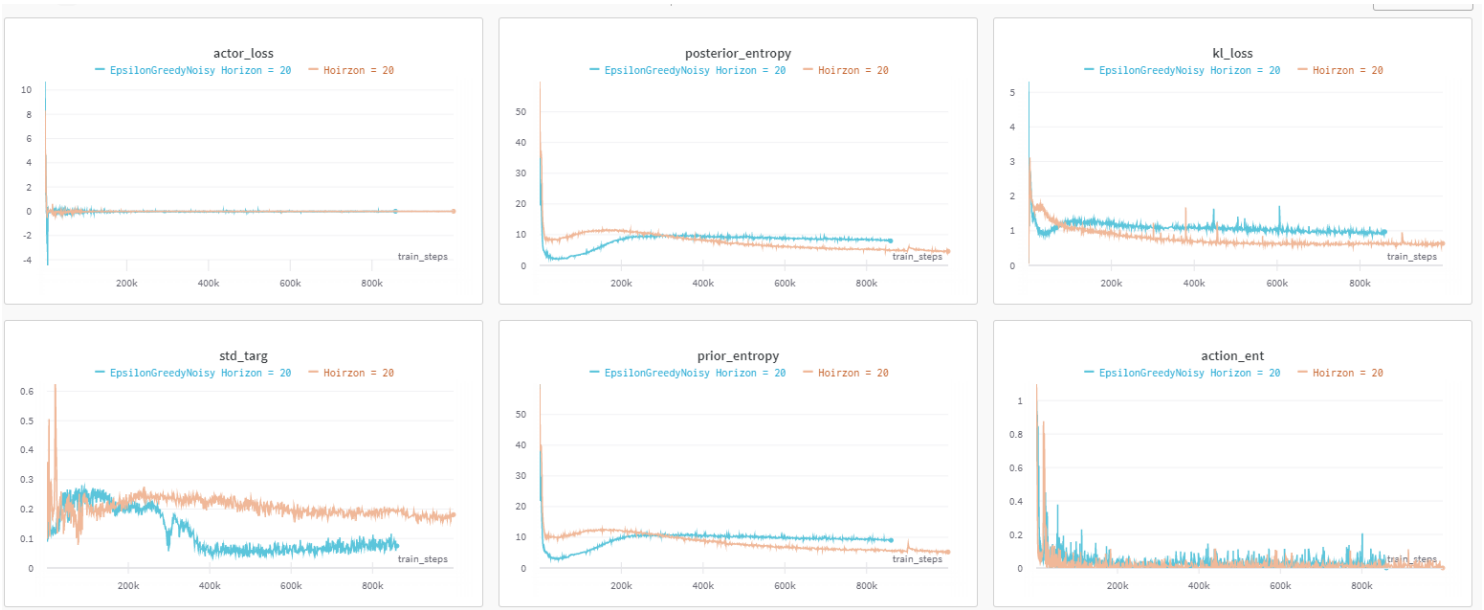
8. **posterior_entropy**: The entropy of the posterior distribution, which represents the model's uncertainty about the next state after observing the actual outcome.

9. **mean_targ**, **min_targ**, **max_targ**, **std_targ**: These metrics are related to the target values used during training. They provide information about the distribution of target values the agent is trying to predict. Specifically, **mean_targ** is the average target value, **min_targ** is the minimum target value, **max_targ** is the maximum target value, and **std_targ** is the standard deviation of the target values. These metrics give insights into how the agent's estimates are evolving during training but are not directly correlated with the number of rewards the agent receives. They can, however, indirectly reflect the agent's learning progress and performance.

Epsilon Greedy with Noisy Neural Networks: <- Biggest Performance Increase




train_steps
EpsilonGreedyNoisy Horizon = 20
Hoirzon = 20
8e+5
6e+5
4e+5
2e+5
0
0
5
10
15
20
25
train_rewards


actor_loss
posterior_entropy
kl_loss
std_targ
prior_entropy
action_ent
EpsilonGreedyNoisy Horizon = 20
Hoirzon = 20
train_steps


train_steps
There's no data for the selected runs.
Try a different X axis setting.
Current X axis: train_steps
mean_targ
min_targ
pcont_loss
value_loss
reward_loss
EpsilonGreedyNoisy Horizon = 20
Hoirzon = 20
train_steps

Explanation of this result:

In the Breakout environment, combining the Epsilon-greedy strategy with Noisy Networks likely led to a significant increase in performance due to the complementary nature of these two exploration techniques.

The Epsilon-greedy strategy is a simple yet effective exploration method that encourages the agent to take random actions with a probability of epsilon. This helps the agent explore the environment and avoid getting stuck in a local optimum. However, the exploration is undirected, meaning that the agent does not prioritize its exploration based on its current knowledge or uncertainty about the environment.

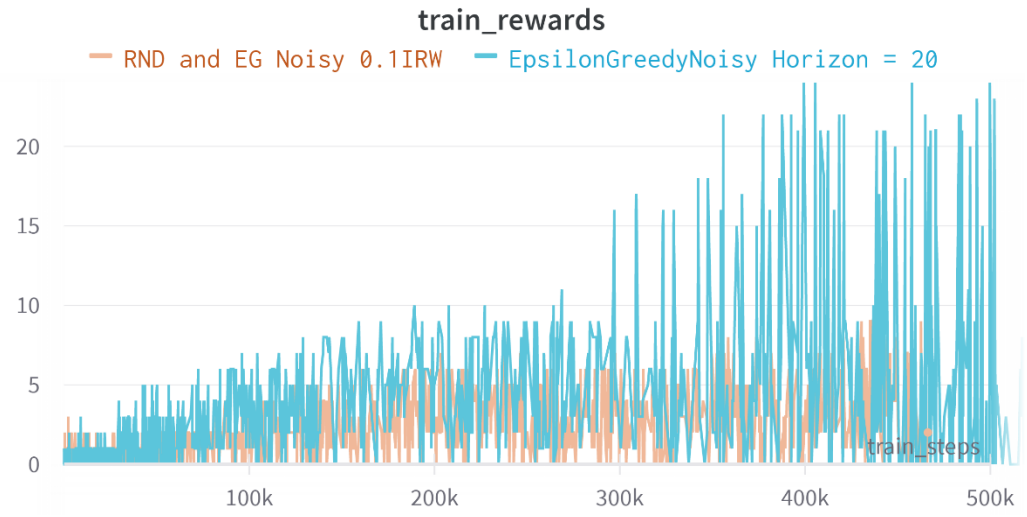
On the other hand, Noisy Networks introduce parameter noise directly into the weights and biases of the neural network layers. This causes the agent to take different actions even when it is in the same state, leading to exploration. Noisy Networks provide a more directed exploration, as the noise is proportional to the agent's uncertainty about its action-value estimates. In other words, the agent explores more in areas where it is uncertain about the optimal action.

