university of groningen / faculty of science and engineering / artificial intelligence

Accelerating Model Based Reinforcement Learning Using GPU Through Parallelization of Dyna-Q Architecture

Bachelor's Project Thesis

Rareș Ștefan Stoian, s4775309, s.rares@student.rug.nl,
Supervisors: R. (Rafael) Fernandes Cunha

Abstract: In this paper, we propose Parallelized Dyna Q-Network (PDQN), a fully online, GPU-accelerated reinforcement learning algorithm that integrates model-based planning with the recently introduced Parallelized Q-Network (PQN). By employing learned world models for short-horizon planning, PDQN seeks to further accelerate convergence while preserving the simplicity of fully online temporal-difference learning. Empirical results on multiple MinAtar environments show that PDQN achieves performance on par with PQN. Our analysis also reveals that selecting appropriate values for planning delay and planning steps is crucial for the performance to be at least on par with the baseline.

1 Introduction

Artificial intelligence has seen an astronomical rise in popularity in recent years due to advancements in machine learning algorithms. This surge in interest has been credited mostly to the development of deep learning [18], which was quickly adopted by both research and industry as its potential was rapidly recognized.

Deep Reinforcement Learning (DRL) has demonstrated amazing performance in environments characterized by large state and action spaces, enabling agents to generalize effectively across diverse scenarios. This capability has been essential in advancing both online and offline learning frameworks. Notably, algorithms such as Deep Q-Networks (DQN) [25] and Proximal Policy Optimization (PPO) [31] have emerged as prominent methods within this domain. However, these advancements are accompanied by increased computational demands, potentially widening the gap in research accessibility due to the need for substantial computational resources [26].

Offline learning, although more successful thus far, is not without its problems [9, 19]. One issue attributed to offline learning is the instability of experience replay when its size becomes too large, causing algorithms to converge slowly [21]. Despite these persistent issues, offline learning methods remain the preferred choice for most because they are more data efficient, improving sample efficiency by allowing agents to learn effectively from pre-trained policies on unseen data [1] or by bootstrapping from existing data without needing to interact with the environment [32].

Despite recent improvements in the field of DQNs [40], these methods are temporal-difference (TD) methods, and some are plagued by certain problems, such as instability and prone to overestimation caused by at least one of the "deadly triad" components: function approximation, bootstrapping, and off-policy learning [39, 40].

A recent study by Gallici et al. [10], which serves as the basis for our work, aims to reintroduce online learning and temporal-difference (TD) methods into the spotlight. Specifically, the authors propose the Parallelized Q-Network (PQN) algorithm, a parallelized and fully online variant of Q-learning designed explicitly for GPU architectures. By eliminating the replay buffer and target network, which are core components in traditional Deep Q-Networks (DQN), PQN significantly reduces the computational and memory overhead. Additionally, PQN's approach of simultaneously running multiple vectorized environments fully on the GPU avoids the traditional communication bottleneck between CPU-based environments and GPU-based learners, leading to markedly faster convergence speeds. In addition to the above changes, PQN uses regularization methods such as Batch Normalization [14] and Layer Normalization [2] for better stability.

Model-based Reinforcement Learning (RL) is a class of algorithms that centers around building a *model* of the environment, allowing agents to predict future states and rewards without needing to interact directly with the environment itself. One of the earliest model-based algorithms was introduced by Sutton et al. [33], who proposed not only the Dyna Q-learning algorithm—an adaptation of the classical Q-learning—but also the widely influential Dyna architecture. Since agents can utilize these world models to learn independently from their real interactions, convergence can often be achieved with significantly lower computational costs and, in certain scenarios, at faster rates compared to purely model-free approaches [20, 23, 24, 29]. Motivated by the potential computational efficiency and accelerated convergence offered by model-based methods like Dyna Q-learning, alongside recent advancements demonstrated by PQN, we pose the following question: To what extent does the improved convergence speed provided by Dyna Q-learning affect Parallelized Temporal-Difference learning in GPU environments, and does this justify its implementation?

In this paper, we adapt the aforementioned PQN algorithm to incorporate world model learning methods by following the Dyna architecture. Our contributions are as follows: (A) We create the Parallelized Dyna Q-network (PDQN) algorithm, and (B) we investigate whether there is a significant improvement in convergence speed without sacrificing the algorithm's performance on the given tasks.

To our knowledge, the implementation of parallelized world model methods in the research field is somewhat unexplored, with only one paper implementing Dyna Q-learning where the parallelization occurs between multiple agents within the same environment [36]. Contributions related to Dyna Q-learning using deep RL methods are more prominent in fields such as robotics [27], recommender systems [45], and task-completion dialogue agents [28], or any domain where training requires extensive interactions with the environment that are either costly or time-consuming. The lack of research on this topic and evidence that Dyna Q-learning greatly increases learning speed further motivate us to document our findings.

For simplicity, we maintain the core concept of the PQN algorithm by using Q-learning as a foundational TD method with function approximation, and augment it by introducing a world model component, thus creating the PDQN algorithm. We have chosen MinAtar [44], a vectorized version of the Atari Learning Environment (ALE) [4], to enable straightforward analysis of performance on a popular benchmark within a fully end-to-end GPU setup.

