Approximate information state for model-augmented recurrent Q-learning

Erfan Seyedsalehi

Department of Electrical and Computer Engineering
McGill University
Montreal, Canada

April 2023

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Acknowledgements

I would like to take this opportunity to express my gratitude to all those who have supported me throughout the process of completing this thesis.

First and foremost, I extend my heartfelt thanks to my supervisor, Prof. Aditya Mahajan, for his guidance, encouragement, and invaluable feedback. His expertise and insights have been instrumental in shaping my ideas and enabling me to produce the best possible work. This thesis is the result of a long and extensive research process which would not have been possible without his mentorship and experience.

I would also like to thank my mother and father for their unwavering love and support. Their constant encouragement, understanding, and patience have been a source of inspiration for me. I am grateful for their sacrifices and belief in me, without which I would not have been able to achieve this milestone. I would also like to thank my sister, Aida for her constant support and belief in me. Her words of encouragement and positive attitude have been invaluable to me.

Additionally, I would like to acknowledge the contributions of my friends, Milad and Mehran, who have provided me with valuable insights and feedback throughout my research journey. Their support has been invaluable, and I am grateful for their contributions.

Thank you all for your contributions towards the successful completion of this thesis.

Abstract

There has been considerable recent interest in representation learning for Reinforcement Learning (RL), primarily motivated by partially observable and potentially high-dimensional environments where a compact state representation might not be available. One promising approach for this kind of representation learning are model-based methods. There has been considerable progress in our understanding of model-based methods and they have been shown to be effective in circumventing some of the short-comings of model-free methods. In this thesis, we leverage a recently proposed model-based reinforcement learning approach for partially observed environments, which is called the approximate information state approach, for representation learning in recurrent Q-learning based RL algorithms. A salient feature of the proposed algorithm is that its computational complexity is comparable to traditional recurrent Q-learning algorithms. We present detailed numerical experiments to compare our proposed approach with traditional Q-learning methods such as the R2D2 approach. Our experiments demonstrate that the AIS-based Q-learning approach performs better than traditional Q-learning approaches, especially for high-dimensional environments with sparse rewards.

Résumé

Il y a eu récemment un intérêt considérable pour l'apprentissage de la représentation pour l'apprentissage par renforcement, principalement motivé par des environnements partiellement observables et potentiellement de grande dimension où une représentation d'état compacte pourrait ne pas être disponible. Une approche prometteuse pour ce type d'apprentissage de représentation sont les méthodes basées sur des modèles. Il y a eu des progrès considérables dans notre compréhension des méthodes basées sur des modèles et elles se sont avérées efficaces pour contourner certaines des lacunes des méthodes sans modèle. Dans cette thèse, nous tirons parti d'une approche d'apprentissage par renforcement basée sur un modèle récemment proposée pour des environnements partiellement observés, appelée approche d'état d'information approximatif, pour l'apprentissage de la représentation dans des algorithmes d'apprentissage par renforcement récurrents basés sur Q-learning. Une caractéristique essentielle de l'algorithme proposé est que sa complexité de calcul est comparable aux algorithmes d'apprentissage Q récurrents traditionnels. Nous présentons des expériences numériques détaillées pour comparer notre approche proposée avec les méthodes traditionnelles d'apprentissage Q telles que l'approche R2D2. Nos expériences démontrent que l'approche d'apprentissage Q basée sur l'état d'information approximatif fonctionne mieux que les approches traditionnelles d'apprentissage Q, en particulier pour les environnements de grande dimension avec des récompenses rares.

Contents

List of Figures

List of Acronyms

AIS	Approximate Information State.
DDPG	Deep Deterministic Policy Gradient.
DQN	Deep Q-Networks.
GRU	Gated Recurrent Unit.
IPM	Integral Probability Metric.
LSTM	Long Short-Term Memory.
MDP	Markov Decision Process.
MMD	Maximum Mean Discrepancy.
MSE	Mean Squared Error.
PER	Prioritized Experience Replay.
POMDP	Partially Observable Markov Decision Process.
PPO	Proximal Policy Optimization.
QL	Q-learning.
R2D2	Recurrent Replay Distributed Deep Q-Networks.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
SAC	Soft Actor-Critic.
TD	Temporal Difference.

Chapter 1

Introduction

The combination of Reinforcement Learning (RL) with deep neural networks has emerged as a powerful tool for solving a variety of complex tasks that seemed intractable until a few years ago [1; 2; 3; 4]. This combination, now known as deep reinforcement learning, has resulted in the creation of artificial agents that can achieve human-level performance in the games of Go [1], Atari [2], and StarCraft [5]. However, most successful use cases of reinforcement learning have been in settings with fully observable state spaces [3; 4; 6; 2], a characteristic that is not always present in many realistic settings. Reinforcement learning with partial observation is significantly harder because we also need to learn state representations from histories of observations, past actions, and rewards. Approaches aimed at solving partially observable problems are usually too complex and are not as universally successful as methods aimed at fully observable problems. Solving partially observable problems usually requires Recurrent Neural Network (RNN) architectures such as Long Short-Term Memory (LSTM) [7] or Gated Recurrent Units (GRU) [8]. There is a large body of literature on utilizing RNNs with reinforcement learning algorithms [9; 10; 11; 12; 13; 14]. Incorporating RNNs

into existing algorithms is especially difficult with off-policy reinforcement learning algorithms such as Deep Q-learning (DQN) [2] and Soft Actor-Critic (SAC) [6]. These algorithms utilize replay buffers filled with agent experience data to train a policy. Changes to the replay buffer structure and sampling procedure are needed to make Q-learning compatible with RNNs. Multiple solutions have been suggested to solve this problem [9; 10] with the R2D2 method [9] generally considered the standard approach. These methods all rely on a combination of using truncated histories, bundling data from subsequent steps and utilizing saved hidden states of RNNs from past iterations.

Approximate Information State (AIS) [15] provides a general framework for state representation learning in partially observable settings. Furthermore, AIS allows for theoretical reasoning about the performance of policies learned using these learned state representations. It has previously been shown that AIS can be used to augment a non-recurrent policy gradient model-free method and this model-augmented algorithm can outperform a recurrent Proximal Policy Optimization (PPO) [3] implementation on a wide variety of partially observable environments [15].

In this thesis, we propose an AIS-based framework to augment R2D2 recurrent Q-learning [9] with a generative model which is trained on experience data. The AIS model allows us to learn state representations that can be used for Q-learning. R2D2 involves learning from truncated histories and using old stored internal RNN states to initialize the RNN components. We show that an AIS-based generative model can be learned on the same batches of data from the R2D2 replay buffer, and an accurate Q function can be trained using the state representations from this generative model. We use a list of partially observable RL environments such as the MiniGrid environments [16] with varying levels of difficulty. Our method shows a

significant advantage over the standard recurrent Q-learning method in most tested environments. A surprising part of our experimental results is that this advantage is more considerable in environments with very sparse reward signals. Many of these sparse-reward environments are very difficult to solve with vanilla RL methods, and usually, methods with explicit exploration strategies ([17; 18; 19; 20; 21]) are needed for successfully solving them. Our method does not rely on any such strategies but is still quite capable of solving all the difficult tasks in these environments. Finally, we show that our method is compatible with prioritized sampling [22] for both Q-learning and model learning. The addition of Prioritized Experience Replay (PER) provides a boost to our proposed method in difficult environments with very sparse reward signals.

For chapter 2, chapter 3, and chapter 5, A. Mahajan suggested the organization, E. Seyedsalehi wrote all the details and then A. Mahajan provided minor editorial corrections. For chapter 3 and chapter 4, the high-level idea of the algorithm was suggested by A. Mahajan. E. Seyedsalehi implemented the algorithm, did the hyperparameter tuning, and generated all the plots, and wrote the detailed description of the results. A. Mahajan provided minor editorial suggestions to the writing of these chapters.

Chapter 2

Background and Related Work

2.1 Reinforcement Learning

The aim in reinforcement learning is to properly train an agent to solve a sequential decision-making task. This agent periodically interacts with an environment and receives scalar reward signals.

At each step t, the agent receives a state representation s_t or observation o_t (for partially observable environments) and has to output an action a_t. After taking an action the agent receives a scalar reward r_t. The reward is a function of state and action at that time. We are interested in maximizing the sum of rewards or return for the agent. Under an infinite horizon setup, the goal is to maximize the following expectation.

$$\mathrm{E}\left[\sum_{t=0}^{\infty} r\left(s_t, a_t\right)\right] \tag{2.1}$$

In this setup the agent's interactions with the environment continue unless the

agent reaches a terminal state. In many scenarios, the agent needs to interact with the environment in an episodic setting where interactions with an environment continue until a maximum number of steps are taken. These scenarios are called finite-horizon and might be desirable for some practical purposes and also to force the agent to learn how to solve a task in a limited amount of time. An episode ends when the terminal state (success or failure), or a maximum number of steps are taken (in the finite-horizion setup). For the purpose of training an agent, after the end of an episode, the problem is reset, and the agent starts another episode from the initial state.

The agent takes actions a_t according to a function called a policy $\pi(s_t)$. The policy is a mapping from the state space to a distribution over the actions space. The action space can be either discrete or continuous based on the problem setup. In some scenarios, $\pi(s_t)$ can be a deterministic function that outputs a_t directly. The goal of a reinforcement learning algorithm is either to learn a policy $\pi(s_t)$ that outputs actions in the deterministic case or, in the stochastic case, to learn the parameters of a distribution over the action space. We want the optimal policy which can maximize the expected cumulative reward. This optimal policy is denoted by π^*.

The state of the environment evolves according to some internal mechanics. The standard mathematical formalism for reinforcement learning problems comes in the form of a Markov Decision Process (MDP), which will be discussed next.

2.1.1 Markov Decision Process

A Markov Decision Process or MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R)$, which contains the following:

- State space $\mathcal{S}$: A set containing all valid values for state.

- Action space $\mathcal{A}$: A set containing all possible action values. This set can either be finite where each action can be represented by a discrete value or it can be infinite.

- Transition function $P(s'|s,a)$: This is usually a stochastic function which formulates the distribution over next state s_{t+1} given the current state s_t and action a_t at each time step t.

- Reward function $R(s,a)$: A function which outputs the reward r_t at each timestep t given the state s_t and action a_t at that timestep.

We add another component to our problem formulation called the discount factor γ which is a real value number in the range $[0,1)$. The discount factor is meant to represent the current value of rewards received in future steps. A 0 discount factor is attributed to a "myopic" scenario where any reward received in future steps is of zero value at the present and at the other end of the spectrum a discount factor of 1 is attributed a "far-sighted" evaluation scenario where rewards are valued the same no matter how far in the future they are received. Other than being intuitively appealing the discount factor simplifies the convergence analysis of many reinforcement learning algorithms.

If the reward and transition functions are known, planning algorithms can be used to compute the optimal policy π^*. Planning algorithms and Dynamic Programming will be discussed later in this chapter. The major problem is that in most problems, these two functions are unknown. These could include real-world scenarios such as a robot interacting with an environment. The main aim of reinforcement learning is to provide algorithms that are capable of solving these problems.

2.1.2 Dynamic Programming and Planning

We assume both the transition function and the reward function are known. The main problems that we are interested in are policy evaluation (measuring the average return of a given policy in any state) and policy improvement (iteratively improving an initial policy until reaching an optimal policy). If the environment can be modeled as an MDP, we could use a Dynamic Programming decomposition to come up with significantly simplified solutions to both problems. These Dynamic Programming decompositions are the basis for many model-free and model-based algorithms, which will be discussed later.

Let τ be a sequence of state and actions $(s_0, a_0, s_1, a_1, ...)$ starting from the initial state s_0 and first action a_0 going forward. Using the discount factor γ discussed before, we write the infinite-horizon discounted return of a trajectory τ which can be defined as follows.

$$\sum_{t=0}^{\infty} \gamma^t r_t \tag{2.2}$$

We now provide additional definitions which will be useful for this chapter and later on. The first is the value function corresponding to a policy starting from state s which is defined as:

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\tau \sim \pi, P} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]. \tag{2.3}$$

Where the trajectories τ are generated by transition function P and actions are taken according to the policy π. Similarly an action-value function for a policy π starting from state s and taking initial action a can be defined as:

$$Q^{\pi}(s,a) = \mathop{\mathrm{E}}_{\tau\sim\pi,P}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t} \mid s_{0}=s, a_{0}=a\right]. \tag{2.4}$$

Next, we define the optimal value function as:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) = \max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi,P}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t} \mid s_{0}=s\right]. \tag{2.5}$$

A policy which maximizes the value function at all states is called an optimal policy and is denoted by π^{*}. Similarly we define the optimal action value function as:

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) = \max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi,P}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t} \mid s_{0}=s, a_{0}=a\right] \tag{2.6}$$

The problem of learning the value and action value functions for a fixed given policy π is called policy evaluation. Like before, we assume we are dealing with a discrete state and action space which are small enough allowing us to represent the values for different states in the state space as a table. The naive way to learn the value function is to gather many samples from the environment by having the agent interact with it and estimate the value for each state by averaging the computed discounted returns for the samples gathered starting from each state. With enough samples, the values for each state can be computed with a small enough error. Furthermore, by leveraging the Markovian structure of the problem we could come up with a more efficient way of computing the value functions. A value function satisfies the following recursion:

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\substack{a\sim\pi \\ s'\sim P}}\left[r(s,a) + \gamma V^{\pi}\left(s'\right)\right] \tag{2.7}$$

Using this recursive formulation, we can come up with a different approach to

iteratively learning the value function for a fixed given policy. In this iterative process we use the current estimate to update the estimated value for each state. We follow the following update rule:

$$V_{k+1}^{\pi}(s) \leftarrow \underset{\substack{a\sim\pi \\ s'\sim P}}{\mathrm{E}} \left[r(s,a) + \gamma V_k^{\pi}(s') \right] \tag{2.8}$$

With a discrete and small enough state and action space this expectation can be computed exactly since we have access to P. At each step k we update the estimated values for all states. It can be proven that with $\gamma < 1$, following this process will converge to the real value function for policy π with a linear convergence rate of γ. The same approach can be used to learn the action-value function Q^{π} for a given policy π as well.

Similar to any value function, the optimal value function V^* can be recursively written as:

$$V^*(s) = \max_a \underset{s'\sim P}{\mathrm{E}} \left[r(s,a) + \gamma V^*(s') \right] \tag{2.9}$$

This is called the **Bellman equation**. In the tabular setting with discrete state and action spaces, the optimal value function can be obtained via a similar update rule as (2.8). It can be proven that with $\gamma < 1$, the following update rule will converge to V^*:

$$V^*(s) \leftarrow \max_a \underset{s'\sim P}{\mathrm{E}} \left[r(s,a) + \gamma V^*(s') \right] \tag{2.10}$$

Following the iterative process of applying the above update rule for every state until convergence is called **Value Iteration**. Having V^* in the tabular setting is

convenient since by having access to the transition probability, V^* can directly be used to take actions as an optimal action by definition is:

$$\underset{a}{\operatorname{argmax}} \underset{s' \sim P}{\mathrm{E}} \left[r(s,a) + \gamma V^* \left(s' \right) \right] \tag{2.11}$$

The requirement of having known transition dynamics and reward functions and also discrete state and action spaces which are relatively small makes this algorithm intractable for most reinforcement learning problems but this simple algorithm is the basis for much more capable reinforcement learning algorithms which will be discussed next.

2.1.3 Model-free Reinforcement Learning

In most reinforcement learning problems, transition and reward functions are unknown, making the previously discussed algorithms useless. For these problems, a good policy needs to be learned from data gathered by the agent. The agent's interactions with the environment give us episodes of experience in the form of tuples of state, action, next state and reward (s, a, s', r) which can be used to learn good policies. Reinforcement learning algorithms can generally be put into two major categories of **Model-free** and **Model-based** reinforcement learning. Model-free reinforcement learning algorithms involve learning policies directly from the data gathered by the agent and model-based algorithms involve learning the transition and reward functions from experience data using machine learning approaches and utilizing the learned models for obtaining a good policy. There are certain algorithms that combine components from both model-free and model-based methods to train the agent. For now, we will discuss the different model-free methods that can be used to train an agent. Most model-free

reinforcement learning algorithms can generally be put into the two categories of:

- **Q-learning algorithms**: In these algorithms, the goal is to learn an approximation of the optimal action-value function Q^*. In this family of algorithms, we utilize the Markovian structure of the MDP and train the action-value approximation function Q_θ with θ as the function parameters using the Bellman equation for the optimal action-value functions. Training Q_θ using this loss is done in an **Off-policy** fashion. This means that learning Q_θ can be done using all the data gathered by the agent. In discrete action settings Q^* can directly be used to get the best policy as the best action $a = \underset{a}{\text{argmax}}\, Q^*(s, a)$. This allows us to train the agent by only learning Q_θ. These algorithms can also be extended to be used with continuous action settings. In this case the policy can be represented by a parametric differentiable model and Q_θ is similarly trained using the Bellman equation. If Q_θ is also a differentiable model, it can directly be used as the loss function for the policy, allowing us to train the policy. Q-learning based algorithms are usually substantially more sample efficient than Policy Gradient based algorithms but they generally suffer from more instability and sensitivity to hyperparameters.