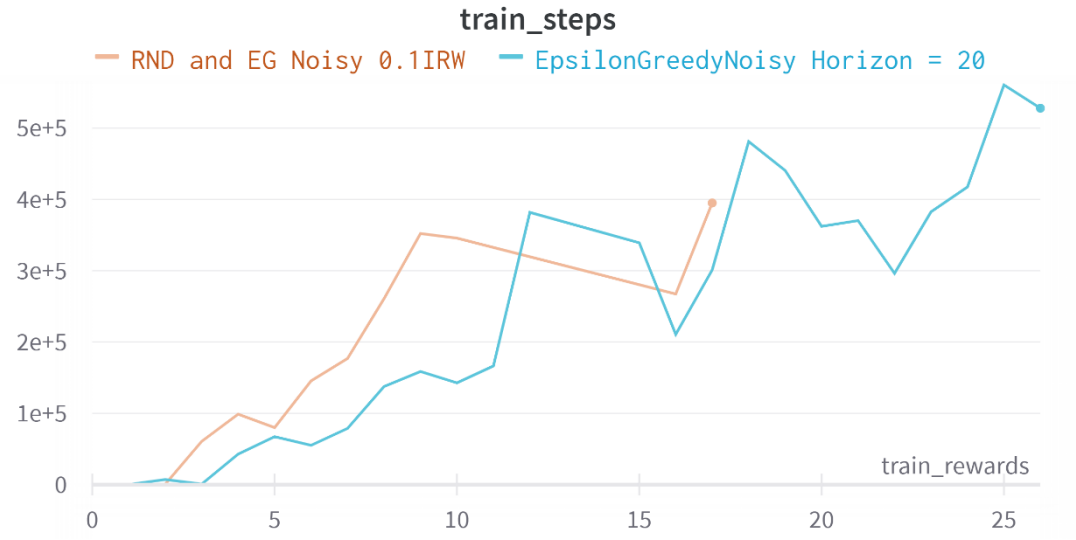
By combining Epsilon-greedy and Noisy Networks, you are leveraging the strengths of both exploration techniques. The Epsilon-greedy strategy ensures that the agent explores the environment in a broad sense, while Noisy Networks encourage more directed exploration based on the agent's uncertainty. This combination likely results in more efficient and effective exploration, enabling the agent to discover better policies and achieve higher performance in the Breakout environment.

Also note that for some reason the experiment was cut short and again due to time constraints I could not re do it just for an extra 100k steps.

Epsilon Greedy with Noisy Networks with Intrinsic reward by RND:

I capped this test to 500k steps.





Experiment results explained:

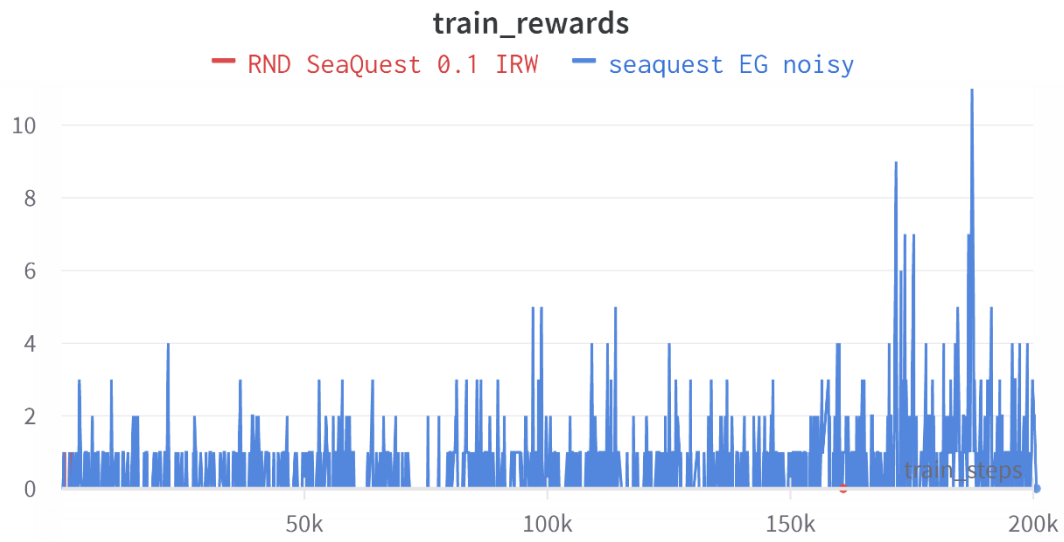
In the experiment conducted, the DreamerV2 agent was applied to the Breakout environment, with a combination of Epsilon-greedy exploration, Noisy Networks, and RND intrinsic rewards. Surprisingly, the results showed a decrease in performance when these elements were combined. Several factors may have contributed to this outcome.

Over-exploration: The integration of multiple exploration strategies, namely Epsilon-greedy and Noisy Networks, alongside the RND intrinsic reward mechanism, might have prompted the agent to over-explore its environment. Consequently, the agent could have spent an excessive amount of time investigating less relevant states and actions, thereby hindering the learning process, and resulting in suboptimal performance.

Intrinsic reward scaling: To effectively integrate RND intrinsic rewards with extrinsic rewards, proper scaling is necessary to balance the contributions of both reward sources. If the scaling is not well-tuned, the agent might prioritize intrinsic rewards over extrinsic rewards, which would result in suboptimal policies that do not maximize the true task rewards. Though I do not believe the issue to be here since I did small tests with other scaling factors, but it did not show promising results though due to time constraints I could not let the tests carry on longer. For this experiment a scaling of 0.1 was used.

Finally, I believe the main reason for this is actually due to the environment being used here as Breakout does not have sparse rewards and so this is not appropriate for curiosity driven algorithms to be deployed.

Final DreamerV2 Experiment – Trying to apply RND to a sparser reward env Sea Quest.