2 Theoretical Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of Machine Learning (ML) an agent is employed to interact with an environment by performing actions and observing the resulting rewards through trial-and-error exploration. Over time, the agent's objective is to maximize cumulative reward, refining a *policy* that governs its behavior. By adjusting its reward-based policy, the agent effectively learns the most suitable actions to achieve a given goal within the environment [34].

A Markov Decision Process (MDP) [5] is a mathematical framework commonly used to formalize the interaction between an agent and its environment in Reinforcement Learning (RL). Formally, an MDP is defined by the 5-tuple (S, A, P, R, γ), where:

1. S: The set of possible **states** the agent can occupy.

2. A: The set of possible **actions** the agent can perform.

3. P: The **transition probability function** $P(s_{t+1}|s_t, a_t)$, which specifies the probability of moving from one state s_t at time t to s_{t+1} when taking action a_t, where $s_t, s_{t+1} \in S$ and $a_t \in A$.

4. R: The **reward function**, $R(s_t, a_t)$, which returns a numerical reward r_t for each state transition.

5. γ: the **discount factor**, with $0 \leq \gamma \leq 1$, and determines the present value of future rewards and balances immediate versus long-term gains.

The objective of RL is to maximize the cumulative discounted reward (or **return**), formally defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

Consequently, the central goal becomes finding an optimal policy π^* that maximizes the expected return starting from every state s. According to Sutton and Barto [34], this optimal policy satisfies:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[G_t \mid S_t = s], \quad \forall s \in S. \tag{2.2}$$

Using G_t, we derive the state-value function $v_\pi(s)$, representing the expected return from state s under policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]. \tag{2.3}$$

Similarly, we define the action-value function $q_\pi(s, a)$ as the expected return when starting from state s, taking action a, and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.4}$$

Given these definitions, the optimal policy explicitly selects actions that maximize the action-value function:

$$\pi^*(s) = \arg\max_{a \in A} q_{\pi^*}(s, a), \quad \forall s \in S. \tag{2.5}$$

This optimal policy ensures the highest possible expected return across all states and actions, serving as the theoretical foundation for various RL algorithms, including PQN and Dyna-Q.

2.1.1 Temporal Difference (TD) Methods

Temporal Difference (TD) learning is a core class of methods in reinforcement learning that update value estimates based on the difference between successive predictions. TD methods combine the strengths of Monte Carlo methods (learning directly from experience) and Dynamic Programming (bootstrapping from current estimates), allowing agents to learn directly from raw experience without requiring a model of the environment.

For value prediction under a fixed policy π, the TD target for the state-value function is defined as:

$$G_t^{(1)} = r_{t+1} + \gamma v_\pi(s_{t+1}), \tag{2.6}$$

and the corresponding TD error becomes:

$$\delta_t = r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t). \tag{2.7}$$

The update rule for the state-value function under a TD(0) scheme is:

$$v_\pi(s_t) \leftarrow v_\pi(s_t) + \alpha \delta_t, \tag{2.8}$$

where α is the step-size (learning rate).

For control tasks, the action-value function $q_\pi(s, a)$ is used. In particular, Q-learning—a widely used off-policy TD control method—aims to learn the optimal action-value function $Q^*(s, a)$. Its update rule is given by:

$$\begin{aligned} q(s_t, a_t) \leftarrow q(s_t, a_t) + \\ \alpha \left[r_{t+1} + \gamma \max_{a'} q(s_{t+1}, a') - q(s_t, a_t) \right]. \end{aligned} \tag{2.9}$$

This formulation bootstraps from the current estimate of the action-value function and enables the agent to approximate $q_{\pi^*}(s, a)$, leading toward the optimal policy π^*. TD methods such as Q-learning form the basis for many modern deep reinforcement learning algorithms, including PQN and Dyna-Q, where tabular value functions are replaced with function approximators such as neural networks.

2.2 Parallelized Q-Network (PQN)

The PQN algorithm mentioned above [10] aims to reintroduce TD methods using end-to-end GPU setup to accelerate learning. The findings expand upon RL through hardware, as is one of the first papers to explore such methods.

The paper claims state-of-the-art performance in terms of training speed (i.e, convergence speed), “*up to 50%*“ in certain environments compared to their DQN counterpart, without sacrificing the overall performance. They achieve this through the use of `JAX` [7, 30] library in Python, a low-level framework for numerical transformations. It uses just-in-time (or jit) compilation and it allows vectorization of environments (each running on different GPU cores, thus running in parallel).

2.2.1 PQN Algorithm

The theoretical implications of vectorization of the environments allowed the authors to adapt DQN and simplify it. With the use of BatchNorm [14] and LayerNorm [2], and through this vectorization, multiple environments run simultaneously on separate GPU cores, allowing parallel sampling of experiences across different episodes. This parallelism removes the need for traditional stabilizing components, such as a replay buffer or a target network, making the learning process fully online.