- **Policy Search algorithms**: In these algorithms, a policy is represented by a parametric differentiable model and a reformulation of the episodic discounted return is used as the loss function. The policy is then learned directly using gradient descent based optimization methods. These methods are called **On-policy** because at each step optimization is done using data gathered by the most recent policy only. Being On policy means that a policy cannot be learned using older data gathered at previous iterations making these algorithms less

data efficient. The most basic algorithm in the family is **REINFORCE** [23]. Using a learned value function (critic) can make this algorithm significantly more efficient resulting in the **Actor-Critic** family of model-free algorithms. Policy Gradient algorithms are compatible with both continuous action and discrete action reinforcement learning settings and they are usually stable and reliable for a variety of reinforcement learning problems.

Model-free algorithms are responsible for some of the biggest successes of reinforcement learning and they are used for training agents that can solve the games of Atari [2; 24; 22; 25], Dota [26] and Starcraft [5]. They are also used for solving a wide variety of Robotic simulation tasks [3; 4; 6; 27]. The versatility and reliability of these algorithms makes them a popular choice for solving reinforcement learning problems but they generally suffer from very high data inefficiency. This high data inefficiency makes these algorithms suitable only in scenarios where vast amounts of data can be gathered very cheaply. Reinforcement learning problems where an efficient simulator is available are usually the most suitable problems for these algorithms.

2.1.4 Q-learning

In discrete-action environments, optimal behaviour can be learned without directly needing to learn a mapping π_θ from the state space to action space. Instead we can learn the optimal action-value function Q^* and take actions by using that because by definition $\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a)$. The optimal action-value function can be written as:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = \max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi,P}[R(\tau) \mid s_0 = s, a_0 = a]$$
$$= \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*\left(s', a'\right)\right]$$

In a Markov Decision Process, sequential states in a trajectory are generated according to a transition probability function $P(s'|s,a)$ which defines the distribution for the state at each step according the state and action at the previous step. The Bellman equation for value functions was discussed before. The second line of the above equation is called the Bellman equation for action value function. This equation is the basis of the vanilla Q-learning algorithm which learns Q^* in the tabular setting using the following update rule:

$$(s,a,r,s') \sim D \; : \; Q(s,a) \leftarrow Q(s,a) + \alpha \; \left[r(s,a) + \gamma \max_{a'} Q\left(s', a'\right) - \; Q(s,a)\right] \quad (2.12)$$

The vanilla Q-learning algorithm is an iterative algorithm which involves using the above update function on agent experience data gathered in a Dataset D. In Q-learning, two types of policy are used. The first is a behaviour policy and it is only used to generate experience data. The above update rule is used on the gathered data. During evaluation, we act greedily with respect to the learned Q-function which we assume approximates Q^*. It is proven that vanilla Q-learning converges to Q^* for an MDP as long as all state-action pairs are updated infinitely often [28]. This means that the behaviour policy should not be a greedy policy with respect to the learned Q-function since it will most likely miss some states. It is standard to use an

exploratory policy such as an ϵ-greedy policy for generating experience data. This is a policy that randomly switches between taking random actions and behaving greedily with respect to Q. ϵ is considered a hyperparameter which needs to be tuned and it determines the probability of the agent behaving randomly. We could extend the tabular Q-learning case to a setting where the action-value function is approximated by a parametric function such as a multi-layer neural network. The vanilla case of Q-learning with gradient descent based optimization is as follows:

$$\phi \leftarrow \phi \; + \; \alpha \nabla_{\phi} \underset{s' \sim D}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q_{\phi'}\left(s', a'\right) - \; Q_{\phi}(s,a) \right]. \tag{2.13}$$

One important detail in this optimization objective is that the target Q-learning approximation $Q_{\phi'}$ uses different parameters from the main Q-function Q_{ϕ}. Using the same parameterized function in both the target and the value prediction causes the optimization to collapse and therefore a different Q-function is used as the target. It is suggested in [2] that the target Q-function $Q_{\phi'}$ should be periodically updated with the most recent Q-function approximation Q_{ϕ}. This approach is generally followed in most Q-learning approaches. Another alternative is to use exponential averaging to get the target Q-function parameters after each gradient descent update [29]. Experimental results in [30] suggest learned approximations of the Q-function can often suffer from overestimation of the real action-values. A solution to this problem called **Double Q-learning** [31] is to learn two Q-function approximations and using each as the target for learning the other. The two approximations are then trained according to the following:

$$\phi_1 \leftarrow \phi_1 \; + \; \alpha \nabla_{\phi_1} \underset{s' \sim D}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q_{\phi_2}(s',a') - \; Q_{\phi_1}(s,a) \right]$$
$$\phi_2 \leftarrow \phi_2 \; + \; \alpha \nabla_{\phi_2} \underset{s' \sim D}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q_{\phi_1}(s',a') - \; Q_{\phi_2}(s,a) \right]$$

Q-learning can utilize data from any policy to learn the optimal behaviour. A learning scheme was suggested in [2] in which an agent periodically interacts with an environment according to an exploratory policy and gathers experience and stores this experience in a large array called the **Replay Buffer**. The Q-learning updates are done using uniformly sampled data from this buffer. This is the standard approach for most Q-learning based algorithms. At any given step Q_ϕ can be very accurate at predicting the action-values for some action-value pairs and it might be very inaccurate with others. This accuracy could be because some states are visited more often and therefore data corresponding to them might be more abundant in the replay buffer. Updating Q_ϕ for states on which it is accurate could result in less meaningful updates than updating it on states on which it is inaccurate. An approach was introduced in [22] where every stored data in the buffer has a priority. This priority is proportional to the action-value prediction error and it determines the probability of that data being sampled for Q-learning. Sampling high priority data might cause Q_ϕ to overfit to that data which is undesirable. According to [22], it is suggested to weigh the updates for each sampled data in a way that higher priority samples have smaller updates. This is intuitively telling the optimizer to do bigger updates when it sees a low priority sample. This approach is called **Prioritized Experience Replay** and involves doing Q-learning updates according to the following:

$$\phi \leftarrow \phi \; + \; \alpha \, w \, \nabla_{\phi} \mathop{\mathrm{E}}_{s' \sim D} \left[r(s,a) + \gamma \max_{a'} Q_{\phi'}\left(s', a'\right) - \; Q_{\phi}(s,a) \right]$$
$$w = \left(\frac{1}{N} \cdot \frac{1}{p} \right)^{\beta}$$

In the above equation, p corresponds to the sampling probability of the given data (s, a, r, s') and N is the replay buffer size with β being a hyperparameter in this algorithm regulating how much the update weights should differ among samples. Prioritized Experience Replay is shown to vastly improve Q-learning performance in many envrironments [22; 24] and is a standard approach used with many off-policy approaches today. Another area where Q-learning performance can significantly be improved is to use multi-step bootstrapping ideas similar to what is used with policy evaluation. The Bellman equation for Q^* of an arbitrary state s_0 can be written as:

$$Q^*(s_0, a) = \mathop{\mathrm{E}}_{s_{0:k}, a_{0:k-1} \sim \pi^*, P} \left[\sum_{t=0}^{k-1} r(s_t, a_t) \; \gamma^t + \gamma^k \max_{a'} Q^*\left(s_k, a'\right) \right] \tag{2.14}$$

An update based on the multi-step Bellman equation for Q^* can be obtained except it requires the k-length sequence of rewards to come from the same policy being updated. This means that we can no longer learn Q^* off-policy removing the biggest advantage of Q-learning based algorithms. In order to solve this, we need to readjust the learning rate with an importance sampling ratio ρ. Assuming we are learning Q^* from a sequence of experience $(s_0, a0, \; ..., s_{k-1}, a_{k-1}, s_k)$ from an older policy π^{old}, the importance sampling ratio becomes $\rho = \prod_{t=0}^{k} \frac{\pi(a_t|s_t)}{\pi^{old}(a_t|s_t)}$. This is meant to readjust for the probability of having that experience with the new policy. Then the multi-step Q-learning update to Q_{ϕ} with importance sampling becomes:

$$\phi \leftarrow \phi \; + \; \alpha \, \rho \nabla_{\phi} \mathop{\mathrm{E}}_{s_{0:k}, a_{0:k-1} \sim D} \left[\sum_{t=0}^{k-1} r(s_t, a_t) \, \gamma^t + \gamma^k \max_{a'} Q_{\phi} \left(s_k, a' \right) - \; Q_{\phi}(s_0, a_0) \right]$$

$$\rho = \prod_{t=0}^{k-1} \frac{\pi \left(a_t \mid s_t \right)}{\pi^{old} \left(a_t \mid s_t \right)}$$

Using the above update rule requires π^{old}. This means we need to save $\pi(a|s)$ for every saved trajectory. In practice omitting ρ from the update often works very well and in most experimental setups multi-step Q-learning is done without importance sampling.

Q-learning based approaches can be extended to Actor-Critic cases as well where we have an explicitly parameterized policy [32; 29]. This approach is useful because it allows us to expand off-policy model-free reinforcement learning to continuous action problems. In continuous action problems, computing $\operatorname{argmax}_a Q^*(s, a)$ might not be easy when a complex function approximator such as a multi-layer neural network is used to model the Q-function. Instead we could learn a parametric function for the policy π_θ and directly use Q_ϕ to optimize that policy. $Q_\phi(s, \pi_\theta(s))$ can be used as the objective for learning π_θ. This allows us to learn π_θ using gradient descent. This approach is the basis of Actor-Critic off-policy algorithms that are generally considered the most sample efficient model-free methods for solving continuous control problems [27; 6; 33].

2.1.5 Model-based Reinforcement Learning

Another approach for solving reinforcement learning problems is to first learn the transition $P(s_t|s_{t-1}, a_{t-1})$ and reward functions $R(s, a)$ and then use the learned

functions to solve the RL problem. As the agent interacts with the environment, we can save the (s, a, r, s') tuples in a dataset and use that data to train the transition and reward functions with standard supervised learning approaches. These approaches all fall under the Model-based reinforcement learning umbrella and there is a wide variety of ways in which a learned model can be utilized. We assume in this part that the problem can be modeled as an MDP.

In scenarios where the transition function is a deterministic one meaning $s_t = f(s_{t-1}, a_{t-1})$, we can use multilayered neural networks to learn the transition function. In stochastic cases with continuous state spaces we can train the multilayered neural network to output the parameters of either a Multivariate Gaussian distribution or a Mixture of multivariate Gaussian distributions. The same can be done for the reward function. The experience in the dataset is gathered by an agent following a specific policy which is initially a completely random agent and in later iterations it is improved using the model. Model-based approaches usually involve iteratively training the models and acting in the environment using a policy which is improved after each iteration. The new experience is then added to the dataset so that the learned model can be further fine-tuned to the new data. Policies could be obtained using decision-time planning approaches such as the Cross-Entropy Method (CEM) [34; 35]. Usually during the initial phases of training when the agent follows a random policy data could be scarce and only a subset of the state-space is explored. This makes Model-based approaches vulnerable to overfitting in the initial phases of training. It is useful if the knowledge that the learned model is not accurate in some parts of the environment can be encoded into the model architecture. We aim to deal with a problem called Epistemic uncertainty which is the learned model's lack of knowledge

about the world. [36] suggests using an ensemble of multilayered neural networks each outputting the parameters of a multivariate Gaussian distributions. Each of these models are trained on a subset of the gathered experience so far. These subsets are randomly created at each step and are reshuffled before each training step. Using an ensamble of learned models during planning allows us to mitigate the problem of model overfitting. The Cross-Entropy method used with ensemble model learning shows great performance in robotic control problems [36].

Even the most data-efficient model-free algorithms such as DQN [2] and SAC [6] are usually very data inefficient for practical real world problems. Another way a learned model can be utilized is by generating synthetic training data for model-free methods such as Q-learning. This approach can be very useful in problems where interacting with the environment can be very costly. Generating data with a learned model can be very cheap and this could potentially provide a big boost to the data requirements of model-free methods. The first approach based on this idea is Dyna and was first suggested in [37] and is a useful model-based technique which is especially powerful when combined with off-policy model-free methods such as Q-learning [2] and Soft Actor-Critic [6]. In Dyna we use the learned models as an alternative to interacting with the real environment where generating data is significantly more expensive. This cheap data generation allows us to significantly expand the replay buffer with samples that could be used to learn the Q-function. In Dyna with off-policy methods, two replay buffers are used. The first only includes agent's interactions with the real environment which are saved as tuples of state, action, reward and next state (s, a, r, s'). These samples are used to train the model. In order to create more data for the model-free algorithm, we sample from this replay buffer and generate

new trajectories of data using the learned models and the policy. The length of these synthetic trajectories (also called imagined trajectories) should be kept limited as learned models are noisy and these errors can become larger the longer an imagined trajectory becomes. The real data samples alongside imagined samples are added to the second buffer. We sample from the second buffer to boost the training of the Q-function for Q-learning or Soft Actor-Critic. [36] shows this approach to be very powerful when combined with an ensemble of probabilistic models similar to [36] in order to mitigate model inaccuracy.

Another way learned models can be used to help with model-free reinforcement learning algorithms is by allowing for on-policy data generation for multi-step Q-learning. As discussed, importance weights are needed to adjust the probability of a trajectory from the replay buffer to the latest policy. A learned model can be used to generate an on-policy multi-step trajectory starting from the sampled state. This can be done on the fly when Q-learning update is being done. [38] combines this approach with DDPG [33] and shows a great improvement in performance in continuous control tasks. Model-based approaches generally help reinforcement learning in fully observable environments by reducing the number of required interactions with the real environment. Model-based algorithms play a big role in reinforcement learning in partially observable environments as well and they often provide perks beyond only data efficiency in these cases. Some of those approaches will be discussed later in this chapter.

2.2 Partially Observable Reinforcement Learning

We assume that states are fully observable in the problems that we are interested in modeling with an MDP. This is in contrast to partially observable environments, which are modeled by Partially Observable MDPs (POMDP). In reality, many problems that we encounter do not have this fully observable characteristic, as knowing the state space requires good knowledge of the underlying dynamics and structure of the environment and also access to some variables that might not be easily accessible. Nevertheless, full observability significantly simplifies the reinforcement learning problem and allows us to use very effective approaches to tackle the problem. Some of these approaches were discussed in the previous sections. Building on those approaches, we can introduce more complicated algorithms for partially observable settings.

2.2.1 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) can be defined by a set $(\mathcal{S}, \mathcal{A}, T, R, \Omega, O)$. The first four components are identical to an MDP in their definition. The final two components are defined as:

- Observation space Ω: A set containing all possible valid values for observation.
- Observation mapping function O: This is a function which formulates the probability distribution over the observation space given the state s_t and actions a_t at timestep t. We represent this distribution as $O(.|s, a)$.

In POMDPs, the state evolves in a Markovian fashion dictated by the transition probability. Similar to an MDP, the distribution of state at each step is dependent on the state and action at the previous step. The crucial difference between an MDP and

POMDP is that the agent does not see the state s_t but sees observations o_t at each step. The requirement of having access to state representations makes MDPs restrictive for modelling many problems. Instead many of these problems can be modeled with POMDPs. Observations can be low dimensional vectors or also high dimensional vectors such as images. It is common in many partially observable problems that observations are images representing what the agent sees when interacting with the environment.

Not having access to state representations makes the problem of learning optimal policies significantly harder. Unlike states in MDPs, observations at each step do not contain all the necessary information for learning good policies and their value functions. To alleviate this problem, the agent needs to take actions based on the entire history of observations that it has seen so far from the start of the episode. We define history h_t of observations as a tuple $(o_1, o_2, ..., o_t)$ including all the observations from the first step until timestep t. In some algorithms the history also includes all the received rewards and actions from the first step to timestep t. A major issue that arrises with using function approximators such as neural networks is that depending on t the history (if represented as a concatenated vector) can constantly expand in terms of size, meaning usual neural network architectures such as the MLP cannot be used to process the history. A natural solution to this problem are recurrent neural networks (RNNs) which are made to deal with variable length sequences. RNNs allow us to work with variable length sequences while having learnable parameters. The Long Short-Term Memory (LSTM) [7] variant of RNNs is chosen because of its improved performance and versatility in sequential machine learning problems. Recurrent neural networks allow us to learn a compact representation from history which then can used

as input to simple neural network architectures such as MLPs which will be used as approximators for the Q-function and the policy.