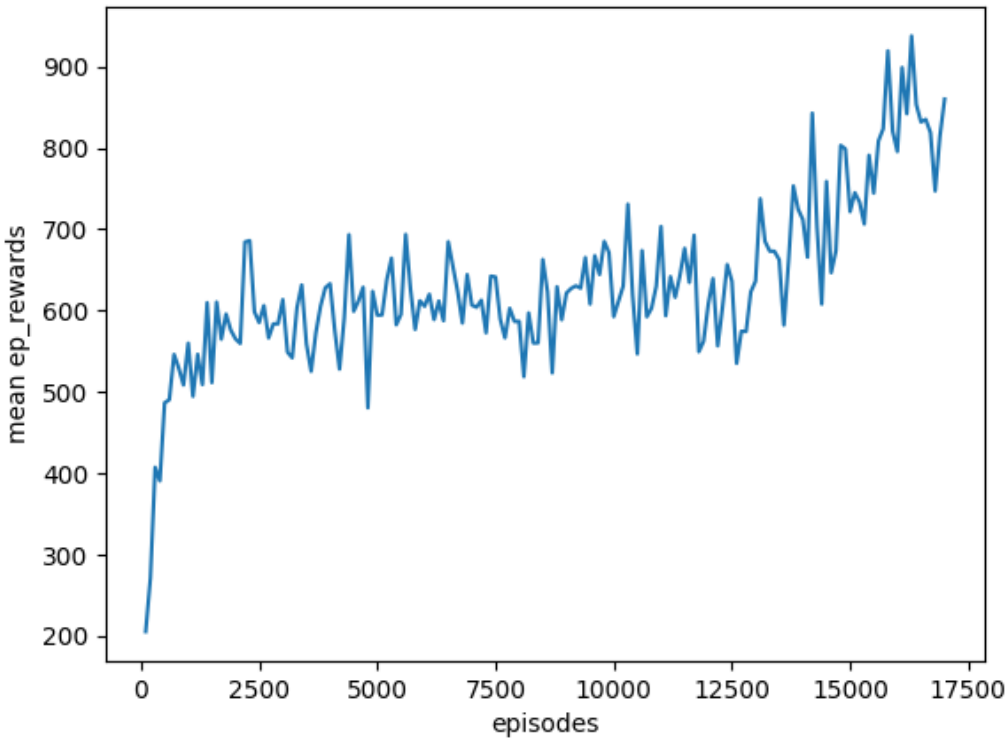


Results Explained.

This was by far the worst results of all experiments. The RND with EG Noisy as was used with Breakout completely failed at getting any rewards other then very little at the start. This was very strange as I assumed because Sea Quest contained more sparse rewards than Breakout it would be better. Though even after multiple re attempts the result was the same. Lowering the weight of the intrinsic reward slightly increased rewards at the start but the same outcome of 0 rewards afterwards persisted.