This approach relies on directly estimating the optimal action-value function $Q^*(s,a)$ which is derived from equation 2.4 and it is defined as follows:

$$Q^*(s,a) = \max_\pi q_\pi(s,a) = \\ \max_\pi \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a\right]. \tag{2.10}$$

The PQN algorithm iteratively approximates this optimal Q^* through the loss function gradient, derived from the temporal difference (TD) learning framework:

$$\theta_{x+1} = \theta_x + \alpha_x \big(r_t + \gamma \max_{a_{t+1}} Q_{\theta_x}(s_{t+1}, a_{t+1}) \\ - Q_{\theta_x}(s_t, a_t)\big) \nabla_\theta Q_{\theta_x}(s_t, a_t), \tag{2.11}$$

Where θ represents the parameters of the Q-network and α is the learning rate. You can see the pseudocode for PQN in Appendix A, specifically in Algorithm A.1.

For simplicity and readability, we drop the explicit time-step indices for states and actions in the following equations and pseudocodes (i.e., s_t is simplified to s, s_{t+1} to s', and similarly for actions and rewards).

2.3 Dyna Q-learning

First introduced by Sutton in 1991 [33], Dyna Q-learning expands upon the already existing Q-learning [42] with the addition of the model component and planning.

In addition to updating the Q-values based on the real transitions (s, a, r, s'), Dyna Q-learning learns an empirical model of the "world" or, more formally, a model of the environment's dynamics, usually denoted as $\hat{P}(s'|s,a)$ and $\hat{R}(s,a)$. This gives the agent the ability to "imagine" the next states and actions as a result of its knowledge of the world (i.e., the learned world model) and also based on its current state.

The ingenuity of model-based algorithms does not necessarily come from the world model component, as this alone does not solve any problems. The true advancements come from how an agent uses these predictions of the future states and actions of the environment. After a number of real interactions, the algorithm performs multiple simulated updates by drawing hypothetical transitions from the learned model and updating the Q-function based on these "imagined" events. This is called the planning phase, and it has some advantages and disadvantages. One such advantage is that it substantially improves sample efficiency by allowing the agent to exploit simulated experiences, reducing the need for extensive real-world interactions [16]. Additionally, this approach can accelerate convergence rates and improve learning stability, particularly in environments where real-world data collection is costly or time-consuming [23].

A big disadvantage of the algorithm is that it is not void of the deadly triad mentioned before. By partially relying on planning, the algorithm can steer the agent onto a local minima, or even make it prone to overestimation [15].

The Dyna-Q algorithm, although intricate, is quite simple at its core. As mentioned before, it maintains the core concept of Q-learning, but adds the two extra components: model learning and planning. For pseudocode of the algorithm, see Appendix A, specifically in Algorithm A.2.

3 Methods

By integrating the previously introduced PQN (Section 2.2) with Dyna Q-learning (Section 2.3), we arrive at the Parallelized Dyna Q-Network (PDQN). The central motivation behind this algorithm is to enhance both the convergence speed and

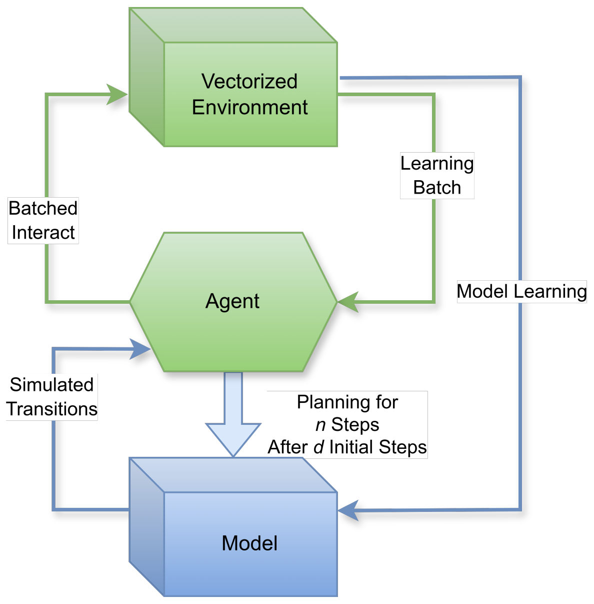


Figure 3.1: Diagram of PDQN: Inner (green) loop represents already existing PQN logic, while the Outer loop (blue) represents the new, PDQN logic, which includes the model, model learning and planning.

performance of PQN by leveraging the Dyna architecture's planning component, which does not rely on additional interactions with the environment to update its Q function [8, 27].

3.1 PDQN

PDQN is thus an expanded version of PQN that incorporates Dyna Q-learning. As depicted in Figure 3.1, the green loop (the inner loop) represents the unaltered PQN framework, maintaining parallelized interactions between the agent and multiple vectorized environments. Meanwhile, the blue loop (the outer loop) introduces the model component, reflecting how the PQN algorithm is combined with the Dyna-Q architecture's planning mechanism.