Most Deep reinforcement learning algorithms for solving POMDPs utilize RNNs for compressing the history into a compact representations. Adding RNNs requires modifications to be made to Deep RL algorithms that were discussed before. First, we will discuss model-free approaches to partially observable reinforcement learning problems.

2.2.2 Model-free partially observable reinforcement learning

In this section, we will discuss how model-free methods can be modified so that they can be used for partially observable reinforcement learning problems. This requires incorporating a memory processing structure such as an RNN into the algorithm. Incorporating RNNs with policy-gradient based methods are generally simpler as they do not deal with replay buffers and therefore agents are trained on entire sequences of newly generated data. An actor-critic methods with RNNs is shown to be a powerful approach for solving some partially observable problems [39; 5; 26].

Deep Q-learning (DQN) can also be modified to work in partially observable settings with history of observations used to determine the policy. In DQN for fully observable problems, agent experience was saved as tuples of state, action, reward and next state (s, a, r, s') in a replay buffer and was later uniformly sampled for training the Q-function approximator Q_ϕ. Each step of the interaction is treated as a separate sample here and the Q-learning update can be done on each sample independently. Assuming we use the sequence of observations $(o_1, o_2, ..., o_t)$ as history h_t input to the RNN model, computing the correct Q-function for each sample requires

the observations from all the preceding timesteps that were encountered during the data gathering phase. Since the preceding observations are not available to us when sampling from the DQN replay buffer, a change needs to be made to the algorithm to accommodate recurrence with DQN.

Deep recurrent Q-learning (DRQN) [10] suggests two solutions to this problem:

- Saving a fixed length preceding history for each sample and initializing the RNN internal state to zero at the start of this fixed length sequence. This approach is the simplest for incorporating RNNs with DQN and allows us to do uncorrelated sampling from the buffer which is an important part of DQN. Initializing the RNN hidden state to zero at the start of each saved sequence is problematic as it prevents the RNN from learning the true hidden state for each sample but it also forces the RNN to learn representations even from an atypical initial hidden state. A problem with this approach is the much higher memory requirement of the replay buffer compared to non-recurrent DQN case and using this approach forces us to save the observations from each state multiple times in the replay buffer.

- Saving entire episodes in the buffer and doing Q-learning updates on all of the states in the sampled episode. This fixes the issue of unreliable RNN hidden states but causes other serious problems. First is that states in a single episode are highly correlated and doing updates on all states from an episode can be problematic and unstable as DQN requires uncorrelated sampling for training. The second problem is with the varying lengths of episodes which combined with the impossibility of parallelizing computation across a sequence results in significant slowdowns. An advantage of this approach is memory efficiency as

each encountered state is saved only once in the buffer. Also, this approach provides computational benefits in a different manner that will be discussed in more detail.

In [10] very little difference in performance was observed between the two strategies in Atari games, and therefore the first approach was chosen due to its simplicity. Most environments in the Atari suite [40] are almost fully observable as it is very common to do frame-stacking with non-recurrent Q-learning to learn optimal Q-functions for these problems [2]. [9] suggests the almost "full-observability" of Atari environments might be the reason that very little difference in performance is observed between the two approaches as in the first approach information from a limited number of preceding states is only considered during training (essentially similar to frame stacking only with RNN based architectures). Using the same approach in problems with higher levels of partial observability such as DMLab [41] could prevent the RNN from learning representations useful to solving the problem. [9] aims to augment the first approach so that it is more suited to harder partially observable reinforcement learning problems. [9] suggests saving the hidden RNN states during data gathering in the replay buffer. During Q-learning these saved hidden states are used to initialize the RNN; Next, similar to the first approach in [10] a fixed length sequence of preceding states (called **Burn-In**) are saved with each sample in the buffer. Both the saved RNN state and the Burn-In sequence are used to initialize the model and afterwards, Q-learning is done on the samples. Using saved hidden states to initialize the RNN could be problematic and cause instability during training as sampled sequences could come from older iterations where the RNN weights were significantly different from the current RNN weights. Having a sufficiently long Burn-In sequence length is suggested in conjuntion

with initializing the RNN hidden state with saved values to allow the RNN to recover good hidden states before the hidden values are used for doing Q-learning.

The second approach in [10] can suffer from training instability and computational problems (stemming from variation in sequence length) but it provides a benefit. Since all consecutive states in an episode share their preceding histories, the second approach provides the benefit of doing Q-learning for all samples in an episode by computing a sequence of RNN hidden states only once. [9] suggests bundling a fixed length sequence of consecutive states in the replay buffer and sampling them all together to do Q-learning updates. This is advantageous as the Burn-In and the saved recurrent hidden state are shared for all states in a sample subsequence. This still could lead to training instability with Q-learning updates and therefore the length of these subsequences should be limited. The approach in [9] also utilizes distributed reinforcement learning similar to [39] but the suggested approach can easily be implemented without distribution of data gathering and learning between a number of threads. Also, [9] uses prioritized experience replay with multi-step Q-learning. The approach in [9] is called Recurrent Replay Distributed DQN (R2D2) and is considered the best performing recurrent Q-learning based algorithm for partially observable environments. The following is a pseudocode of the non distributed R2D2 style Q-learning algorithm with single-step Q-learning updates:

The hyperparameters in this algorithm include:

- N or "Burn-In Length" is the length of the preceding burn-in sequence used during Q-learning sampling.

- L or "Subsequence Length" is the length of the subsequence of states from each episode which are stored together in the replay buffer. Later Q-function training

Algorithm 1 Recurrent Replay Q-learning with uniform sampling

init σ, θ to random networks, init $\theta' \leftarrow \theta$ and $\mathbf{D} \leftarrow \{\}$
for $k \in 0, \ldots, M$ **do**
  start episode, init history $h \leftarrow \{\}$
  while not done **do**
    receive observation o_t, append to history h
    take some action a_t given by $\text{argmax}_{a_t} Q_\theta(\sigma(h, 0))$ with probability $1 - \epsilon$ and uniform random action with probability ϵ {Epsilon-Greedy Action Selection}
    Sample batch of experience sequences $(h_{1:N}, z, o_{1:L}, r_{1:L}, a_{1:L})$ from $\mathbf{D}$
    Initialize σ with z and burn-in history to σ and detach gradients: $z' = \sigma(h_{1:N}, z)$
    $y_i \leftarrow r_i + \gamma \, \text{argmax}_a \, Q'(\sigma(o_{1:i+1}, z'))$ {Compute Target}
    Update θ and σ by minimizing $\frac{1}{L} \sum^{L} ||Q(\sigma(o_{1:i}, z')) - y_i||^2$ {Update critic}
    Update $\theta' \leftarrow \sigma\theta' + (1 - \sigma)\theta$ {Target update using exponential averaging}
    if t mod L is 0: append $(o_{i-L-N:i-L-1}, \sigma(h_{i-L-N}, 0), o_{i-L:i}, r_{i-L:i}, a_{i-L:i})$ to $\mathbf{D}$ {Add subsequence to buffer}
  end while
end for=0

will be done on all of these states together at each iteration.

The original approach in [9] uses multi-step Q-learning and prioritized experience replay which are not included in Alg 1. Also, Alg 1 uses exponential averaging similar to [27; 6; 33] for target Q updates which is different from the original R2D2 approach which uses hard updates for the target Q-function [9]. Alg 1 is the basis for a more complicated model-based approach which is the focus of this thesis and will be introduced in the next chapter. First, we need to introduce model-based techniques for Partially observable environments.

2.2.3 Model-based Partially observable reinforcement learning

Similar to Model-free methods for partially observable problems, model-based methods can also be extended to these problems. The problem of not having access to state

representations forces us to change the structure of the approximate transition and reward models. Also the observation mapping function is another unknown that needs to be learned. Similar to model-free methods recurrent architectures are necessary for doing model-based RL in partially observable environments. If the unknown transition, reward and observation mapping function are learned they can be used in similar ways that were discussed before. We could do online planning with the learned models using the cross-entropy method (CEM) [34; 35] for continuous action environments and Monte-Carlo tree search [42; 43] for discrete action environments.

The idea of the state in an MDP (or POMDP) is a compact representation of the history of every observation, action and reward encountered so far and is denoted by $h_t = (o_1, a_1, r_1, ...o_t)$ at every timestep t. A state gives us all the information about what has happened before and what is about to happen next and it is sufficient for taking optimal actions. If we were to create a custom hidden state representation, we want to be able to predict the values of this hidden state representation at next timestep $t + 1$. We also want to be able to predict rewards at each step r_t given the hidden state representations and action at the same timestep. The problem of model-based RL in POMDPs then becomes:

- Learning a function which maps current observation and previous actions, rewards and hidden states to the hidden state representations at the current step.

$$z_t = f_h(o_t, a_{t-1}, r_{t-1}, z_{t-1})$$

- Learning a function which predicts observations from hidden state representa-

tions.

$$o_{t+1} = f_o(z_t, a_t)$$

- Learning a function which maps hidden state representation and action at current step to the reward at current step.

$$r_t = f_r(z_t, a_t)$$

The first component maps history at each timestep to hidden state representations. Since history at each step is a subset of history at later stages, we can use a recursive function which maps observations at the current step and reward, action and the hidden state at the previous step to the hidden state at the current step. This formulation is convenient since it allows us to process actions, rewards and observations at each step only once but more importantly this formulation can be modelled using a recurrent neural network (RNN) as the hidden state of the RNN at each step can be used as the hidden state of the POMDP and (o_t, a_{t-1}, r_{t-1}) can be used as inputs to the RNN at each step. We are assuming the hidden state can be deterministically predicted at each step given the previous value and the latest observation, action and reward. This may not be a correct assumption as some environments have stochastic elements in their transitions as well. [13; 11] suggests adding stochasticity by including a stochastic variable which is predicted from the hidden state at each step and is fed as an input to the RNN at the next step. In this work, we assume next state transitions can be predicted deterministically. The other two components include functions that predict the reward and next observations given the current hidden state and action.

Since the ground truth values for the hidden state are unknown, we cannot directly

train f_h on sequences of experience data but rewards and observations are known. Therefore, we can train f_h, f_o and f_r together so that they can correctly predict observations and rewards.

In many partially observable problems such as Atari [40], DMLab [41], DMcontrol [44] and the MiniGrid environments [16], observations are high-dimensional (such as Images) and are not well shaped be used directly as inputs to the RNN. Another function is needed to map these high dimensional observations to lower dimensional compact representations which are better suited to be processed by an RNN. In the World Models approach [14] the encoder unit of a pretrained Variational Autoencoder (VAE) [45] is used. This VAE is trained on a dataset of random policy experience and the encoder weights are fixed in the reinforcement learning phase. More advanced methods such as Dreamer [11] and PlaNet [13] jointly train the observation encoder with the transition dynamics function and reward prediction function using a loss derived from the variation lower bound on the log-likelihood of observations.

It is shown in [11; 15; 14] that if we follow the discussed process of training the RNN encoder so that it can predict observations and rewards, then the hidden internal state of the RNN can serve as a replacement for the state representations and it can be used as an input to value function and parameterized policies which are modelled with non-recurrent neural network architectures and do not have access to full histories. This thesis proposes using the learnable state representations of a transition model for doing non-recurrent Q-learning for partially observabele problems. The idea of using pretrained observation encoders from the World Models approach [14] will also be used for some of our experiments.

2.3 Approximate Information State

In this section, we aim to go over the various topics and concepts which are relevant to the understanding of the Approximate Information State (AIS) concept which was introduced in [15]. This section includes a summary of parts of [15] which are instrumental in the understanding of AIS. We will also go over the model-augmented policy-gradient algorithm which was introduced in [15]. We will later build on these concepts to introduce an AIS-based model-augmented recurrent Q-learning method which is the main focus of this thesis.

In POMDPs, the belief state of the agent is the posterior belief of the unobserved state given all the observation history gathered by the agent. It is known that Partially Observable MDPs could be transformed into fully observable Markov Decision Processes by utilizing the belief state as an information state [46]. This information state allows us to use Dynamic Programming based approaches for policy evaluation and policy iteration on POMDPs. Assuming we have a series of history compression functions called information state generators: $\{\sigma_t : \mathrm{H}_t \to \mathrm{Z}_t\}_{t=1}^{T}$. The hidden states at each timestep t are $Z_t = \sigma_t(H_t)$. Given these definitions, the desired information state has to satisfy the following properties:

- **P1:** It has to be sufficient for performance evaluation or it has to be sufficient for reward prediction at each step. This condition can be written as:

$$\mathbb{E}[R_t \mid H_t = h_t, A_t = a_t] = \mathbb{E}[R_t \mid Z_t = \sigma_t(h_t), A_t = a_t]$$

- **P2:** It has to be sufficient to predict itself. The means that it has to be sufficient to predict future hidden states given current hidden states. This condition can

be written as:

$$\mathbb{P}\left(Z_{t+1} \mid H_t = h_t, A_t = a_t\right) = \mathbb{P}\left(Z_{t+1} \mid Z_t = \sigma_t\left(h_t\right), A_t = a_t\right)$$

While condition **P1** is easier to verify, verifying condition **P2** can be more complicated depending on the problem setup. Instead it would be more helpful if the following stronger conditions were verified:

- **P2a:** It has to evolve in a state-like manner. This means that series of functions $\{\varphi_t\}_{t=1}^{T}$ exist that for any timestep t, we have:

$$\sigma_{t+1}\left(h_{t+1}\right) = \varphi_t\left(\sigma_t\left(h_t\right), y_t, a_t\right)$$

  φ_t is an alternative function which uses the information state at previous steps Z_{t-1} as input. The φ_t formulation is more useful as it can be modelled with a recurrent neural network.

- **P2b:** It is sufficient for predicting future observations. This means that for any timestep t we should have:

$$\mathbb{P}\left(Y_t \mid H_t = h_t, A_t = a_t\right) = \mathbb{P}\left(Y_t \mid Z_t = \sigma_t\left(h_t\right), A_t = a_t\right)$$

It is proven in [15] that if both conditions **P2a** and **P2b** hold, then **P2** also holds. It is also shown in [15] that $Z_t = \sigma_t\left(H_t\right)$ can be used to turn a POMDP into an MDP with Z_t as state representations at each step. Having the information state allows us to used MDP algorithms on a Partially Observable problem setting.

The most obvious examples for an information state as defined above would be the full history of actions and observations $h_t = (o_{1:t-1}, a_{1:t-1})$ and also the hidden POMDP state s_t as described in 2.2.1. The first option is a hidden state representation that linearly grows in dimensionality with each timestep making this option unsuitable for even the simplest partially observable environments. The second option requires knowledge and information of the hidden parts of the underlying system which is not available as their availability would turn this problem from a partially observable problem into a fully observable problem modelled by an MDP. Neither option is feasible for most partially observable problems. In the case of an arbitrary partially observable problem with no knowledge of the underlying dynamics, it is unclear if an information state formulation (other than the above two) exists at all. Using the definition of information state, we now move onto the Approximate Information State (AIS) formulation. With AIS, we want to come up with approximate conditions that are both feasible and verifiable in a data-driven experimental setting. First we have to define Integral Probability Metrics (IPM) which are fundamental to our definition of an approximate information state.