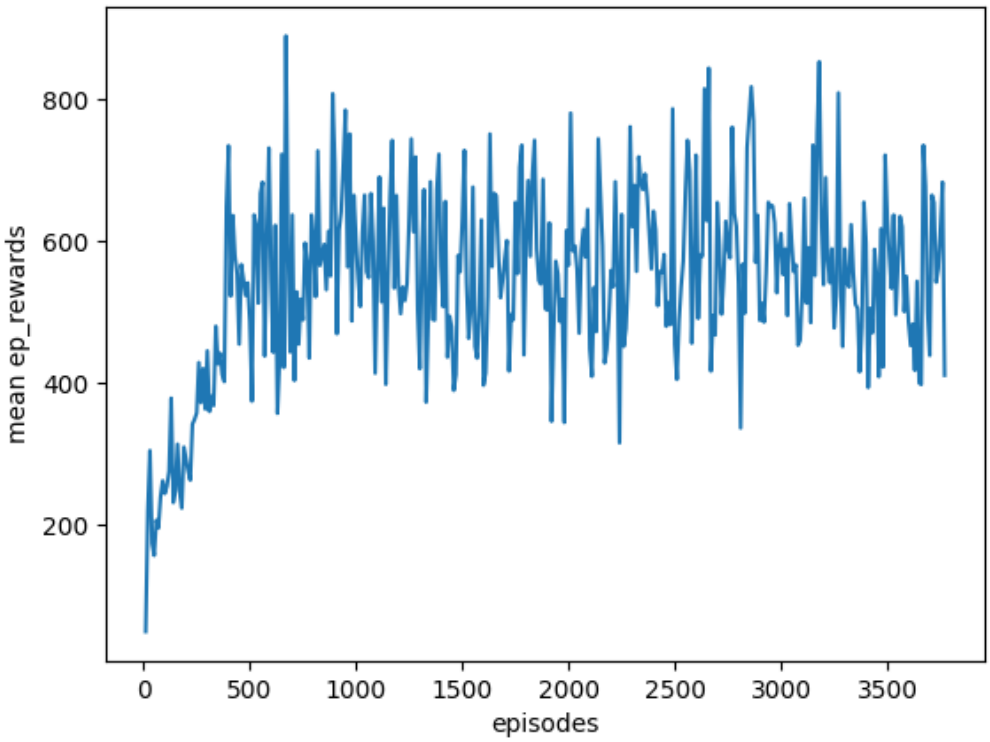
Experiment 5 – DQN and RND with Mario

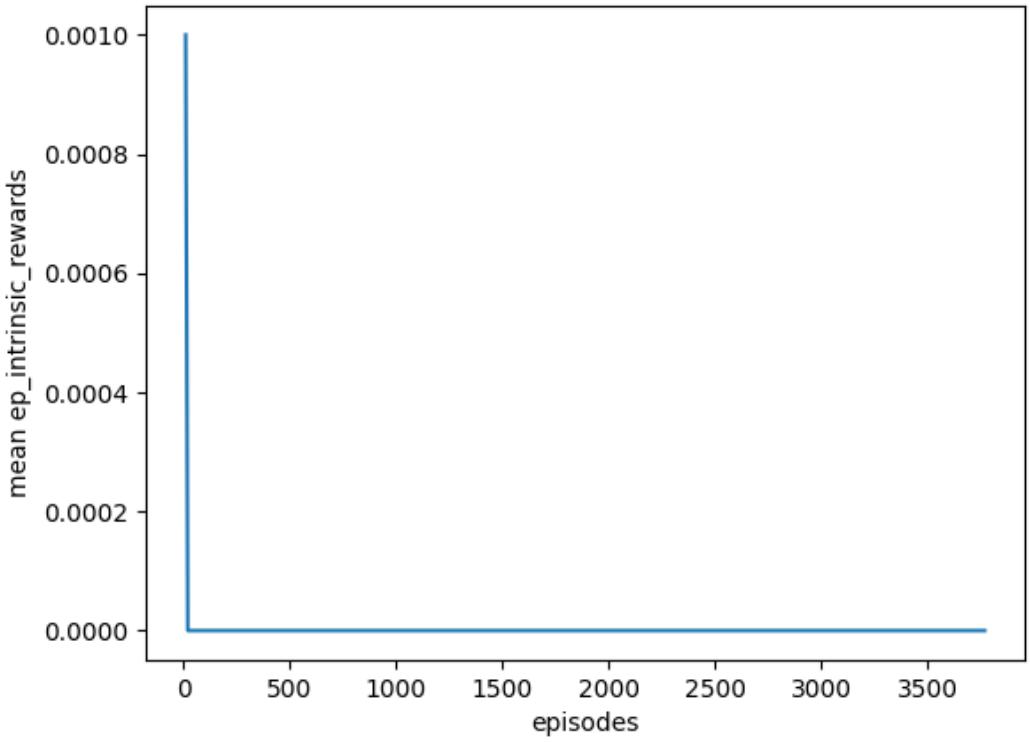
I wanted to see the effect of DQN and RND on super Mario as this is a known benchmark to use in RL especially with curiosity since there is an element of sparse rewards in Mario however this experiment was limited because of its high computing power required and so more time is required to find more interesting results.

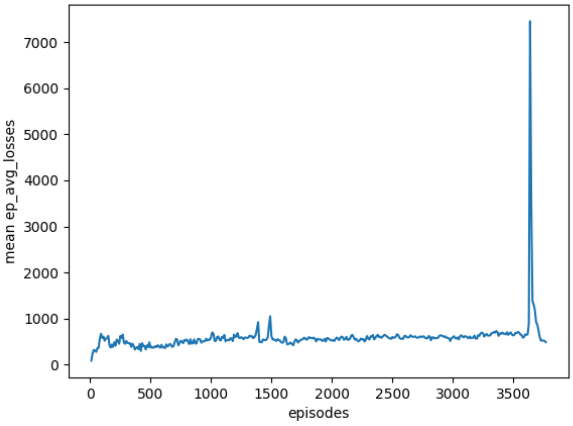


DQN on its own. This was run for over 12 hours for as many episodes however the RND versions didn't run for as long. I also ran the DQN for a shorter period of time though unfortunately it seems as though in the time of writing this that I lost the graphs for it.

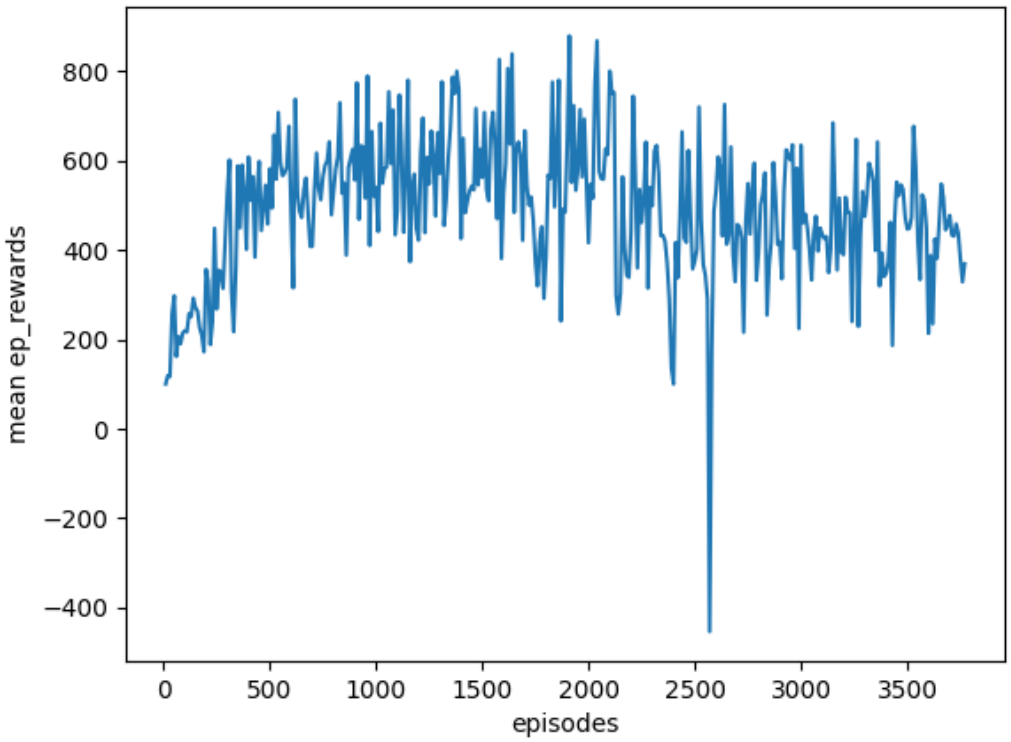
DQN with RND (MSE Loss) – I experimented also with different ways to calculate the prediction error between the 2 networks as some resulted in extremely miniscule intrinsic rewards.