3.1.1 Algorithm and Innovation

To better understand the inner workings of PDQN, we will begin with its first component: the **model loss function**. This function is central to model-based learning methods, including the approach used here. Immediately after PQN updates its network weights and Q-function (see lines 10 and 11 in Algorithm 3.1), the model $\mathcal{M}_\phi$ is updated by minimizing the following mean-squared error (MSE) inspired loss (line 12 in Algorithm 3.1):

$$\begin{aligned}\mathcal{L}_\mathcal{M} = &\|P^S_\phi(s_i, a_i) - s'_i\|^2 \\ &+ \|P^R_\phi(s_i, a_i) - r_i\|^2 \\ &+ \|P^{\text{done}}_\phi(s_i, a_i) - \text{done}_i\|^2,\end{aligned}$$

where P^S_ϕ, P^R_ϕ, and P^{done}_ϕ denote the learned model components for predicting the next state, the reward, and whether the episode has ended, respectively. By minimizing this squared error for each transition sample $(s_i, a_i, s'_i, r_i, \text{done}_i)$, PDQN refines its internal model without requiring additional interactions with the environment. The implementation of the P^{done}_ϕ was done because we do not have any direct information about the environment, thus predicting the end state helps with the overall performance and stability of the model.

Once the loss $\mathcal{L}_\mathcal{M}$ is computed, the parameters ϕ of the model network are updated via gradient descent (line 13 in Algorithm 3.1):

$$\phi \leftarrow \phi - \beta \nabla_\phi \mathcal{L}_\mathcal{M},$$

where β is the learning rate. Notably, PDQN's *model network* follows the same general structure as PQN's network, ensuring compatibility and efficient parameter updates.

After said updates on the model are done, the planning can finally start. There are two hyperparameters that are specific to the planning phase, namely u_{delay} and n, both of which can be found in Algorithm 3.1 line 15. The first of the two tells the algorithm after how many update steps planning can start, which we also call *planning delay* (or **PD**), whereas the second one, or *planning steps* (or **PS**), tells the algorithm how many "imaginary steps in advance the algorithm can take".

The purpose of this study is to find out whether PDQN performs better in terms of convergence speed and performance than the baseline, PQN. In order to correctly assess the performance of PDQN, we will compare it with PQN, the exact details of which will be described in the upcoming section (3.2).

Algorithm 3.1 (PDQN) (s, a, a', r, target are B-dim vectors)

```
1: $\theta \leftarrow$ initialize $Q$-network parameters
2: $\phi \leftarrow$ initialize $\mathcal{M}$ parameters
3: $s \leftarrow$ initial state $s_0 \sim P_0$
4: set parameters: learning rate $\alpha$ for $Q_\theta$, $\beta$ for $\mathcal{M}_\phi$, discount $\gamma$, exploration rate $\epsilon$, planning steps $n$, planning delay updates $u_{\text{delay}}$
5: initialize environments $B$ in parallel with initial states $s_i$, for each $i \in B$
6: for each episode do
7:   for each step $i$ in $B$ (in parallel) do
8:     $a_i \leftarrow \begin{cases} a_i \sim \text{Unif}(A), & \text{with prob. } \epsilon, \\ \arg\max_{a'} Q_\theta(s_i, a'), & \text{otherwise,} \end{cases}$
9:     $s'_i$, $r_i$, done$_i \leftarrow\sim P(s_i, a_i)$
10:    target$_i \leftarrow r_i + \gamma\, 1(\neg \text{done}_i)\ \max_{a'} Q_\theta(s'_i, a')$
11:    update $\theta$: $\theta \leftarrow \theta - \alpha \nabla_\theta \big\|\text{StopGrad}[\text{target}] - Q_\theta(s_i, a_i)\big\|^2$
12:    update $\mathcal{M}_\phi$: $\mathcal{L}_\mathcal{M} \leftarrow \|P^S_\phi(s_i, a_i) - s'_i\|^2 + \|P^R_\phi(s_i, a_i) - r_i\|^2 + \|P^{\text{done}}_\phi(s_i, a_i) - \text{done}_i\|^2$
13:    $\phi \leftarrow \phi - \beta \nabla_\phi \mathcal{L}_\mathcal{M}$
14:    $s_i \leftarrow s'_i$
15:    if current update step $\geq u_{\text{delay}}$ and $n > 0$ then
16:      for $j = 1$ to $n$ (model-based planning steps) do
17:        $a_j \leftarrow \begin{cases} a_j \sim \text{Unif}(A), & \text{with prob. } \epsilon, \\ \arg\max_{a'} Q_\theta(s_j, a'), & \text{otherwise,} \end{cases}$
18:        $s'_j$, $r_j$, done$_j \leftarrow\sim P_\phi(s_j, a_j)$
19:        target$_j \leftarrow r_j + \gamma\, 1(\neg \text{done}_j)\ \max_{a'} Q_\theta(s'_j, a')$
20:        update $\theta$: $\theta \leftarrow \theta - \alpha \nabla_\theta \big\|\text{StopGrad}[\text{target}] - Q_\theta(s_j, a_j)\big\|^2$
21:        $s_j \leftarrow s'_j$
22:      end for
23:    end if
24:  end for
25: end for
```

3.2 Experimental Setup

To maintain a proper ground for analysis of the additions, we kept the structural PQN code and algorithm unchanged. The only changes/additions to the original code were made in regards to the Dyna architecture i.e., the implementation of the model network and the planning.