Let's assume we have $(\mathrm{X}, \mathcal{G})$ as a measurable space [47] and have $\mathfrak{F}$ as the set of all uniformly bounded measurable functions on $(\mathrm{X}, \mathcal{G})$. The Integral Probability Metric between two distributions $\mu, \nu \in \Delta(\mathrm{X})$ with respect to function class $\mathfrak{F}$ can be written as:

$$d_{\mathfrak{F}}(\mu, \nu) := \sup_{f \in \mathfrak{F}} \left| \int_{\mathbf{X}} f d\mu - \int_{\mathbf{X}} f d\nu \right| \tag{2.15}$$

Total variation distance [48] and Maximum mean discrepancy [49; 50; 51; 52] are examples of IPMs [53]. We also use $\rho_{\mathfrak{F}}$ which is the Minkowski functional defined for

function space $\mathfrak{F}$:

$$\rho_{\mathfrak{F}}(f) := \inf \left\{ \rho \in \mathbb{R}_{>0} : \rho^{-1} f \in \mathfrak{F} \right\}$$

Furthermore, the contraction factor $\kappa_{\mathfrak{F}}(\ell)$ for arbitrary function l from function space $\mathfrak{F}$ is defined as:

$$\kappa_{\mathfrak{F}}(\ell) = \sup_{f \in \mathfrak{F}} \rho_{\mathfrak{F}}(f \circ \ell).$$

Both the contraction factor and the Minkowski functional will be used later. Now we can define the approximate conditions which are necessary for an Approximate Information State formulation. Similar to the information state case, we have a series of history compression functions $\left\{ \hat{\sigma}_t : \mathrm{H}_t \to \hat{\mathrm{Z}}_t \right\}_{t=1}^{T}$ which map full histories of action and observations to hidden state representations. We also have approximate update kernels $\left\{ \hat{P}_t : \hat{Z}_t \times \mathrm{A} \to \Delta\left(\hat{Z}_{t+1}\right) \right\}_{t=1}^{T}$ and reward approximation functions $\left\{ \hat{r}_t : \hat{Z}_t \times \mathrm{A} \to \mathbb{R} \right\}_{t=1}^{T}$. This collection is called an $\{(\varepsilon_t, \delta_t)\}_{t=1}^{T}$-AIS generator if the process $\left\{ \hat{Z}_t = \hat{\sigma}_t\left(H_t\right) \right\}_{t=1}^{T}$ satisfies the following conditions:

- **AP1:** It has to be sufficient for approximate performance evaluation. Meaning reward prediction error at each timestep t should be upper-bounded as follows:

$$\left| \mathbb{E}\left[R_t \mid H_t = h_t, A_t = a_t \right] - \hat{r}_t\left(\hat{\sigma}_t\left(h_t\right), a_t \right) \right| \leq \varepsilon_t$$

- **AP2:** It has to be sufficient to approximately predict itself. Meaning:

$$d_{\mathfrak{F}}\left(\mathbb{P}\left(\hat{Z}_{t+1} \mid H_t = h_t, A_t = a_t \right), \hat{P}_t\left(\hat{Z}_{t+1} \mid \hat{\sigma}_t\left(h_t\right), a_t \right) \right) \leq \delta_t$$

If these conditions hold, we call this collection an $\{(\varepsilon_t, \delta_t)\}_{t=1}^{T}$-AIS generator. Similar to an Information state generator, we provide an alternative to **AP2** which is more practical to work with [15]. We replace **AP2** with **AP2a** and **AP2b** as follows:

- **AP2a:** The AIS hidden state evolves in a state-like manner. Meaning there is a function $\left\{\hat{\varphi}_t : \hat{Z}_t \times \mathrm{Y} \times \mathrm{A}\right\}_{t=1}^{T}$ such that for history h_{t+1} for timestep $t+1$ we have:

$$\hat{Z}_{t+1} = \hat{\sigma}_{t+1}\left(h_{t+1}\right) = \hat{\varphi}\left(\hat{\sigma}_t\left(h_t\right), y_t, a_t\right)$$

- **AP2b:** It is sufficient for predicting future observations approximately. This condition is very similar to **P2b** of the Information state case. For this condition to hold, there should exist a measurable observation prediction kernels $\left\{\hat{P}_t^y : \hat{Z}_t \times \mathrm{A} \to \Delta(\mathrm{Y})\right\}_{t=1}^{T}$ for each timestep t then:

$$d_{\mathfrak{F}}\left(\mathbb{P}\left(Y_t \mid H_t = h_t, A_t = a_t\right), \hat{P}_t^y\left(Y_t \mid \hat{\sigma}_t\left(h_t\right), a_t\right)\right) \leq \delta_t / \kappa_{\mathfrak{F}}\left(\hat{\varphi}_t\right)$$

  Here, $\kappa_{\mathfrak{F}}\left(\hat{\varphi}_t\right)$ is defined as $\sup_{h_t \in \mathrm{H}_t, a_t \in \mathrm{A}_t} \kappa_{\mathfrak{F}}\left(\hat{\varphi}_t\left(\hat{\sigma}_t\left(h_t\right), \cdot, a_t\right)\right)$. For example, if Total Variation Distance is chosen as the IPM, $\kappa_{\mathfrak{F}}\left(\hat{\varphi}_t\right) = 1$.

Similar to the Information state case, In [15] it is proven that **AP2a** and **AP2b** imply **AP2**. Working with **AP2a** and **AP2b** is significantly easier in practical settings as both conditions can be easily verified. The significance of this Approximate Information State formulation is that if the corresponding generator and prediction functions exist and they satisfy the stated conditions, the hidden representations can be used to do both value function approximation and optimal policy derivation with provable error bounds.

We can use history generator functions corresponding to an Approximate Information State for value function approximation and approximately optimal policy derivation. We now assume that the problem is infinite horizon and discounted with a γ as the discount factor. If we have $\left(\{\hat{\sigma}_t\}_{t\geq 1}, \hat{P}, \hat{r}\right)$ as a time-homogeneous (ε, δ)-AIS generator, we can define approximate action-value functions $\left\{\hat{Q} : \hat{Z} \times \mathbf{A} \to \mathbb{R}\right\}$ and value functions $\left\{\hat{V} : \hat{Z} \to \mathbb{R}\right\}$ as:

$$\hat{Q}(\hat{z}, a) := \hat{r}(\hat{z}, a) + \gamma \int_{\hat{Z}} \hat{V}(\hat{z}') \hat{P}(d\hat{z}' \mid \hat{z}, a),$$
$$\hat{V}(\hat{z}) := \max_{a \in \mathrm{A}} \hat{Q}(\hat{z}, a).$$

We know that the above equation has a fixed point solution for $\gamma < 1$. If the fixed point of the above equation for the action-value function was denoted by $\hat{Q}^*$ and the corresponding value function for this fixed point solution were $\hat{V}^*$, then the following bounds for value function approximation was shown in [15]:

$$\left|Q_t(h_t, a_t) - \hat{Q}^*(\hat{\sigma}_t(h_t), a_t)\right| \leq \alpha \quad \text{and} \quad \left|V_t(h_t) - \hat{V}^*(\hat{\sigma}_t(h_t))\right| \leq \alpha$$

with $\alpha = \frac{\varepsilon + \gamma \rho_{\mathfrak{F}}(\hat{V}^*)\delta}{1-\gamma}$. The above bounds mean that we can get sufficiently accurate action-value and value function approximations in a POMDP by relying on AIS generated hidden states. We can also derive bounds over value and action-value functions for an optimal policy $\hat{\pi}^* : \hat{Z} \to \Delta(\mathrm{A})$ that satisfies:

$$\mathrm{Supp}(\hat{\pi}^*(\hat{z})) \subseteq \arg\max_{a \in \mathrm{A}} \hat{Q}^*(\hat{z}, a)$$

If we were to define policy $\pi_t : \mathrm{H}_t \to \Delta(\mathrm{A})$ by $\pi_t := \hat{\pi}^* \circ \hat{\sigma}_t$. Then, for any timestep

t, we have:

$$|Q_t(h_t, a_t) - Q_t^{\pi}(h_t, a_t)| \leq 2\alpha \quad \text{and} \quad |V_t(h_t) - V_t^{\pi}(h_t)| \leq 2\alpha.$$

Furthermore, [15] shows that an Approximate Information State generator satisfying AP1, AP2a and AP2b can be used for deep reinforcement learning. As the hidden representations generated by an AIS generator can be used to train a policy using the vanilla policy gradient model-free algorithm. The performance of this approach is demonstrated over a wide variety of partially observable environments. First, the two IPMs used in [15] for constructing the AIS generators should be discussed first as different IPMs affect both the loss functions and also the structure of the neural network models.

One choice for the IPM is the the Total Variation Distance. Using Pinsker's inequality [54], we can derive an upper bound for the Total Variation Distance involving the Kullback–Leibler divergence:

$$d_{\mathrm{TV}}(\mu, \nu) \leq \sqrt{2 D_{\mathrm{KL}}(\mu \| \nu)}.$$

KL divergence between two densities μ and ν over $\Delta(\mathrm{X})$ is defined as [55]:

$$D_{\mathrm{KL}}(\mu \| \nu) = \int_{\mathrm{X}} \log \mu(x) \mu(dx) - \int_{\mathrm{X}} \log \nu(x) \mu(dx)$$

The aim is to learn a parametric function approximating μ while ν is the real distribution generating the real data. We can omit the second term of the above expression as it does not depend on ν and use the first term as a loss to be used with gradient descent based optimizers. The first term is the cross-entropy between the

distributions μ and ν which can be easily computed over minibatches of experience data. Another choice for the IPM is the distance-based MMD (Maximum Mean Discrepency) which can be written as:

$$d_{\mathfrak{F}_p}(\mu,\nu) = \sqrt{\mathbb{E}\left[d_{\mathrm{X},p}(X,W)\right] - \frac{1}{2}\mathbb{E}\left[d_{\mathrm{X},p}\left(X,X'\right)\right] - \frac{1}{2}\mathbb{E}\left[d_{\mathrm{X},p}\left(W,W'\right)\right]}$$

with $X, X' \sim \mu, W, W' \sim \nu$ and all being independent random variables. $d_{\mathrm{X},p}\left(x,x'\right) = \|x - x'\|_2^p$ and for $p = 2$, we have the following simplified form of an MMD-based loss for learning AIS generators:

$$\left(M_\xi - 2X\right)^\top M_\xi$$

Here M_ξ is the approximated mean of the distribution and X is a sample from the distribution. Mean of the above over minibatches of experience data can be used to train an AIS generator. Here, for continuous valued random variables, M_ξ would be the mean of the random variable and for discrete valued variables, we would be using one-hot encoded vectors of the random variable as X and the vector of probabilities (the probability mass function of the multinomial distribution) as M_ξ. The random variables for which we want to learn the distribution over are the next step observations in the case of going with AP2a and AP2b. The KL-divergence loss includes log probability terms for data samples and therefore we are required to learn a parametric function for the whole distribution. For continuous-valued variables, the distributions are modelled with Mixture of Gaussians and all the components such as the means and variances for all Gaussian distributions and the multinomial probabilities for each Gaussians should be modelled. For the MMD-based loss, we only

need to learn the mean of the distribution and therefore the function approximators are much simpler. Please note that the generative models are not used for simulating the environment and creating future trajectories as with many other model-based approaches. The generative model component is only trained with respect to a model loss (MMD or KL-divergence) and then the hidden states of the RNN component are used for reinforcement learning. The aim is to seperate the RNN learning part from the reinforcement learning part. For this purpose, the MMD-based loss is suitable both for being simpler and also that full sample generation is not needed at all.

In [15], a time-homogeneous AIS-generator $\left(\hat{\sigma}, \hat{r}, \hat{\varphi}, \hat{P}^y\right)$ that satisfies AP1, AP2a and AP2b is used with the two IPM choices outlined above. $\hat{\sigma}$ and $\hat{\varphi}$ can be both modelled using a time-series function approximator. In this case an LSTM [7] is used to model these two components together. $\hat{r}$ can be modelled with any function approximator such as multi-layer perceptron. $\hat{P}^y$ has a different structure depending on the IPM used and also whether observations are continuous or discrete random variables. For the MMD case, $\hat{P}^y$ can be modelled with an MLP outputting the mean of the observations distribution. For discrete valued observations, an MLP is used with a Softmax final layer used to model the probability mass function of the multinomial distribution for the observations. For the KL case, $\hat{P}^y$ has a much more complex structure as it has to model a mixture of Gaussian distributions with three seperate MLPs modelling the multinomial component, the means and the variances. For discrete valued observations, the KL case is exactly the same as the MMD case. $\left(\hat{\sigma}, \hat{r}, \hat{\varphi}, \hat{P}^y\right)$ are all trained using the following loss:

$$\frac{1}{T}\sum_{t=1}^{T}\left[\lambda\left|R_t - \hat{r}\left(\hat{Z}_t, A_t\right)\right|^2 + (1-\lambda)d_{\mathfrak{F}}\left(\mu_t, \nu_t\right)^2\right]$$

with the second term being the MMD-based or the KL loss. [15] uses the above loss to train an AIS generator to be used with a stochastic policy which in turn is trained using the vanilla policy gradient approach [56]. The RNN-based history compression component is only trained on the above loss and later the hidden AIS representations are used as inputs to a stochastic policy which is trained using the policy gradient objective:

$$\widehat{\nabla}_\theta J(\bar{\xi}, \theta) = \sum_{t=1}^{T} \left(\sum_{\tau=1}^{t} \nabla_\theta \log \pi_\theta \left(A_t \mid \hat{Z}_t \right) \right) \gamma^{t-1} R_t$$

Value functions are not used and the reward-to-go approach is used for approximating returns for each state. It is demonstrated that this approach shows a performance superior to recurrent on-policy model-free approaches (PPO+LSTM) on a wide variety of environments ranging from simple discrete observation space environments to the much more difficult and diverse family of MiniGrid environments [15]. For MiniGrid, the high-dimensional observations are first compressed using a simple autoencoder [57] and the compressed representations are used as input observations to the AIS components. The simple autoencoder is pretrained on a dataset of random agent experience similar to the World Models approach [14]. The autoencoder is not updated during the reinforcement learning phase. Building on these findings, in the next chapter, we outline an approach to augment recurrent Q-learning using AIS models.

Chapter 3

Recurrent Q-learning with AIS models

In this chapter we outline the main proposed algorithm which is an AIS based model-augmented recurrent Q-learning approach to solve partially observable environments. Similar to what is done in [15] and was discussed in the previous chapter, the aim is to learn a time-homogeneous AIS-generator $\left(\hat{\sigma}, \hat{r}, \hat{\varphi}, \hat{P}^y\right)$ which satisfies AP1, AP2a and AP2b but this time in conjunction with doing Q-learning. The major difference is that the AIS components will be trained on data coming from an R2D2-style [9] recurrent replay buffer. This means that: 1) the data is generated by various different policies (unlike the approach in [15] in which AIS model learning was done on-policy), 2) histories for sampled trajectories will be truncated up to fixed lengths in accordance to the R2D2 approach discussed in Alg 1.

We will use the MMD loss for future observation prediction which was outlined in the previous chapter. The KL based loss does not perform well in our experiments and using the MMD loss also provides computational benefits because of its simpler structure (we only need to predict the mean). Our approach involves training the AIS model components on the same minibatch of samples that we do Q-learning on.

Similar to the vanilla policy gradient approach in [15] and the Dreamer approach [11] the hidden representation of the RNN are used as state representations inputs to the non-recurrent Q function approximator. The parameters of the AIS components remain fixed during the Q-learning phase and the Q-learning loss is only used for training the Q-function. We have two versions of this algorithm:

- Uniform replay buffer: Samples will be drawn from the R2D2 replay buffer uniformly.
- Priority replay buffer: Data will be saved in a prioritized replay buffer [22]. Priorities are based on Q-funtion errors. Q-function and AIS model updates are both done using weighted losses.

We use the standard prioritized experience replay implementation [22]. The samples are drawn from the replay buffer with probability $(\delta_i + e)^\alpha$ with δ_i being the Q-function prediction error. Also, e is a hyperparameter used to prevent samples with very low error having an extremely small priority. α is also a prioritized replay hyperparameter. More details about the PER hyperparameters are provided in appendix A. Similar to the standard prioritized experience replay buffer approach [22], the weights for each sample are:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{p_i}\right)^\beta$$

Here, N is the size of the buffer, p is the sampling probability and β is another prioritized replay hyperparameter. Smaller values of β allow for closer to uniform weights. We start with a small value for β and slowly anneal its value to 1.

We use the MMD-based loss discussed in the previous chapter to train the observation prediction component. Here, Y_t is the observations for a sample (or observation encodings in the Minigrid case) and M_t is: 1) The probability mass in the discrete observation case which is modelled with a softmax layer, 2) The mean of the continuous valued observation encodings which are in both cases predicted by $\hat{P}^y$. The loss is as follows for each sample in the batch:

$$\left(M_t^y - 2Y_t\right)^\top M_t^y \tag{3.1}$$

For predicting rewards at each step, we use MLPs which need to predict rewards at each step given the hidden state representations and action taken at that step. A standard Mean Square Error (MSE) loss is used for training the reward prediction component. The joint model loss is as follows which includes the hyperparameter λ:

$$\frac{1}{T}\sum_{t=1}^{T} w_i \cdot \left[\lambda \left| R_t - \hat{r}\left(\hat{Z}_t, A_t\right)\right|^2 + (1-\lambda) d_{\tilde{\mathfrak{F}}}\left(\mu_t, \nu_t\right)^2\right] \tag{3.2}$$

The weight of this sum λ is between 0 and 1 and is a hyperparameter which requires tuning during experimentation. In our experiments, it is set to numbers very close to 0 as we found the observation prediction component to have a much bigger impact on the performance of Q-learning algorithm relying on the hidden state representations. w_i is equal to 1 for the uniform variant and equal to the previously discussed prioritized replay weight.