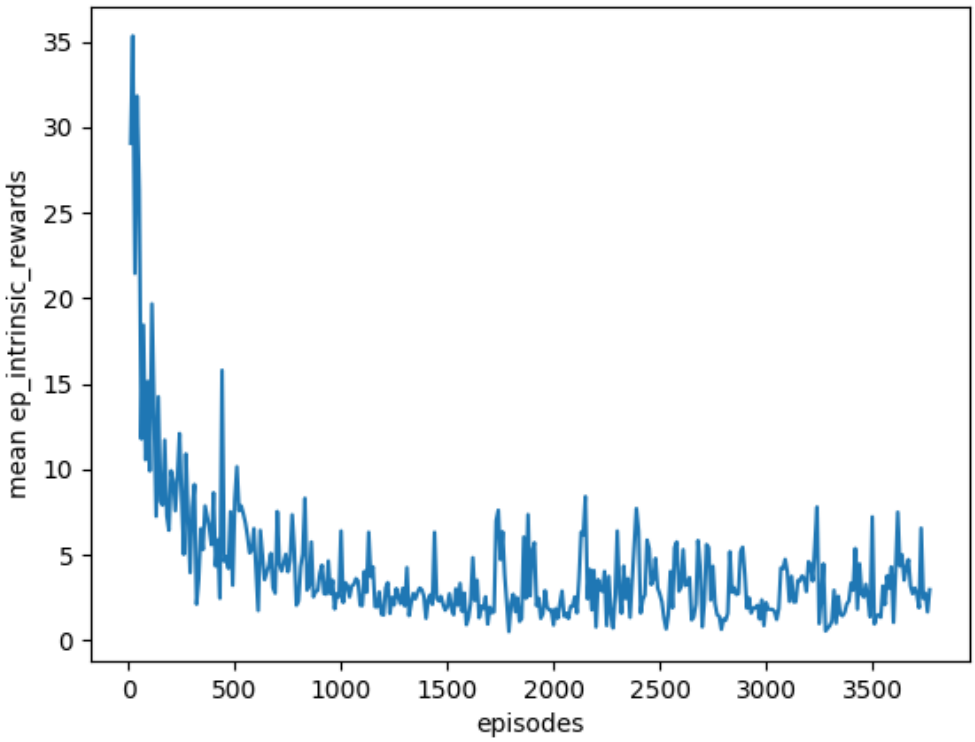

mean ep_rewards
800
600
400
200
0
500
1000
1500
2000
2500
3000
3500
episodes

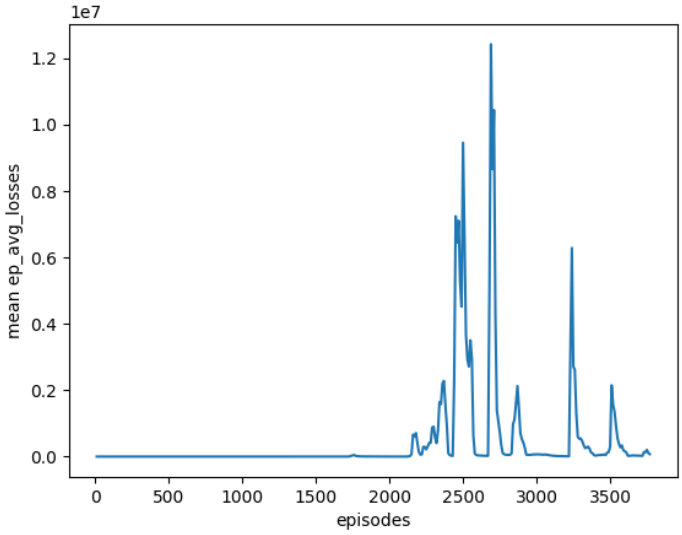

mean ep_intrinsic_rewards
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
0
500
1000
1500
2000
2500
3000
3500
episodes



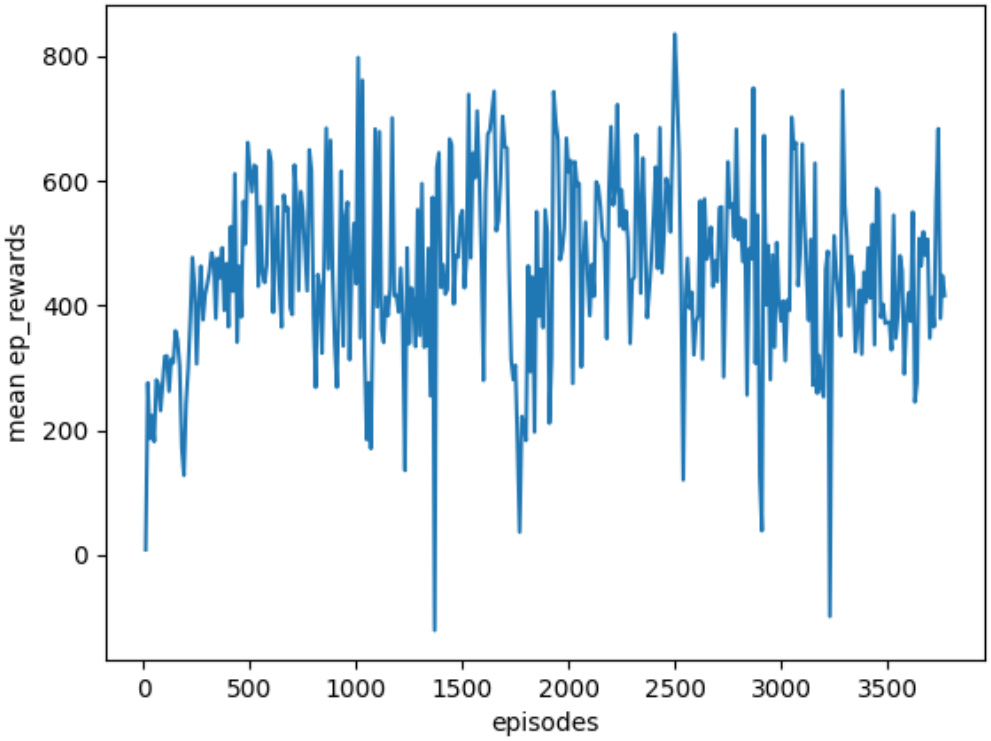
DQN with RND (Absolute difference Loss):