We have chosen the MinAtar environment as the test benchmark for the comparison between the two algorithms, running the following environments: *Asterix, Breakout, Freeway and Space Invaders*. We ran both PQN and PDQN with the same general hyperparameters (i.e., not PDQN specific) for a total of 10 runs and 3.0×10^7 steps on each individual environment.

In order to find the best combination of values for PD and PS, we performed a grid-style hyperparameter search of the following values seen in Table 3.1.

One such example of a run can be seen in Figure 3.2. After running this grid search for every environment, we noticed that the most common occurrence of the best performing combination of hyperparameters is with PS $= 2$ and PD $= 1000$ for Space Invaders and Freeway, PS $= 5$ and PD $= 1000$ for Asterix and PS $= 20$ and PD $= 500$ for Breakout. See Appendix B for the rest of the plots showing the performed grid searches on their respective environments.

3.2.1 Reproductibility

All experiments are fully reproducible using the code and data made publicly available at https://github.com/stoianraresstefan/pdqn. Also see the full list of hyperparameters found in Appendix C and the implementation details in Appendix D.

3.2.2 Hardware

The experiments were conducted on personal hardware, being equipped with an NVIDIA GeForce

Table 3.1: Hyperparameter Search and Their Values.

PS	2	5	10	20	
PD	0	100	500	1000	5000

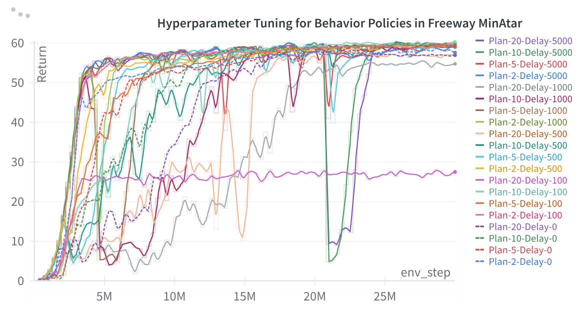


Figure 3.2: Diagram of hyperparameter grid-search performed on Freeway for PDQN. Best performing combination here is PS = 2 and PD = 1000; Other combinations such as PS = 5 and PD = 0 performed similarly.

RTX 4080 Mobile GPU and Intel Core i9-14900HX CPU.

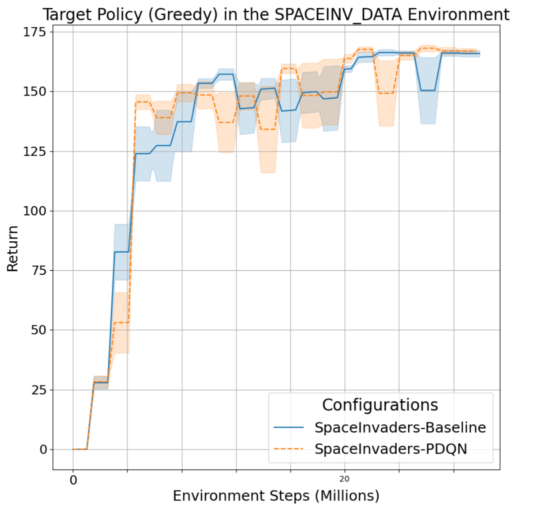
4 Results

In order to asses the performance of the algorithms, we will compare the two given their mean and standard deviation to analyze the overall performance. We will also look at the convergence speed that is observable in the plots.

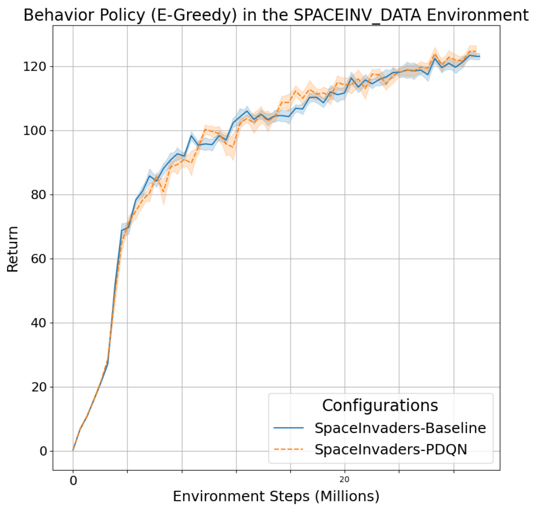
Inspecting the performance of PDQN versus the baseline in Space Invaders, we observe that there is an overlap in performance, PDQN having a mean of 167.00 ± 1.14 on the Target Policy and 125.39 ± 2.06 on the behavior policy, whereas the PQN (i.e., the baseline) has a mean of 165.90 ± 1.26 on the target policy and 123.39 ± 1.57 on the behavior policy. This tells us that model learning and planning do work, but do not outperform PQN. For all the results and plots of both results and world model losses, see Appendix E.

The plots for behavior and target policies above are found in Figures 4.1a and 4.1b.