The Q-function is modelled by a multi-layered MLP which receives the hidden state representations of the RNN component and outputs approximated Q values for all possible actions at that step. We use the multi-step TD loss to train the Q-function

while the parameters of the RNN component are frozen. The target for the multi-step Q-learning loss is as follows:

$$\hat{y_t} = \sum_{k=0}^{n-1} r_{t+k}\gamma^k + \gamma^n \, Q\left(h_{t+n}, a^*; \theta^-\right) \, , \quad a^* = \arg\max_a Q\left(h_{t+n}, a; \theta\right). \tag{3.3}$$

We use θ^- to compute target Q-values. θ^- refers to the target Q-function weights which are updated via exponential averaging at every step similar to [6]. a^* refers to the optimal action at timestep $t+n$ and is obtained from the Q-function rather than the target network. This approach is similar to the double Q-learning approach used in [31] which was previously discussed. The length of the multi-step updates is also a hyperparameter which requires tuning during experimentation. The following loss is used for training the Q-function:

$$\frac{1}{T}\sum_{t=1}^{T} w_i \cdot \left[|y_t - \hat{y_t}|^2\right] \tag{3.4}$$

Here, similar to the model training loss, the sample weights are set to 1 for the uniform sampling variant and to the prioritized weights for the variant using prioritized experience replay.

Similar to the R2D2 recurrent Q-learning algorithm discussed before, samples are drawn from an R2D2 style replay buffer and are used to train the different model components and the Q-function. Also, for the variant with the prioritized replay, priorities are updated after batches are used to train the different components. In the high-dimensional environments, we use pretrained autoencoder functions to compress the higher dimensional observations into more compact continuous representations

before the reinforcement learning phase. The encoders are trained jointly alongside a decoder so that together they can recreate the observations. The observations come from a dataset of random agent experience. We follow an approach similar to [15; 14].

The following is a pseudocode of the model-augmented recurrent Q-learning algorithm with a uniform sampling replay buffer:

Algorithm 2 Q-learning+AIS algorithm

init σ, $P^{y,r}$, θ to random networks, init $\theta' \leftarrow \theta$ and $\mathbf{D} \leftarrow \{\}$
for $k \in 0, \ldots, M$ **do**
  start episode, init history $h \leftarrow \{\}$
  while not done **do**
    receive observation o_t, append to history h
    take some action a_t given by $\text{argmax}_{a_t} Q_\theta(\sigma(h, 0))$ with probability $1 - \epsilon$ and uniform random action with probability ϵ {Epsilon-Greedy Action Selection}
    Sample batch of experience sequences $(h_{1:N}, z, o_{1:L}, r_{1:L}, a_{1:L})$ from $\mathbf{D}$
    Initialize σ with z and burn-in history to σ and detach gradients: $z' = \sigma(h_{1:N}, z)$
    Compute model loss for $(z', o_{1:L}, r_{1:L}, a_{1:L})$
    Update σ and $P^{y,r}$, σ weights are fixed for the next part
    $y_i \leftarrow r_i + \gamma \, \text{argmax}_a \, Q'(\sigma(o_{1:i+1}, z'))$ {Compute Target}
    Update θ by minimizing $\frac{1}{L} \sum^{L} ||Q(\sigma(o_{1:i}, z')) - y_i||^2$ {Update critic}
    Update $\theta' \leftarrow \rho\theta' + (1 - \rho)\theta$ {Target update using exponential averaging}
    if t mod L is 0: append $(o_{i-L-N:i-L-1}, \sigma(h_{i-L-N}, 0), o_{i-L:i}, r_{i-L:i}, a_{i-L:i})$ to $\mathbf{D}$ {Add subsequence to buffer}
  end while
end for=0

Chapter 4

Experiments

To evaluate the proposed algorithms, two different sets of environments are used. The first set includes two discrete observation space environments with low dimensional observations. The second set is a list of procedurally generated environments from the MiniGrid family [16]. These environments include very strong elements of partial observability and are very difficult to solve without the addition of recurrent networks. We will compare six algorithms on these environments. The last two are included to decouple the effect of using pretrained encoders (which can also be used with the recurrent Q-learning algorithm) from the effect of AIS models and their learned representations. Please note that the autoencoders are only useful in the MiniGrid environments as they include high dimensional observations. In the discrete observation environments, we only test the the first four algorithms.

- **QL+AIS+U**: In this algorithm, the recurrent unit is trained using the AIS generative model training approach discussed in the previous chapter. The Q-function will receive the inputs from the recurrent unit at each step and it will

be trained with the multi-step Q-learning loss discussed previously. The data for training the recurrent unit, the model prediction components and the Q-function are sampled uniformly from a replay buffer. For the MiniGrid environments, high-dimensional observations will be compressed first using an encoder unit from a simple autoencoder. The encoder is pretrained before the Q-learning phase alongside a decoder on a dataset of random policy interactions. The weights of the encoder do not change during the main RL phase.

- **QL+AIS+PER**: This algorithm is similar to the previous algorithm as it includes a main loop consisting of both model-learning and Q-learning for training the recurrent unit and the Q-function respectively. The data for training the recurrent unit, the model prediction components and the Q-function are sampled non-uniformly from a Prioritized Experience Replay buffer and the prioritized experience replay weights are applied to both the model-learning and the Q-learning losses as was discussed in the previous chapter. Similar to the previous algorithm, a pretrained encoder is used to compress the observations for the MiniGrid environments and the weights of the encoder do not change during the main RL phase.

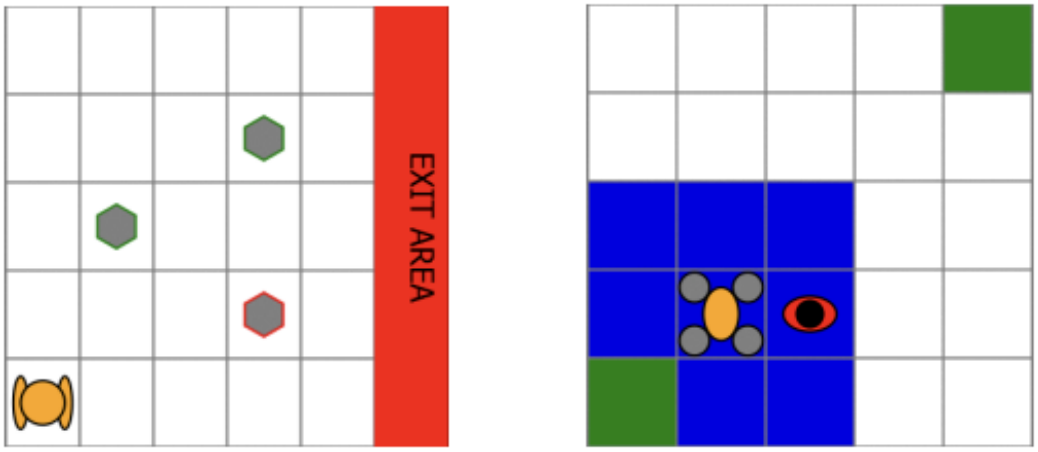
- **QL+U**: This is the recurrent Q-learning algorithm discussed before (Alg 1). In this method, both the recurrent unit and the Q-function are trained using the multi-step Q-learning loss. The data for learning these two components is sampled uniformly from the replay buffer. This component does not use a pretrained encoder but the MLP layers for encoding the higher dimensional observations are included in the recurrent unit.

- **QL+PER**: This is our recurrent Q-learning algorithm with Prioritized Experience Replay. The recurrent unit and the Q-function are trained using the multi-step Q-learning loss similar to the uniform variant. The prioritized weights are used for training both components. This algorithm also does not use pretrained encoders for MiniGrid environments but includes the MLP layers in the recurrent unit.

- **QL+U+AE**: This variant is similar to the recurrent Q-learning algorithm 1 "QL+U" but also uses pretrained encoders on input observations.

- **QL+PER+AE**: This variant is similar to the recurrent Q-learning algorithm with prioritized replay "QL+PER" but also uses the pretrained encoders.

4.1 Discrete Observation Environments



(a) Rock Sampling (b) Drone Surveillance

Fig. 4.1 A visualization of the two discrete action-space environment used in this part.

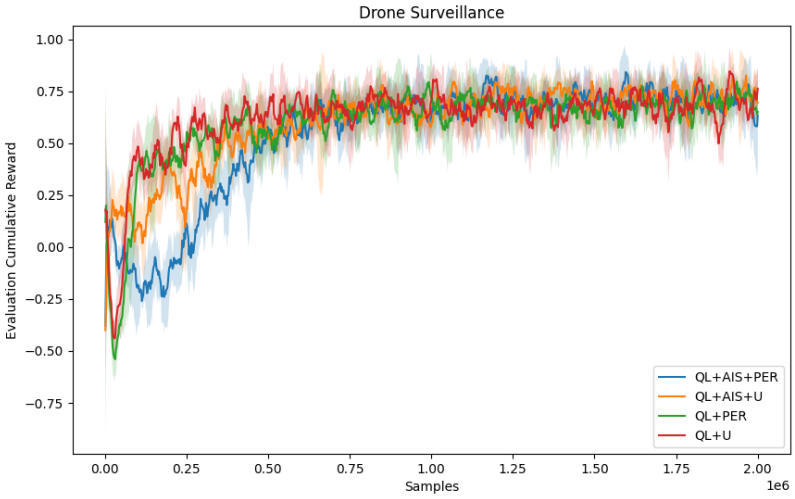
First, we will test the proposed algorithms on the following two environments. Both

have discrete observation space. Due to the low dimensionality of the observations, the observations can be directly fed to the recurrent unit without relying on pretrained observation encoders.

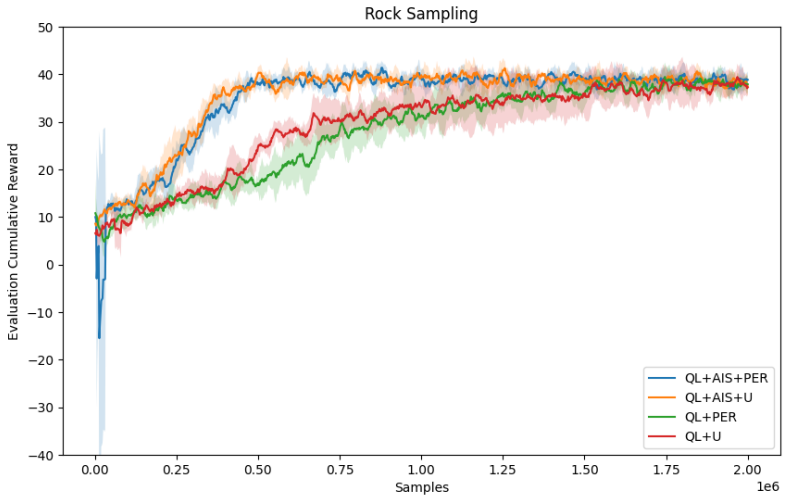
- **Rock Sampling:** This environment was first introduced in [58] and involves a rover which is tasked with exploring an area and finding rocks with good scientific value. There are both good rocks and bad rocks in the environment. The goal of the agent is to sample the good rocks while avoiding the bad ones and exit the area afterwards. The number of rocks k and the size of the $n \times n$ grid is custom but is chosen as 3 and 5 respectively. The agent needs to choose from $k+5$ actions at each step: 4 move actions in each direction, 1 sample action and k sense action. Each sense action corresponds to one of the rocks. The position of these rocks are fixed for the environment but the type of the rocks (whether each of them is a good rock or no) randomly changes at the start of each episode. The observation space is discrete and observations can take three values. The agent receives a null observation unless it takes a sense action corresponding to a rock. In that case the observation can take two values according to a binomial distribution with the p-value corresponding to the distance to the specific rock. The sensor becomes more accurate the closer the rover gets to the specific rock. For good rocks the probability of seeing 1 becomes larger as the rover gets closer the the rock and for bad rocks it is the opposite. The challenge of this environment is that the agent is effectively blind and has a very noisy tool for learning about the rock types. Sampling good rocks gives the agent a reward of +20 and sampling bad rocks gives the agent a -10 reward. Exiting the area also has a +10 reward for the agent.

- **Drone Surveilance:** This environment was first proposed in [59]. The goal is to move a drone in a grid while avoiding being in the same location as a randomly moving ground agent. If the drone and the ground agent are in the same location a reward of -1 is received and the episode ends. The drone always starts the episode at the bottom left corner of the grid and the goal state is always at the upper right corner of the grid. The ground agent starts the episode at a random location and it can never enter the start and goal location. Both the drone and the ground agent move one tile at each step. The drone can only see the ground agent if it is located in a 3×3 grid underneath the drone. The size of the overall grid is 5×5.

The performance plots for these two environments can be seen in 4.2. The model-augmented variants and especially the model-augmented variant with prioritized replay perform worse on the Drone Surveillance environment. On Rock Sampling, the model-aumgented variants show a significant and clear boost to performance. Another observation is that using Prioritized Experience Replay either does not provide any advantage or it hurts performance in these two environments. This phenomena will be seen again on many MiniGrid environments and it is only on the significantly more challenging MiniGrid environments that Prioritized Experience Replay starts to show a benefit over uniform sampling. We believe the reason for this is that it will take some time for the Q-function to have a good approximation for action-values and having good action-values is necessary for priorities in the PER to be accurate. Also, the AIS model takes some time to train and we generally see that in the easier environments which can be very quickly solved with Q-learning, the model-augmented approach provides a smaller benefit.



(a) Drone Surveillance



(b) Rock Sampling

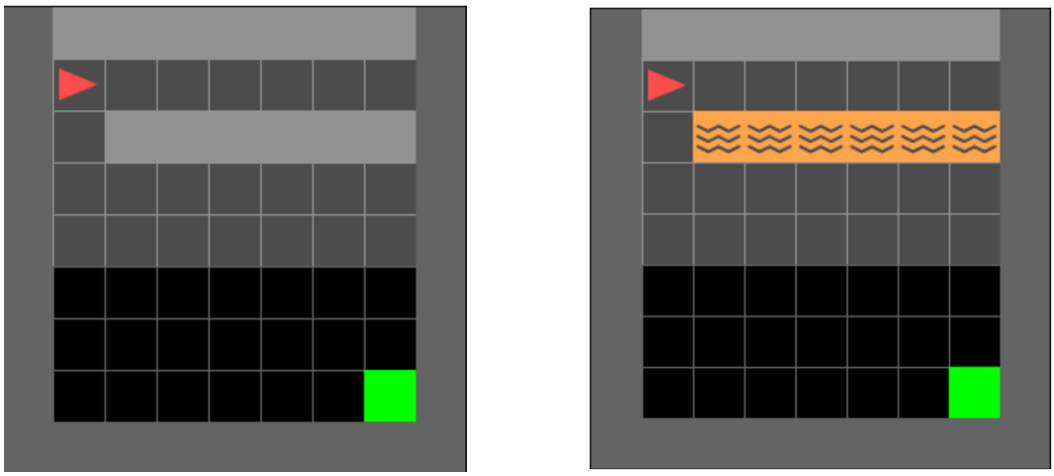
Fig. 4.2 The performance of recurrent Q-learning and Q-learning with AIS model learning on Rock Sampling and Drone Surveillance environments. Both algorithms have variants with Prioritized Experience Replay and uniform sampling buffers.

4.2 MiniGrid

Now we will test the six mentioned algorithms on the environments from the MiniGrid family [16]. These are gridworld environments where the agent needs to solve a variety of tasks from navigation to object manipulation. We consider these environments to be significantly more challenging than the previously discussed discrete observation space environments. First, the observation space in these environments is a $7 \times 7 \times 3$ vector corresponding to the 7×7 grid around the agent. Please note that this is not an RGB image but is a custom encoding including information about each tile in the 7×7 grid around the agent. Second, these environments feature very sparse rewards. Except for the Dynamic Obstacles environments, in all MiniGrid environments the agent only receives a nonzero reward after a successful episode. No rewards are given before the end of the episode and only successfully solving the task results in the agent receiving a $+1$ reward. Depending on the environment, there may be objects randomly located in the world where the agent has to interact with. These objects can be boxes, doors, keys, etc. We have divided these environments into groups based on the nature of their respective tasks and we will analyze the performance of the two variants of the model-augmented algorithm against the four variants of the recurrent Q-learning algorithm.

4.2.1 Crossing Environments

The first group of environments are the Crossing environments. The goal in these environments is for the agent to learn how to navigate an area and reach a goal. There are two categories of environment in this group:



(a) Simple Crossing (b) Lava Crossing

Fig. 4.3 A visualization of the Simple Crossing and the Lava Crossing environments.