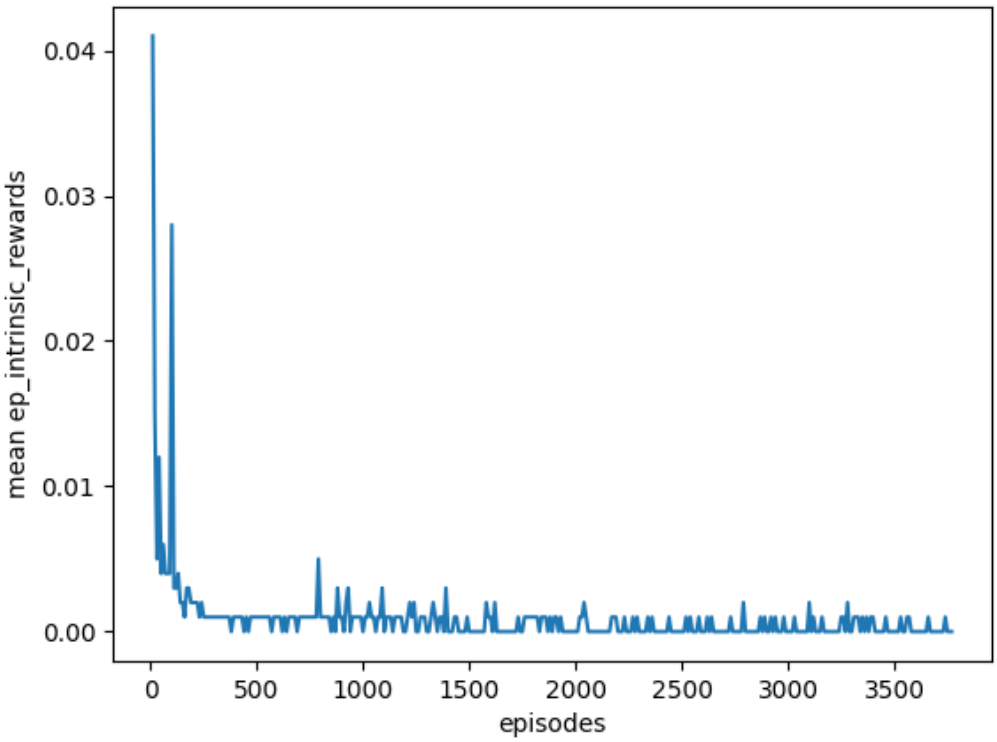


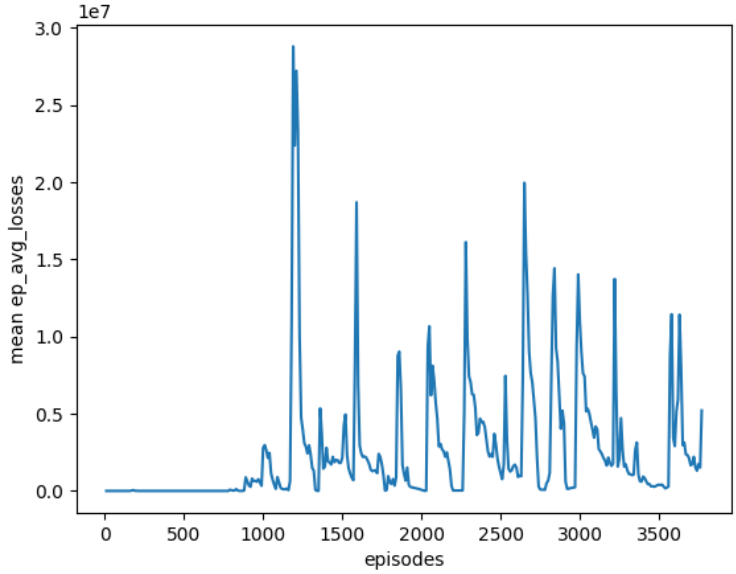

mean ep_intrinsic_rewards
episodes
0
5
10
15
20
25
30
35
0
500
1000
1500
2000
2500
3000
3500


1e7
mean ep_avg_losses
episodes
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
500
1000
1500
2000
2500
3000
3500

DQN with RND (Sum of squared differences divided by 2):




mean ep_intrinsic_rewards
0.04
0.03
0.02
0.01
0.00
0
500
1000
1500
2000
2500
3000
3500
episodes


1e7
mean ep_avg_losses
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000
1500
2000
2500
3000
3500
episodes

Explanation of results:

In the conducted experiment, a Mario RL agent was trained using DQN and a combination of DQN with RND. The results demonstrated that the DQN with RND significantly outperformed the DQN-only agent, suggesting that incorporating intrinsic rewards helped the agent explore the environment more effectively and learn a better policy.

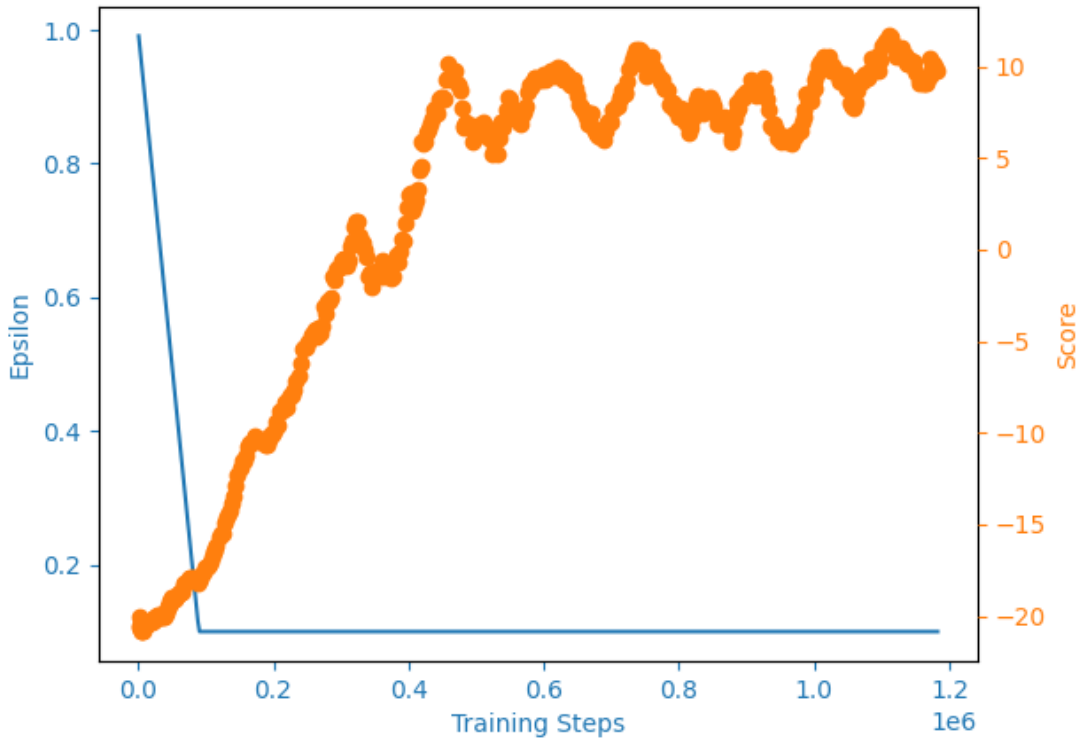
Additionally, the RND-enhanced agent was tested with different loss metrics, including absolute loss. It was observed that using absolute loss led to improved performance initially, but over time the performance started to decrease. This decrease in performance can be attributed to the high intrinsic rewards, which may have led the agent to over-explore the environment. When an agent over-explores, it tends to focus excessively on novel states, possibly neglecting the task's primary objective.