Another example is PDQN versus baseline in Asterix. Here, the results are quite similar, with the PDQN having a mean of 72.96 ± 0.21 on the target policy and 50.43 ± 0.49 on the behavior policy, while the PQN (i.e. the baseline) has a mean of 73.10 ± 0.16 on the target policy and 49.39 ± 0.59 on the behavior policy. The results tell us that PDQN does not outperform or underperform compared to PQN.



(a) Space Invaders Target Plot.



(b) Space Invaders Behavior Plot.

Figure 4.1: Blue (full) line represents the baseline (PQN), while the orange (dotted) line represents PDQN. The plots represent the learning curves of both algorithms on the Space Invaders environment averaged over 10 runs with PS = 2 and PD = 1000. Both see similar growth and variance.

Looking at the plots of PDQN versus baseline in Asterix (see Figures 4.2a and 4.2b), we notice similar performance in both, but a slight drop in performance for PDQN, caused by the delay in planning, but it does reach back to peak performance quickly.

For a full statistical comparison between PDQN and PQN, see Table E.3 in Appendix E.

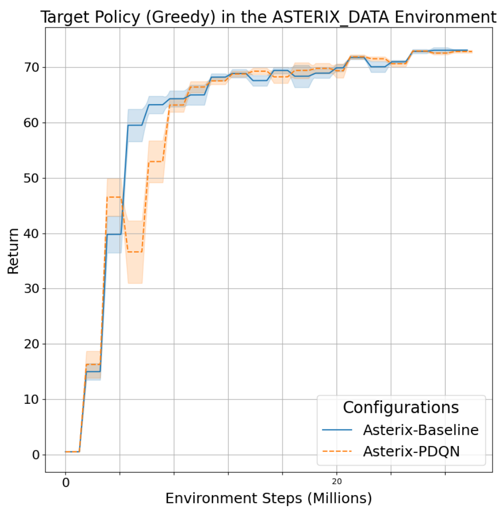
5 Discussion

In this study, we investigated the effects of integrating Dyna Q-learning into the Parallelized Q-Network (PQN) framework, creating Parallelized Dyna Q-Network (PDQN)—a fully online, GPU-accelerated reinforcement learning algorithm. Our primary goal was to assess whether incorporating a planning component through the Dyna architecture could improve convergence speed without compromising performance. Given the limited research on model-based learning in fully GPU-accelerated settings, this work aimed to provide insight into how parallelized model-based methods behave in such a setting.

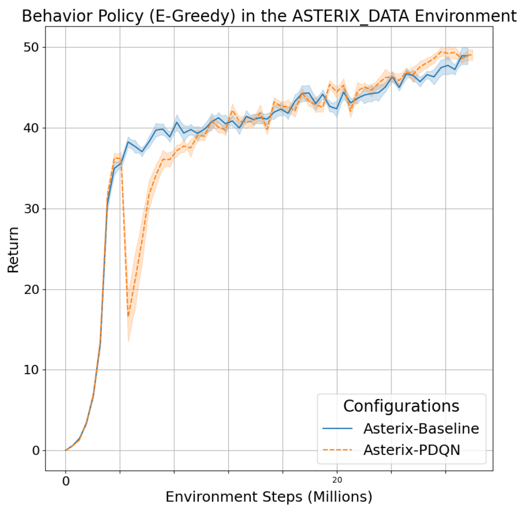
As highlighted in Section 1, end-to-end GPU training in reinforcement learning (RL) remains in its early stages, particularly regarding model learning. Most existing research on world models and model-based RL has been conducted using CPU-based or hybrid CPU-GPU architectures [11, 22, 43], leaving a gap in understanding how these methods perform in a fully GPU-parallelized context.

Motivated by this gap, one of the key questions we explored was whether incorporating model-based planning can provide significant advantages in scenarios that leverage high parallelization. Traditional model-based RL algorithms such as Dyna-Q [33], Dreamer [13], and MuZero [12]—all of which have been extensively studied—rely on different architectural training methods, with Dreamer and MuZero in particular known for their high computational demands.

In contrast, our PDQN algorithm adheres to the fully online and parallelized approach introduced by PQN. The advantages provided by the PQN architecture naturally carry over into our implementation, including the high degree of parallelization enabled by end-to-end GPU training and the removal of the replay buffer. Building upon this



(a) **Asterix Target Plot.**



(b) **Asterix Behavior Plot.**

Figure 4.2: Blue (full) line represents the baseline (PQN), while the orange (dotted) line represents PDQN. The plots represent the learning curves of both algorithms on the Asterix environment averaged over 10 runs with PS = 5 and PD = 1000. At around 5 million steps, PDQN has a drop in performance, but at 8 million steps it stabilizes back. Both see the same consistent growth in score up until 30 million steps.

foundation, we hypothesized that augmenting PQN with Dyna Q-learning would either improve overall performance or accelerate convergence, given the known benefits of model-based learning. However, considering the limited research specifically examining the interplay between parallelization and model-based learning in deep RL, further investigation is necessary to fully understand the trade-offs and potential benefits.