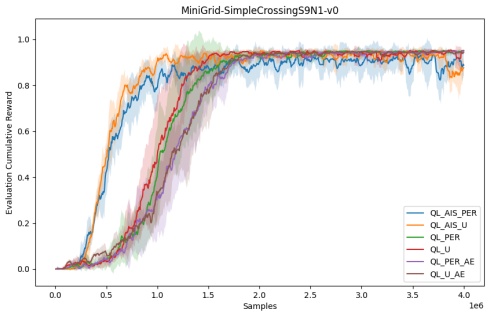
Simple Crossing: The environments in this group consist of columns of walls and a crossing. This is the simplest MiniGrid environment used for our empirical evaluation. The Simple Crossing environments are identified by SimpleCrossingSnNm with n being the size of the $n \times n$ grid and m being the number of columns that the agent needs to traverse.

Lava Crossing: In these environments the agent needs to navigate its way in a room filled with columns of lava. Entering these columns of lava results in failing of the episode. The Lava Crossing Environments are identified by LavaCrossingSnNm with n being the size of the $n \times n$ room and m being the number of lava blocks.

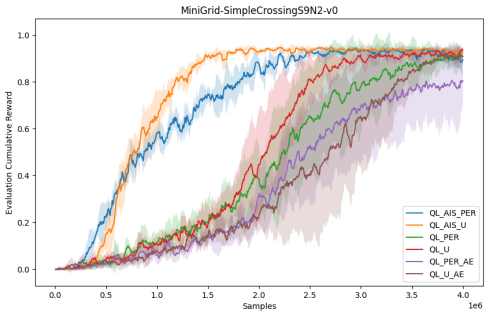
The performance of the six algorithms on four Simple Crossing and four Lava Crossing environments can be seen in 4.4. On Simple Crossing environments, there is a clear gap between the model-augmented Q-learning and Q-learning algorithms. Using prioritized Experience replay on model-augmented algorithms does not provide any benefits in the two smaller environments but shows improvements in the two

larger environments. Especially on SimpleCrossingS11N5, we see a big gap between "QL+AIS+PER" and everything else. This is expected as we believe the advantage of using PER grows in more difficult and larger environments. Using a pretrained encoder does not provide a tangible benefit to the recurrent Q-learning approaches.

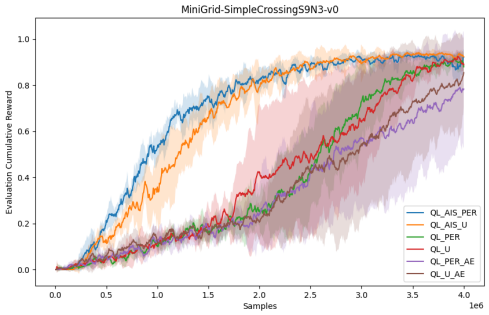
On Lava Crossing environments, the uniformly sampled Q-learning does much better, coming very close to the model-augmented variants in the three smaller environments. Still, the model-augmented algorithms do better on average on the three smaller environments. On LavaCrossingS11N5, the six algorithms do not do very well. Both "QL+AIS+PER" and "QL+U" do better than the other two with the model-augmented algorithm leading by a small margin in terms of average performance. Using a pretrained encoder with Q-learning algorithms shows a small advantage in these environments.



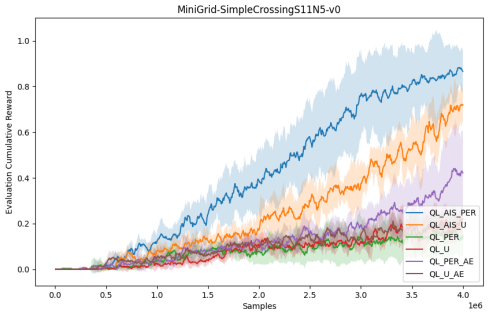
(a) SimpleCrossingS9N1



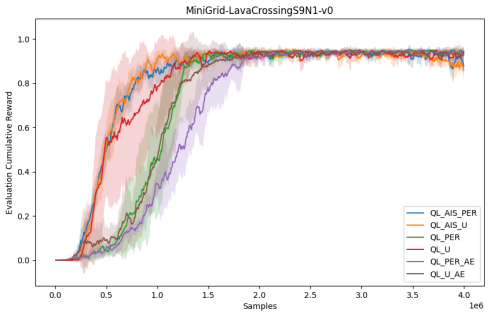
(b) SimpleCrossingS9N2



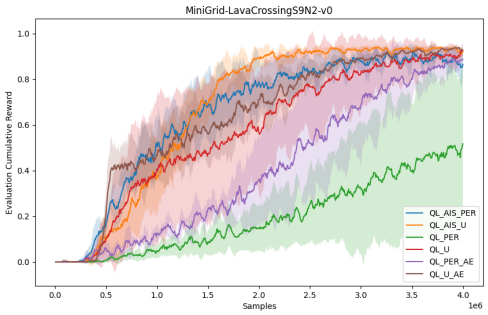
(c) SimpleCrossingS9N3



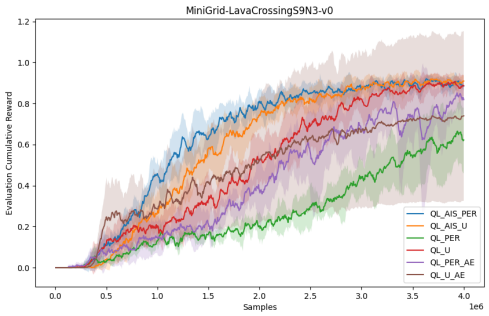
(d) SimpleCrossingS11N5



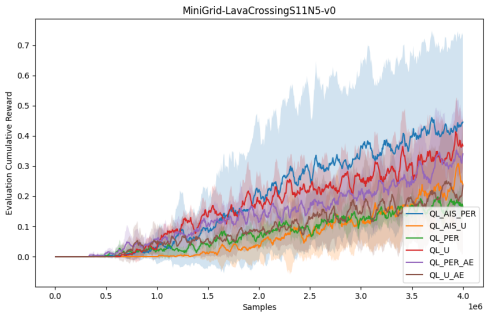
(e) LavaCrossingS9N1



(f) LavaCrossingS9N2



(g) LavaCrossingS9N3



(h) LavaCrossingS11N5

Fig. 4.4 Performance plots for the eight crossing environments comparing the six algorithms.

4.2.2 Key Corridor Environments

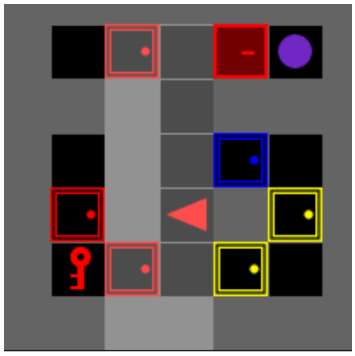
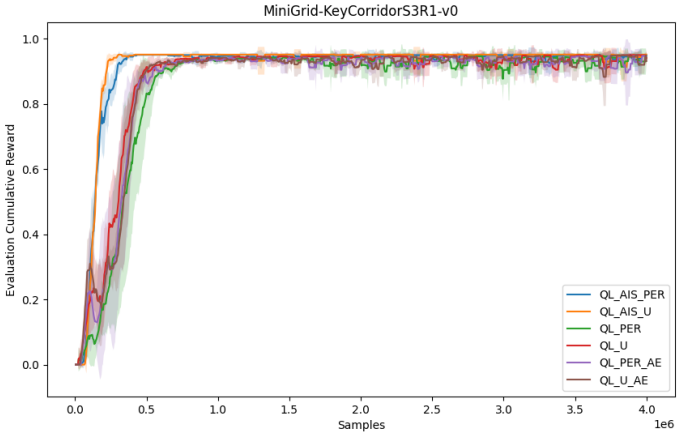


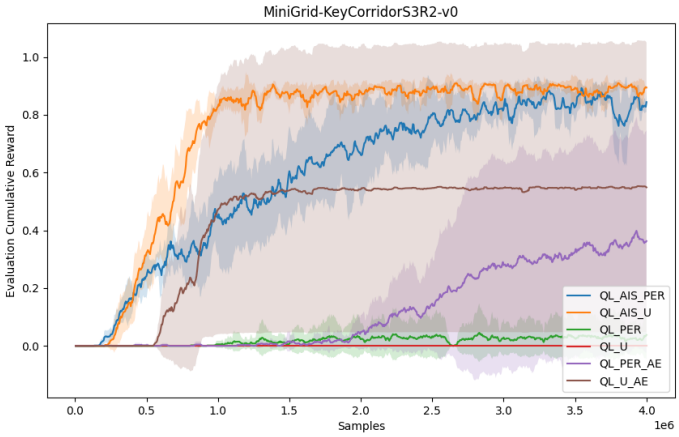
Fig. 4.5 A visualization of a Key Corridor environment.

In Key Corridor environments, the goal is to reach a red ball behind a locked door. The agent has to navigate the environment and find the key to the locked door. The environment consists of multiple hallways all including a number of closed but unlocked doors which the agent has to learn to open. The agent needs to be able to successfully navigate these hallways and find the key. The sequence of tasks required to successfully solve these environment makes them significantly harder than the previous Crossing environments. We will compare the six algorithms on three environments in this group. The environments are identified by KeyCorridorSnRm with n being the size of the $n \times n$ room and m being the number of columns on the grid. The Q-learning algorithms are only able to solve the smallest environment and are generally incapable of solving the bigger environments. Prioritized experience replay does not provide any benefits in these environments to either the model-augmented or the recurrent Q-learning algorithms. The performance of the six algorithms can be seen in 4.6. Using a pretrained encoder with vanilla Q-learning provides a big advantage in KeyCorridorS3R2 but KeyCorridorS3R3 is generally unsolvable by all the Q-learning variants. We see that with harder environments, the model-augmented

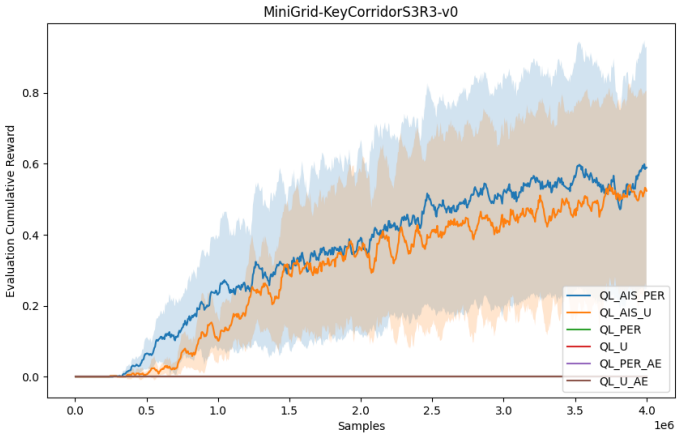
approach shows a bigger advantage.



(a) KeyCorridorS3R1



(b) KeyCorridorS3R2



(c) KeyCorridorS3R3

Fig. 4.6 Performance plots for the three Key Corridor environments comparing the six algorithms.

4.2.3 Door Key Environments

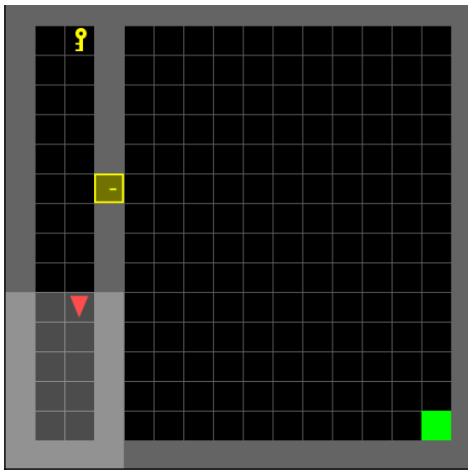
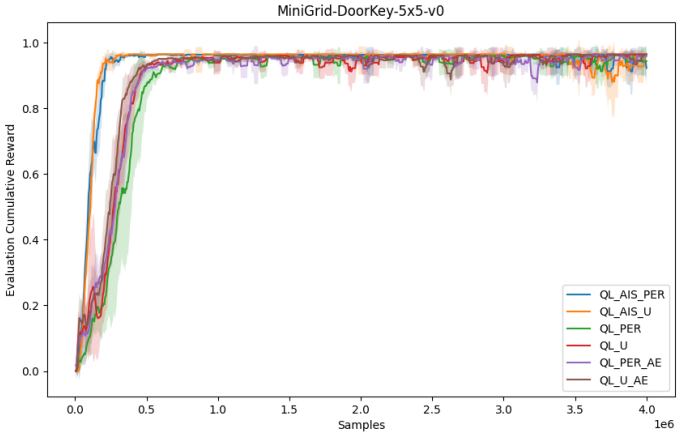
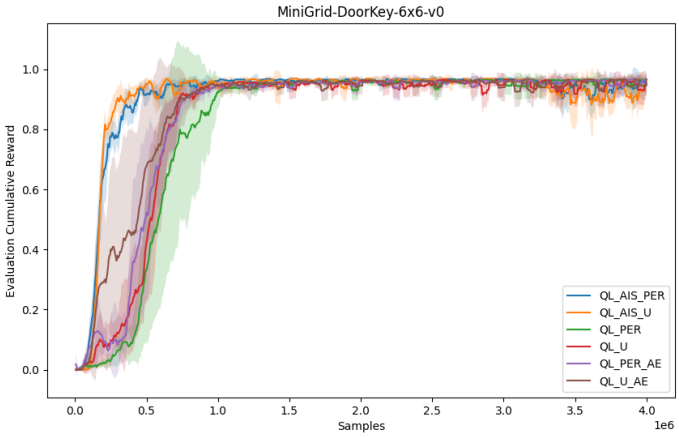


Fig. 4.7 A visualization of the Door Key environment.

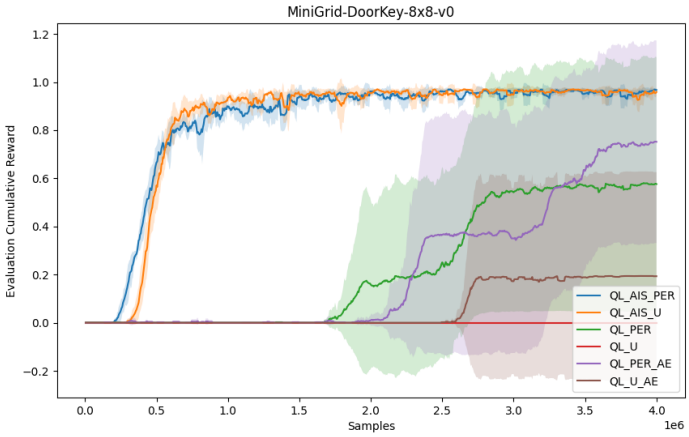
The Door Key environments consist of two rooms. First, the agent needs to navigate the first room and find a key. The key then can be used to open a locked door to the second room. After entering the second room, the agent has to navigate it to find a goal and reach that goal. There are three variants of this environment with their suffixes indicating the combined size of the two rooms. The problem becomes significantly harder as the room sizes become larger and the largest variant of this environment is very difficult to solve with vanilla RL methods. In all environments, the model-augmented methods show a clear advantage. The performance of the six approaches can be seen in 4.8.



(a) DoorKey-5x5



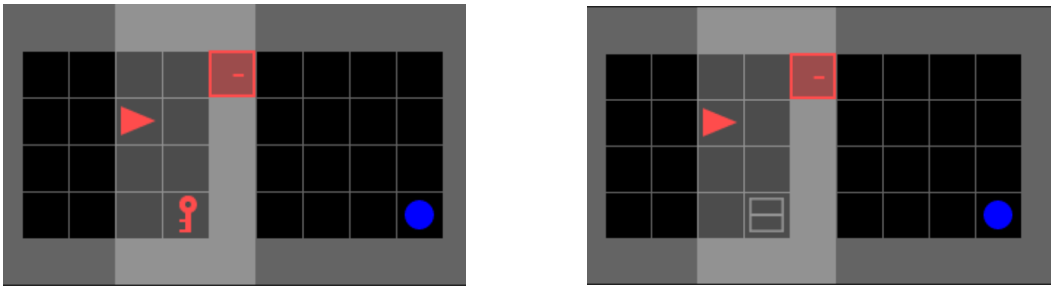
(b) DoorKey-6x6



(c) DoorKey-8x8

Fig. 4.8 Performance plots for the three Door Key environments comparing the six algorithms.