Potential improvements:

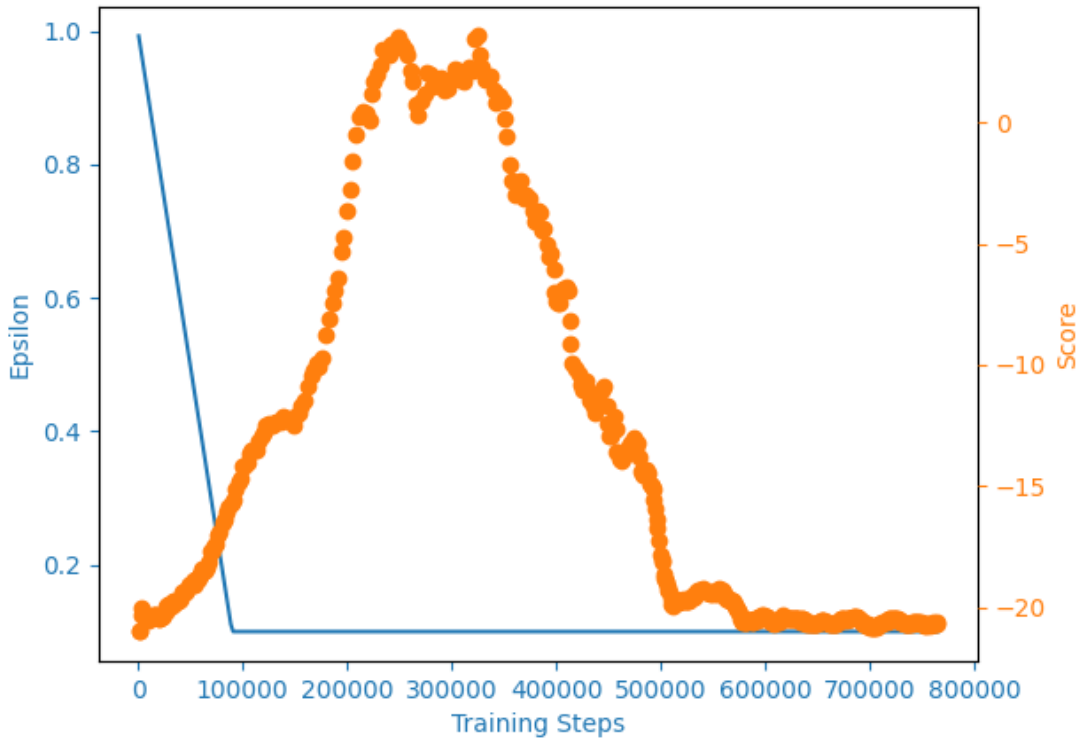
Adaptive scaling of intrinsic rewards. To counter the high intrinsic rewards, implement an adaptive scaling mechanism that reduces the contribution of intrinsic rewards over time. This can help maintain a balance between exploration and exploitation, ensuring that the agent does not over-explore while still benefiting from the RND-driven intrinsic rewards.

Early stopping or learning rate schedule. If the agent starts over-exploring after achieving a certain level of performance, employing early stopping or implementing a learning rate schedule to reduce the learning rate over time. This can help stabilize the agent's performance and prevent it from focusing excessively on exploration at the expense of the primary task objective.

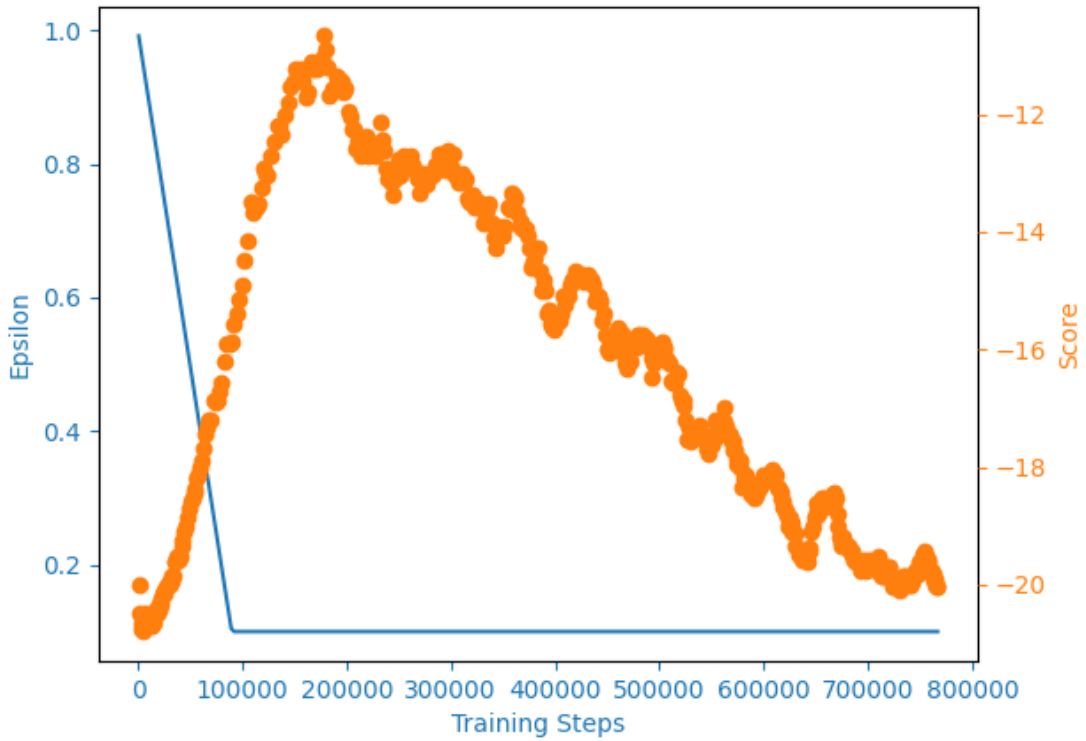
Final experiment – Compression of Episodic Memory in DQN applied to Mountain Cart



No compression



Compression Factor 2.



Compression Factor 5

The impact of compressing episodic memory on the performance of a DQN agent in the Pong environment was investigated. The motivation behind this experiment was to understand the effects of reducing the memory footprint of the agent while still maintaining its ability to learn effectively from past experiences. Two different compression factors were tested: a factor of 2, where the two oldest states were compressed into one, and a factor of 5, where five oldest states were compressed into one.

The results of the experiment showed that initially, the agent's learning process was not affected by the compression, as the episodic memory had not yet reached its capacity. However, as the memory filled up and the compression process began, a decrease in performance was observed. This performance degradation can be attributed to the loss of important information about the environment as a result of the compression process.