To answer the above questions, we chose four environments from the MinAtar benchmark suite to conduct our research: Asterix, Breakout, Freeway, and Space Invaders. These environments serve as simplified versions of their ALE counterparts [4], conveying significantly less data and requiring less complex policies for learning. However, an important limitation is that our study only explored a subset of environments, meaning that we cannot generalize PDQN's effectiveness to more complex or diverse benchmarks.

Our experimental results indicate that model-based methods can function effectively in a parallelized GPU setting. Although the observed performance remains on par with that of PQN (with the exception of Breakout), they remain promising given the limited number of environments and training runs performed. It is also worth considering whether this performance is influenced by the simplicity of the MinAtar environments, which might not fully leverage the benefits of model-based planning. More extensive testing across different benchmarks would be necessary to establish a stronger claim about PDQN's promising results.

The results in Table E.3 provide statistical evidence regarding the performance differences between PDQN and its baseline PQN across four MinAtar environments. Notably, PDQN demonstrates a statistically significant and large improvement in both target and behavior policies for Breakout, with Cohen's d values exceeding 2. In Freeway, the results also indicate large effects for both policy types, although the direction of the effect differs, with negative effect size for the target policy and positive for the behavior policy. For Asterix and Space Invaders, however, the differences are not statistically significant, and the effect sizes are small. These findings suggest that PDQN can outperform PQN in certain environments, particularly those where planning provides an advantage, but the benefits do not generalize consistently across all tasks. Further analysis is required to understand the environment-specific characteristics that influence the effectiveness of integrated planning in parallelized settings.

On a more pessimistic note, these results were expected to some extent. A research performed by Taher Jafferjee et. al [15] come up with Hallucinated Value Hypothesis (HVH), which suggests that when Dyna-style agents update value functions using simulated experiences from imperfect models, they may generate "hallucinated" states. The study further investigates this by running several experiments, and some results, which are based on Dyna variations, perform worse than normal Q-learning. Aligning our results to this show that PDQN performed as well as normal PQN and, in the case of Breakout, worse, which should add credibility to their findings.

Our research also highlights the importance of PDQN's two distinct hyperparameters: planning steps (PS) and planning delay (PD). These parameters appear to be environment-dependent, influencing the overall performance and convergence speed. This is particularly evident in the observed differences in optimal PS values between Asterix and Space Invaders, as discussed in Section 4. A potential rule of thumb emerging from our results is that short-horizon planning (lower PS values) tends to perform better, while higher PD values allow the model to stabilize before planning begins, leading to more effective updates.

5.1 Future Work

Our results highlight an interesting trade-off between architectural simplicity and learning robustness in model-based RL. Although PDQN successfully integrates planning into a parallelized GPU setting, it does not outperform PQN. The balance between real and simulated experience remains an open challenge in reinforcement learning, and further exploration of this aspect could yield valuable insights for the development of next-generation parallelized model-based algorithms.

To further validate the applicability of PDQN, a natural extension of this work would be to evaluate its performance across a wider range of environments, including larger-scale benchmarks such as ALE [4], the DM Control Suite [35], or Mujoco-based continuous control tasks [38]. Such an evaluation would provide a clearer understanding of how PDQN scales with task complexity and whether the observed trends hold in more challenging and diverse settings.

In parallel, improvements in the model learning component itself present another promising research direction. Future work could explore better architectural designs for the learned world model, incorporating techniques such as contrastive learning [17], transformer-based dynamics models [6], or Bayesian uncertainty estimation [37]. Enhancing model accuracy in this way could mitigate the risks associated with model errors, potentially leading to faster convergence and stronger final policy performance.

Beyond model refinement, modifications to the algorithmic framework of PDQN could also be explored. One promising avenue would be testing variations of the Dyna algorithm itself, such as the Multi-step Predecessor Dyna, which has been proposed to alleviate issues related to hallucinated value updates (HVH) [15]. Alternatively, transitioning to different model-based families, such as the state-of-the-art Dreamer [13] or MuZero [12] algorithms, could offer further performance gains, especially in environments where Dyna-style methods are known to have limitations [3, 41].

Finally, while the current implementation of PDQN ensures reproducibility through a containerized Docker setup, it also introduces deployment constraints. Future work could aim to enhance flexibility by enabling execution outside of Docker while preserving the benefits of an end-to-end parallelized GPU architecture.

6 Conclusions

In conclusion, our exploration of the Parallelized Dyna Q-Network (PDQN) demonstrates that integrating a model-based planning component into a purely online, GPU-accelerated reinforcement learning framework is both feasible and potentially beneficial. Across the selected MinAtar environments, PDQN maintained comparable performance to the baseline Parallelized Q-Network (PQN). Although there are no significant gains or losses, the results suggest that short-horizon planning—with a learned model refined over time—can help stabilize learning in a fully online setting. Notably, the two hyperparameters specific to PDQN, planning steps (PS) and planning delay (PD), were shown to be environment-dependent, hinting that tuning these carefully is key to enhancing performance.