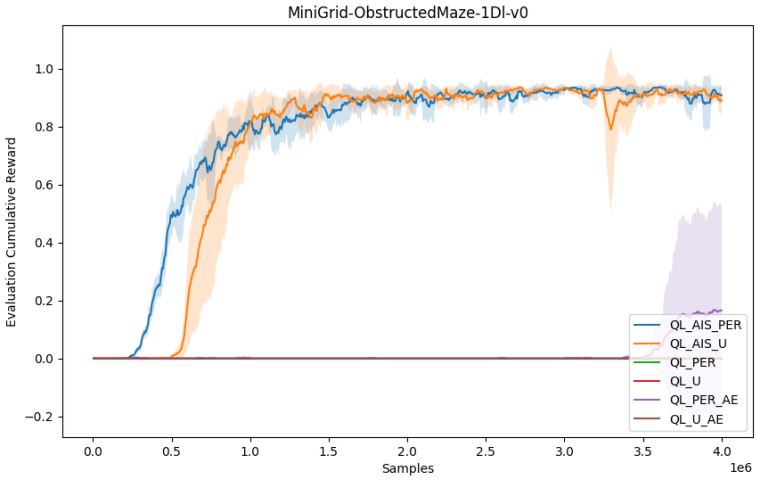
4.2.4 Obstructed Maze



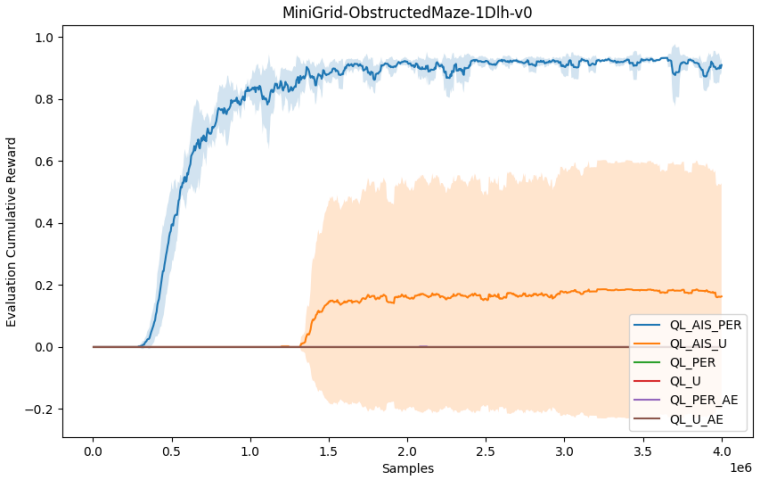
(a) ObstructedMaze-1Dl (b) ObstructedMaze-1Dlh

Fig. 4.9 A visualization of the two variants of Obstructed Maze environment used in this section. 4.9b includes a hidden box which includes the key to the door.

In this family of environments, the agent has to find the keys to a locked door, open the door and reach a goal. In the "h" variant, the key is hidden in a box. Because of their extremely sparse rewards, these environments are generally considered one of the most difficult environments in the MiniGrid family. We see a clear advantage to using the proposed model-augmented Q-learning approach and also using PER in these two environments. None of the variants of the recurrent Q-learning algorithm perform well in these environments. The performance of the six approaches can be seen in 4.10.



(a) ObstructedMaze-1Dl



(b) ObstructedMaze-1Dlh

Fig. 4.10 Performance plots for the two Obstructed Maze environments comparing the six algorithms.

4.2.5 Red Blue doors environments

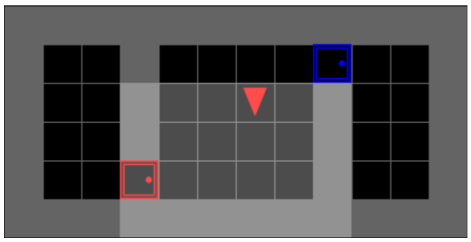
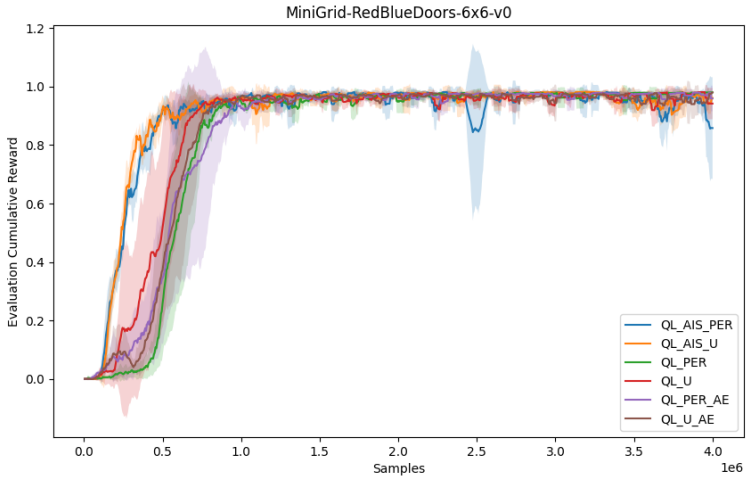
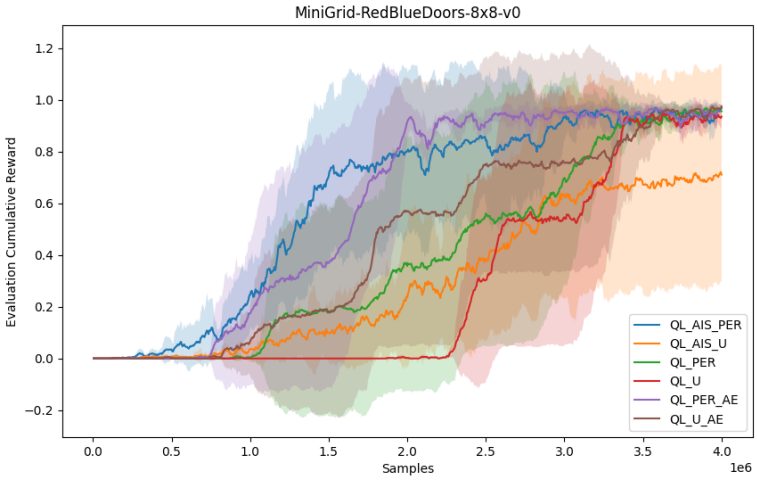


Fig. 4.11 A visualization of the Red Blue door environment.

In these environments, the agent has to first open a red door and next open a blue door. The position of the doors randomly changes at each episode and the two variants have different room sizes. We see a clear advantage in using the model-augmented Q-learning approach in the smaller variant but the gap is smaller in the bigger variant. Also, using a pretrained encoder with recurrent Q-learning provides a big boost to performance in the bigger environment. The performance of the six approaches can be seen in 4.12.



(a) RedBlueDoors-6x6



(b) RedBlueDoors-8x8

Fig. 4.12 Performance plots for the two Red Blue door environments comparing the six algorithms.

4.2.6 Multi-Room environments

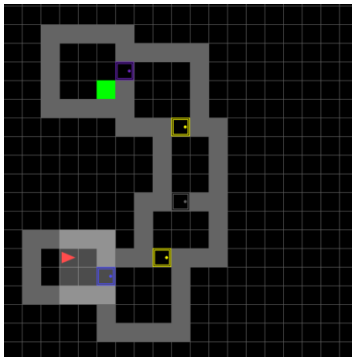
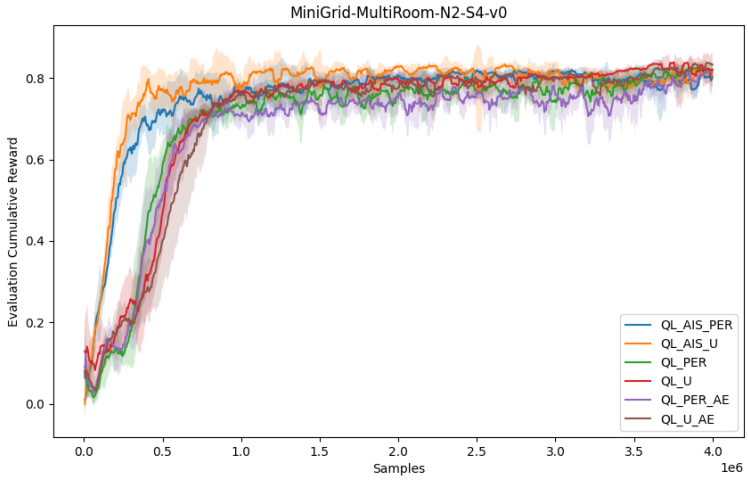
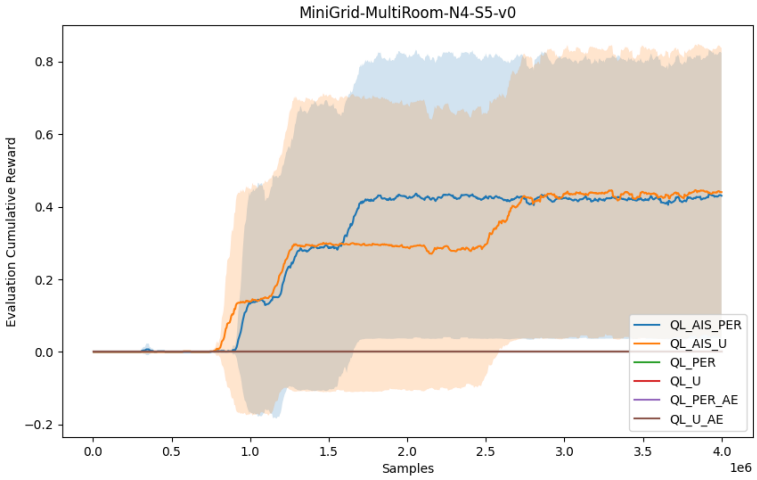


Fig. 4.13 A visualization of the Multi-Room environment.

In these two environments, the agent has to navigate a number of rooms and find the final goal. Exiting each room requires finding and opening a door. The different variants have different number of rooms and the general layout of the environment changes with each episode. These environments also feature extremely sparse rewards. The model-augmented approaches show a clear advantage in both environments and they are the only algorithms capable of solving the task in the bigger environment. Using a pretrained encoder does not provide any tangible benefits with vanilla Q-learning. The performance of the six approaches can be seen in 4.14.



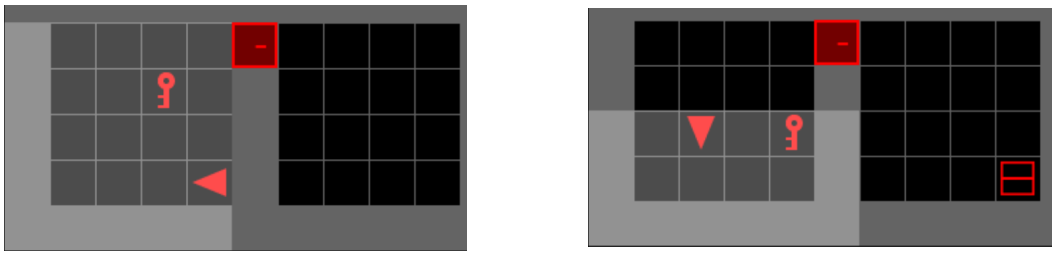
(a) MiniGrid-MultiRoom-N2-S4



(b) MiniGrid-MultiRoom-N4-S5

Fig. 4.14 Performance plots for the two Multi room environments comparing the six algorithms.

4.2.7 Unlock-Pickup environments environments



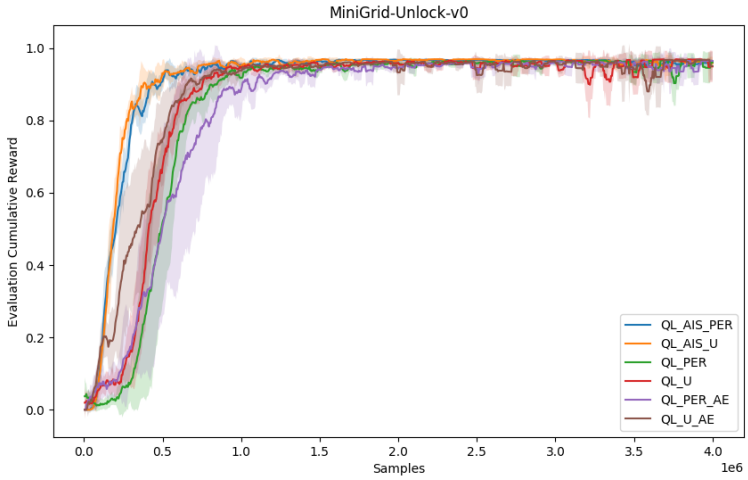
(a) Unlock (b) Unlock-Pickup

Fig. 4.15 A visualization of the two "Unlock" environments uses in this section.

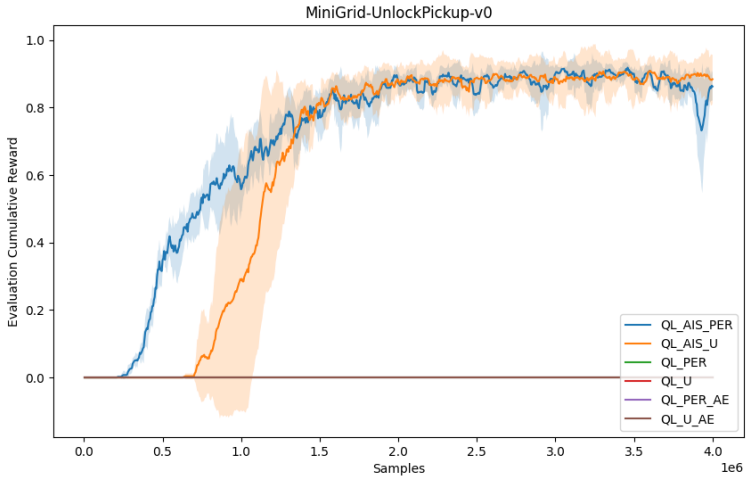
This group consists of two environments each requiring solving an extra task to reach the goal.

- **Unlock:** This is the easiest environment. The goal is to find a key to a locked door and open that door 4.15a.
- **Unlock-Pickup:** In this environment, the agent has to solve all the tasks in the previous environment but after unlocking the door, it also has to pickup a box 4.15b.

The first environment is considered one of the easiest MiniGrid environments and all six algorithms can solve it very quickly. The second is significantly more difficult and only the model-augmented approaches can solve the second. Again, Prioritized experience replay shows a clear advantage in the harder exploration environments. Even in the easiest environments, the model-augmented approaches show a clear advantage over recurrent Q-learning. The performance of these algorithms can be seen in 4.16.



(a) Unlock



(b) UnlockPickup

Fig. 4.16 Performance plots for the two Unlock environments comparing the six algorithms.

4.2.8 Dynamic Obstacles Environments

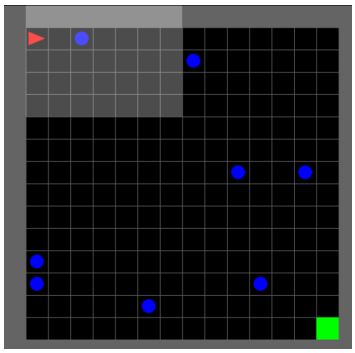
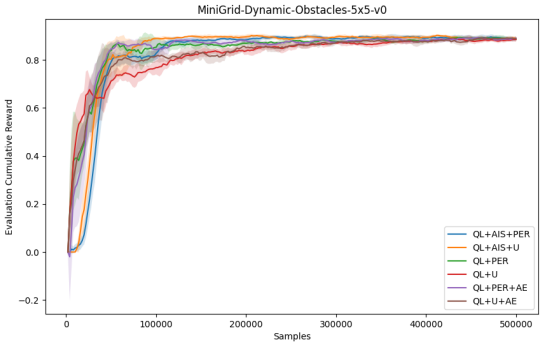
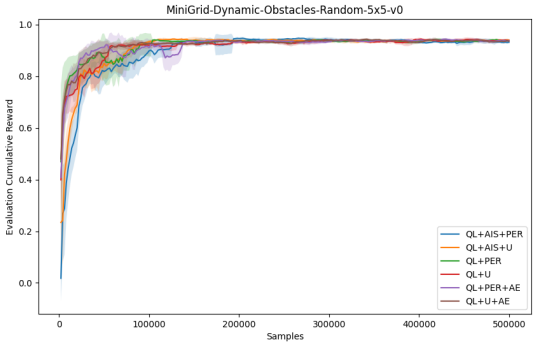


Fig. 4.17 A visualization of the Dynamic Obstacles environment.

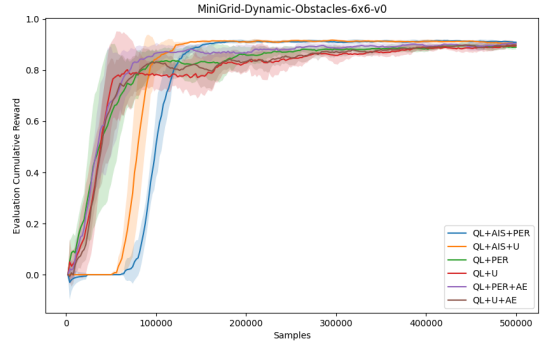
This family of environments are the only MiniGrid environments tested without sparse rewards. The agent is tasked with reaching a randomly placed goal in a room while avoiding collision with moving obstacles. Collision with obstacles results in a -1 reward and the termination of the episode. Because of the abundance of rewards, these environments are solved much more quickly than the previously tested environments. In some instances, the agents learns to perfectly solve the environments in less than 100K steps. These are the only tested environments in which the model-augmented approach does worse than vanilla Q-learning. We believe the time taken to properly learn the AIS model to be the reason for this inferior performance. The performance of the six algorithms can be seen in 4.18. The empirical results on these environments alongside the previous Minigrid environments suggests our proposed model-augmented method to be more useful in sparse reward settings.