Furthermore, it was found that the choice of compression factor had a significant impact on the agent's performance. When using a compression factor of 5, a more dramatic decrease in performance was observed compared to when using a compression factor of 2. This finding suggests that a higher compression factor may lead to a greater loss of valuable information, which in turn hinders the agent's ability to learn effectively.

These results highlight the trade-offs associated with compressing episodic memory in reinforcement learning agents. While memory compression can potentially reduce the computational resources required to store past experiences, it may also lead to a loss of crucial information, ultimately affecting the agent's performance. Further research could explore the use of more sophisticated compression techniques or adaptive compression

strategies to minimize the impact on agent performance while still maintaining the benefits of reduced memory footprint.

Chapter 6: **Conclusion**

This report presents a series of experiments conducted to investigate the effects of different exploration strategies and reinforcement learning algorithms on agent performance in various environments. The primary focus was on DreamerV2, PPO, ICM, and DQN, and their combination with epsilon-greedy, Noisy Networks, and RND intrinsic rewards.

The experiments demonstrated that using advanced exploration techniques and intrinsic rewards can significantly improve agent performance. For instance, combining PPO and ICM resulted in better overall performance compared to using PPO alone, indicating the importance of incorporating intrinsic motivation for exploration. Similarly, combining epsilon-greedy with Noisy Networks led to superior performance in the Breakout environment, highlighting the value of leveraging complementary exploration strategies.

However, the experiments also revealed potential drawbacks associated with some of these techniques. For example, the combination of multiple exploration strategies and intrinsic rewards in the DreamerV2 agent led to a decrease in performance, likely due to over-exploration. Additionally, the choice of loss metrics in the RND agent influenced the agent's performance trajectory, underscoring the importance of appropriate tuning and selection of hyperparameters.

Moreover, the results showed that different exploration strategies and reinforcement learning algorithms might yield varying performance depending on the specific environment and problem. For instance, the continuous state representation in DreamerV2 led to decreased performance in the Breakout environment, which is not well-suited to continuous state-action spaces.

In addition to the exploration strategies and reinforcement learning algorithms examined in this report, the compression of episodic memory was also investigated. The experiment on memory compression in a DQN agent revealed the trade-offs between reducing memory footprint and maintaining agent performance. It was found that compressing episodic memory could lead to a decrease in performance, as crucial information about the environment might be lost in the process. The extent of the performance degradation was also observed to be sensitive to the choice of compression factor. This finding underscores the importance of carefully considering the impact of memory compression techniques on agent performance and highlights the potential need for more advanced compression methods or adaptive strategies that can preserve important information while still reducing memory requirements.

In conclusion, this report demonstrates the potential benefits and challenges associated with combining different exploration strategies and reinforcement learning algorithms. Future research should continue to explore the interplay between these techniques, as well as investigate additional ways to optimize agent performance. Moreover, more extensive experimentation across a range of environments and problem settings will be crucial in understanding the generalizability and limitations of these methods. By further investigating and refining these techniques, we can enable the development of more robust, adaptive, and efficient reinforcement learning agents capable of tackling a diverse array of real-world challenges.

Potential Future Works

The conclusions drawn from this report open up several avenues for future research in reinforcement learning and exploration strategies. One such direction is the utilization of the official TensorFlow implementation of DreamerV2 to conduct further experiments. By using the official implementation, it would be possible to ensure that the experiments are conducted in a consistent and reliable manner, potentially providing more accurate results.

Another important aspect of future work involves extending the experiments to a wider range of environments. By exploring the performance of the reinforcement learning algorithms and exploration strategies in more diverse settings, it would be possible to gain a better understanding of the generalizability and limitations of these methods. This expanded scope of environments would also allow for more insightful comparisons between the techniques.

Given the absence of project deadlines in future works, it would be feasible to train models for longer periods and in more complex environments. Longer training times and increased environmental complexity would likely lead to the discovery of more interesting results, shedding light on the nuances and potential benefits of the reinforcement learning techniques and exploration strategies under investigation.

Furthermore, an intriguing area for future research would be a comparative analysis of the performance of agents using different combinations of reinforcement learning algorithms and intrinsic motivation methods, such as ICM + DQN versus RND + DQN. Although this comparison may be conducted outside the project deadline, it would still provide valuable insights into the effectiveness of different approaches and their applicability to various problem settings. By pursuing these potential future works, we can contribute to the ongoing development and understanding of advanced reinforcement learning techniques, ultimately leading to more robust, adaptive, and efficient agents capable of tackling a wide array of real-world challenges.

References

Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Badia, A.P., Sprechmann, P., Vitvitskyi, A., Guo, D., Piot, B., Kapturowski, S., Tieleman, O., Arjovsky, M., Pritzel, A., Bolt, A. and Blundell, C., 2020. Never give up: Learning directed exploration strategies. arXiv preprint arXiv:2002.06038.

Hafner, D., Lillicrap, T., Norouzi, M. and Ba, J., 2020. Mastering atari with discrete world models. arXiv preprint arXiv:2010.02193.

Hafner, D., Lillicrap, T., Ba, J. and Norouzi, M., 2019. Dream to control: Learning behaviors by latent imagination. arXiv preprint arXiv:1912.01603.

Wu, P., Escontrela, A., Hafner, D., Goldberg, K. and Abbeel, P., 2022. Daydreamer: World models for physical robot learning. arXiv preprint arXiv:2206.14176.

Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z.D. and Blundell, C., 2020, November. Agent57: Outperforming the atari human benchmark. In International Conference on Machine Learning (pp. 507-517).

MLR. Burda, Y., Edwards, H., Storkey, A. and Klimov, O., 2018. Exploration by random network distillation. arXiv preprint arXiv:1810.12894.

Ha, D. and Schmidhuber, J., 2018. World models. arXiv preprint arXiv:1803.10122.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T. and Lillicrap, T., 2020. Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839), pp.604-609.

LeCun, Y. and Bengio, Y., 1995. Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, 3361(10), p.1995.

Savinov, N., Raichuk, A., Marinier, R., Vincent, D., Pollefeys, M., Lillicrap, T. and Gelly, S., 2018. Episodic curiosity through reachability. arXiv preprint arXiv:1810.02274.

Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T., 2017, July. Curiosity-driven exploration by self-supervised prediction. In International conference on machine learning (pp. 2778-2787). PMLR.

Mania, H., Guy, A. and Recht, B., 2018. Simple random search provides a competitive approach to reinforcement learning. arXiv preprint arXiv:1803.07055.