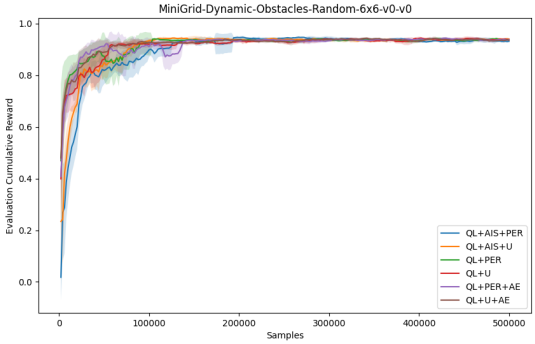
(a) Dynamic-Obstacles-5x5



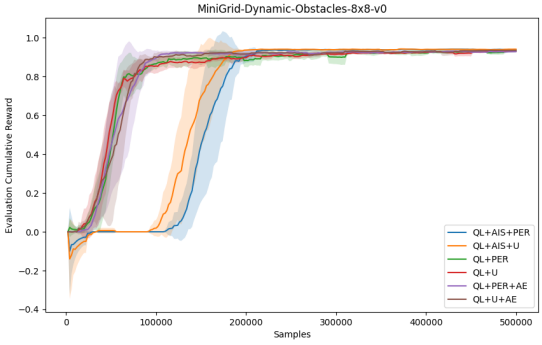
(b) Dynamic-Obstacles-Random-5x5



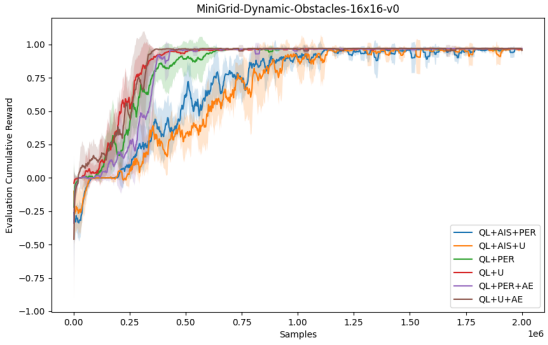
(c) Dynamic-Obstacles-6x6



(d) Dynamic-Obstacles-Random-6x6



(e) Dynamic-Obstacles-8x8



(f) Dynamic-Obstacles-16x16

Fig. 4.18 Performance plots for the six Dynamic Obstacles environments comparing the six algorithms.

Chapter 5

Conclusion and Future works

Throughout this thesis, we proposed a framework to augment standard R2D2 style recurrent Q-learning with a learned transition and reward model. The purpose of learning this model was to obtain state representations that could improve Q-learning performance. We showed that AIS could improve the performance of recurrent Q-learning. Our proposed model-augmented Q-learning algorithm consistently outperformed the vanilla recurrent Q-learning method across most tested environments. Furthermore, we demonstrated that model-augmented Q-learning could also work with prioritized sampling and benefit from it in more complex problems. Many model-based methods involve separate loops (and sometimes separate datasets) for model training and reinforcement learning, but our method is very simple as both model training and Q-learning are done in the same loop and on the same batch of samples. We also do not see significant differences in computation complexity between the model-augmented and vanilla recurrent Q-learning methods. We also demonstrated that the boost in performance provided by model-augmented Q-learning is more significant in sparse-reward environments. The applicability of these methods in hard exploration RL

problems could be a fruitful future avenue for research.

We can build upon our proposed method in multiple ways. In the current implementation, the learned transition models are fully trained but only used for their hidden state representations. These trained transition models could be better utilized by allowing for synthetic data generation. This can be through a Dyna style [37] expansion of the replay buffer with hallucinated trajectories. It has been shown that synthetic data generation by a learned model can improve the data efficiency of model-free methods [38; 60]. We could further improve the performance of multi-step Q-learning by allowing for synthetic trajectory generation for bootstrapping [38]. Furthermore, it was shown [61] that the model loss can be a helpful signal for designating the priorities of each sample in prioritized experience replay. Using the model loss as an auxiliary component of sample priorities [61] was shown to boost performance of model-free methods in fully observable environments. We believe that such an approach could be applied to our algorithm and be potentially beneficial.

Appendix A

Implementation details

Here, we will provide implementation details for all the algorithms used in the experiments section. First, we will detail the hyperparameters that are shared among all the environments. The discount factor γ is set to 0.99 for all environments. The learning rate for all components (for both the Q-learning and model-augmented Q-learning algorithms) is 0.001 and the ADAM optimizer [62] is used as the Stochastic Gradient Descent (SGD) based optimizer. Please note that the AIS components ($\hat{\sigma}$, $\hat{r}$ and $\hat{P}_y$) have a seperate optimizer from the $\hat{Q}$ for the model-augmented approach. The AIS hidden state size ($d_{\hat{Z}}$) is also set to 128 for all experiments. All experiments are run on 5 seeds. Experience data is split to subsequences of length 10 and are stored in the buffer similar to the R2D2 approach [9]. The Burn-in length which is the maximum length of the preceding history for each stored subsequence is set to 50 for all experiments. Batch sizes are the number of subsequences sampled at each training step.

In all tested algorithms and their variants, epsilon-greedy is used as the exploration strategy. At the start of the training procedure, the ϵ value is set to 1.0 and it

is exponentially decayed until it reaches 0.05 by the end of the training procedure. The rate of exponential decay is proportional to the number of steps taken in the environment during training so in experiments that involve more interaction with the environment, more exploration is conducted. During evaluation, the agent chooses actions greedily with respect to $\hat{Q}$ and all plots are generated from 10 episodes of testing which is done in regular intervals between the training steps. For all experiments, multi-step updates are done for training the Q-function with the multi-step length being equal to 5.

For prioritized experience replay (PER), we follow the standard PER implementation [22] and use Min-Max heaps [63] for storing priorities and doing weighted sampling. The PER involves two hyperparameters as was discussed before. The β hyperparameter is set to 0.4 at the start of training and it is linearly increased until it reaches 1 at the end of the training process. The α is set to 0.6.

The following are some environment related parameters:

Environment	No. of actions n_A	No. of obs. n_O	Batch Size
Rock Sampling	8	3	64
Drone Surveillance	5	10	64
MiniGrid	7	$7 \times 7 \times 3$	256

As was discussed, the model-augmented approach and the different variants of the recurrent Q-learning algorithm all have different components which are all modelled with neural networks using linear layers and non-linear functions in between. During our experimentation, we have found the Exponential Linear Units (ELUs) [64] to

peform quite well and we use this nonlinear layer alongside Rectified Linear Units (ReLUs) for nonlinear layers. For the recurrent component, we are using an LSTM [7] function. Linear layers with input sizes n and output sizes m are denoted by Linear(n, m) and an LSTM cell with input of size n and a hidden vector of size m is denoted by LSTM(n, m).

For the MiniGrid environments, we are using a simple autoencoder [57] to get more compact representations from the relatively high-dimensional observations. The encoders are trained alongside a decoder for 100 epochs on 4M samples of observations gathered by a random agent for each environment. The encoder weights are frozen during the reinforcement learning phase. We use these encoders in the model-augmented Q-learning approach and the Q-learning variant with autoencoders. The encoder and decoder architectures are as follows:

Encoder	*Decoder*
Linear $(147, 96)$	Linear $(64, 96)$
ReLU	ReLU
Linear $(96, 64)$	Linear $(96, 147)$
	Tanh

Now, we will discuss model architectures used for the different components of the model-augmented Q-learning approach (QL with AIS). This algorithm uses four different components: the recurrent history compression fuction $\hat{\sigma}$, the reward prediction function $\hat{r}$, the observation prediction function $\hat{P}^y$ and the action-value function $\hat{Q}$. For both the discrete observation environments and the MiniGrid environments, all the components except for the $\hat{P}^y$ are the same. For the MiniGrid environments,

the observations which are outputs of the pretrained encoder and for the discrete observation environments we use the raw one-hot encoded observations. The following are the model architectures for the $\hat{\sigma}$, the $\hat{r}$ and the $\hat{Q}$:

$\hat{\sigma}$	$\hat{r}$	$\hat{Q}$
$\text{Linear}\,(n_O + n_A + 1, d_{\hat{Z}})$	$\text{Linear}\left(n_A + d_{\hat{Z}}, \frac{1}{2}d_{\hat{Z}}\right)$	$\text{Linear}\,(d_{\hat{Z}}, d_{\hat{Z}})$
ELU	ELU	ELU
$\text{LSTM}\,(d_{\hat{Z}}, d_{\hat{Z}})$	$\text{Linear}\left(\frac{1}{2}d_{\hat{Z}}, 1\right)$	$\text{Linear}\,(d_{\hat{Z}}, d_{\hat{Z}})$
		ELU
		$\text{Linear}\,(d_{\hat{Z}}, d_{\hat{Z}})$
		ELU
		$\text{Linear}\,(d_{\hat{Z}}, n_A)$

For the obsevation prediction function $\hat{P}^y$, we use a slightly different architecture between the MiniGrid experiments and the discrete observation space environments. In the MiniGrid experiments, $\hat{P}^y$ has to output continuous variables. For the discrete observation space environments, we use a final softmax layer so that class probabilities are outputted. We also add another observation category representing the final state so that $\hat{P}^y$ is forced to also predict the end of an episode. This provided improvements in performance in our experiments. The model architectures for the two $\hat{P}^y$ is as follows:

$\hat{P}^y(MiniGrid)$	$\hat{P}^y(Discrete)$
Linear $\left(n_A + d_{\hat{Z}}, \frac{1}{2}d_{\hat{Z}}\right)$	Linear $\left(n_A + d_{\hat{Z}}, \frac{1}{2}d_{\hat{Z}}\right)$
ELU	ELU
Linear $\left(\frac{1}{2}d_{\hat{Z}}, n_O\right)$	Linear $\left(\frac{1}{2}d_{\hat{Z}}, n_O + 1\right)$
	Softmax

Next, we will discuss the model architectures used for the recurrent Q-learning algorithm for the discrete observation space environments and the recurrent Q-learning variant which uses pretrained encoder inputs for the MiniGrid cases. Please note that we have divided the architecture into two components: the recurrent $\hat{\sigma}$ and the Q-function $\hat{Q}$. This is done for simplicity of implementation but both are trained using the Q-learning loss.

$\hat{\sigma}$	$\hat{Q}$
Linear $(n_O + n_A + 1, d_{\hat{Z}})$	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
ELU	ELU
LSTM $(d_{\hat{Z}}, d_{\hat{Z}})$	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
	ELU
	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
	ELU
	Linear $(d_{\hat{Z}}, n_A)$

The following are the model architectures used for the different components of the recurrent Q-learning algorithm which receives raw observations in the MiniGrid case. The $\hat{\sigma}$ component has an extra layer to be better able to process high dimensional

observations.

$\hat{\sigma}$	$\hat{Q}$
Linear $(n_O + n_A + 1, d_{\hat{Z}})$	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
ELU	ELU
Linear $(d_{\hat{Z}}, d_{\hat{Z}})$	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
ELU	ELU
LSTM $(d_{\hat{Z}}, d_{\hat{Z}})$	Linear $(d_{\hat{Z}}, d_{\hat{Z}})$
	ELU
	Linear $(d_{\hat{Z}}, n_A)$

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2015.

[5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, Nov 2019.

[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[8] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[9] S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos, "Recurrent experience replay in distributed reinforcement learning," in *International Conference on Learning Representations*, 2019.

[10] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," 2015.

[11] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," 2019.

[12] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," 2015.

[13] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," 2018.

[14] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[15] J. Subramanian, A. Sinha, R. Seraj, and A. Mahajan, "Approximate information state for approximate planning and reinforcement learning in partially observed systems," 2020.

[16] M. Chevalier-Boisvert, L. Willems, and S. Pal, "Minimalistic gridworld environment for gymnasium." https://github.com/Farama-Foundation/MiniGrid, 2018.

[17] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," 2016.

[18] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "exploration: A study of count-based exploration for deep reinforcement learning," 2016.

[19] J. Schmidhuber, "A possibility for implementing curiosity and boredom in model-building neural controllers," in *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, (Cambridge, MA, USA), p. 222–227, MIT Press, 1991.

[20] R. Zhao and V. Tresp, "Curiosity-driven experience prioritization via density estimation," 2019.

[21] G. Ostrovski, M. G. Bellemare, A. v. d. Oord, and R. Munos, "Count-based exploration with neural density models," 2017.

[22] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015.

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992.

[24] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," 2017.

[25] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," 2017.

[26] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019.

[27] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[28] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[29] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 387–395, PMLR, 22–24 Jun 2014.

[30] H. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[31] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[32] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," 2012.

[33] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[34] Z. I. Botev, D. P. Kroese, R. Y. Rubinstein, and P. L'Ecuyer, "Chapter 3 - the cross-entropy method for optimization," in *Handbook of Statistics* (C. Rao and V. Govindaraju, eds.), vol. 31 of *Handbook of Statistics*, pp. 35–59, Elsevier, 2013.

[35] S. Mannor, R. Rubinstein, and Y. Gat, "The cross entropy method for fast policy search," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, p. 512–519, AAAI Press, 2003.

[36] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," 2018.

[37] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *SIGART Bull.*, vol. 2, p. 160–163, jul 1991.

[38] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine, "Model-based value estimation for efficient model-free reinforcement learning," 2018.

[39] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[40] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, June 2013.

[41] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, and S. Petersen, "Deepmind lab," 2016.

[42] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games* (H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, eds.), (Berlin, Heidelberg), pp. 72–83, Springer Berlin Heidelberg, 2007.

[43] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006* (J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds.), (Berlin, Heidelberg), pp. 282–293, Springer Berlin Heidelberg, 2006.

[44] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller, "Deepmind control suite," 2018.

[45] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013.

[46] K. Åström, "Optimal control of markov processes with incomplete state information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, 1965.

[47] W. Rudin, *Principles of mathematical analysis*. McGraw-Hill New York, 3d ed. ed., 1976.

[48] G. Shorack, *Probability for Statisticians*. 01 2017.

[49] B. Sriperumbudur, A. Gretton, K. Fukumizu, G. Lanckriet, and B. Schölkopf, "Injective hilbert space embeddings of probability measures.," pp. 111–122, 01 2008.

[50] A. Berlinet and C. Thomas-Agnan, *Reproducing Kernel Hilbert Space in Probability and Statistics*. 01 2004.

[51] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *Journal of Machine Learning Research*, vol. 13, no. 25, pp. 723–773, 2012.

[52] A. Gretton, K. Borgwardt, M. Rasch, B. Schölkopf, and A. Smola, "A kernel method for the two-sample-problem.," pp. 513–520, 01 2006.

[53] B. K. Sriperumbudur, K. Fukumizu, A. Gretton, B. Schoelkopf, and G. R. G. Lanckriet, "On the empirical estimation of integral probability metrics," *Electronic Journal of Statistics*, vol. 6, pp. 1550–1599, 2012.

[54] I. Csiszár and J. Körner, *Information Theory*. Cambridge University Press, June 2011.

[55] D. J. C. MacKay, *Information Theory, Inference amp; Learning Algorithms*. USA: Cambridge University Press, 2002.

[56] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 1999.

[57] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *Aiche Journal*, vol. 37, pp. 233–243, 1991.

[58] T. Smith and R. Simmons, "Heuristic search value iteration for pomdps," in *Proceedings of 20th Conference on Uncertainty in Artificial Intelligence (UAI '04)*, pp. 520 – 527, July 2004.

[59] M. Svoreňová, M. Chmelík, K. Leahy, H. F. Eniser, K. Chatterjee, I. Černá, and C. Belta, "Temporal logic motion planning using pomdps with parity objectives: Case study paper," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, (New York, NY, USA), p. 233–238, Association for Computing Machinery, 2015.

[60] M. Janner, J. Fu, M. Zhang, and S. Levine, "When to trust your model: Model-based policy optimization," 2019.

[61] Y. Oh, J. Shin, E. Yang, and S. J. Hwang, "Model-augmented prioritized experience replay," in *International Conference on Learning Representations*, 2022.

[62] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[63] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Commun. ACM*, vol. 29, pp. 996–1000, 1986.

[64] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," 2015.