At the same time, the study highlights several avenues for future work. Extending PDQN to larger-scale benchmarks, such as the full Atari suite or more complex continuous-control domains, would clarify how well its advantages transfer beyond simplified environments. Moreover, experimenting with alternative model architectures could mitigate model inaccuracies, particularly when faced with more complex dynamics or longer-horizon tasks. By pursuing these directions, researchers can further evaluate and refine the balance between computational efficiency, sample efficiency, and stability when employing model-based planning in parallelized, end-to-end GPU reinforcement learning.

References

[1] A. Andres, L. Schäfer, S. V. Albrecht, and J. Del Ser. Using offline data to speed up reinforcement learning in procedurally generated environments. *Neurocomputing*, 618: 129079, 2025. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2024.129079. URL https://www.sciencedirect.com/science/article/pii/S0925231224018502.

[2] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

[3] B. Barkley and D. Fridovich-Keil. Stealing that free lunch: Exposing the limits of dyna-style reinforcement learning, 2024. URL https://arxiv.org/abs/2412.14312.

[4] 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*, 47: 253–279, June 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL http://dx.doi.org/10.1613/jair.3912.

[5] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057,

19435274. URL http://www.jstor.org/stable/24900506.

[6] P. Bera and J. Mondal. Predicting future kinetic states of physicochemical systems using generative pre-trained transformer. *bioRxiv*, 2024. doi: 10.1101/2024.05.22.595440. URL https://www.biorxiv.org/content/early/2024/06/19/2024.05.22.595440.

[7] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. Jax: Composable transformations of python+numpy programs. http://github.com/google/jax, 2018. Accessed: 2025-03-25.

[8] K. Farooghi, M. Samadi, and H. Khaloozadeh. Tabular dyna-q algorithm for online calculation of lqr gains: A simulation study. In *2023 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6, 2023. doi: 10.1109/ICEET60227.2023.10525853.

[9] R. Figueiredo Prudencio, M. R. O. A. Maximo, and E. L. Colombini. A survey on offline reinforcement learning: Taxonomy, review, and open problems. *IEEE Transactions on Neural Networks and Learning Systems*, 35 (8):10237–10257, 2024. doi: 10.1109/TNNLS.2023.3250269.

[10] M. Gallici, M. Fellows, B. Ellis, B. Pou, I. Masmitja, J. N. Foerster, and M. Martin. Simplifying deep temporal difference learning, 2024. URL https://arxiv.org/abs/2407.04811.

[11] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2829–2838, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/gu16.html.

[12] H. Guei, Y.-R. Ju, W.-Y. Chen, and T.-R. Wu. Interpreting the learned model in muzero planning, 2024. URL https://arxiv.org/abs/2411.04580.

[13] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models, 2024. URL https://arxiv.org/abs/2301.04104.

[14] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/ioffe15.html.

[15] T. Jafferjee, E. Imani, E. Talvitie, M. White, and M. Bowling. Hallucinating value: A pitfall of dyna-style planning with imperfect environment models, 2020. URL https://arxiv.org/abs/2006.04363.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey, 1996. URL https://arxiv.org/abs/cs/9605103.

[17] P. H. Le-Khac, G. Healy, and A. F. Smeaton. Contrastive representation learning: A framework and review. *IEEE Access*, 8:193907–193934, 2020. doi: 10.1109/ACCESS.2020.3031549.

[18] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. ISSN 1476-4687. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

[19] S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020. URL https://arxiv.org/abs/2005.01643.

[20] C. Li, R. Jia, J. Liu, Y. Zhang, Y. Niu, Y. Yang, Y. Liu, and W. Ouyang. Theoretically guaranteed policy improvement distilled from model-based planning, 2023. URL https://arxiv.org/abs/2307.12933.

[21] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, 1992. ISSN 1573-0565. doi: 10.1007/BF00992699. URL https://doi.org/10.1007/BF00992699.

[22] Z. Liu, X. Xu, P. Qiao, and D. Li. Acceleration for deep reinforcement learning using parallel and distributed computing: A survey. *ACM Comput. Surv.*, 57(4), Dec. 2024. ISSN 0360-0300. doi: 10.1145/3703453. URL https://doi-org.proxy-ub.rug.nl/10.1145/3703453.

[23] F.-M. Luo, T. Xu, H. Lai, X.-H. Chen, W. Zhang, and Y. Yu. A survey on model-based reinforcement learning, 2022. URL https://arxiv.org/abs/2206.09328.

[24] U. A. Mishra, S. R. Samineni, P. Goel, C. Kunjeti, H. Lodha, A. Singh, A. Sagi, S. Bhatnagar, and S. Kolathaya. Dynamic mirror descent based model predictive control for accelerating robot learning, 2021. URL https://arxiv.org/abs/2112.02999.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

[26] J. S. Obando-Ceron and P. S. Castro. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research, 2021. URL https://arxiv.org/abs/2011.14826.

[27] M. Pei, H. An, B. Liu, and C. Wang. An improved dyna-q algorithm for mobile robot path planning in unknown dynamic environment. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(7):4415–4425, 2022. doi: 10.1109/TSMC.2021.3096935.

[28] B. Peng, X. Li, J. Gao, J. Liu, K.-F. Wong, and S.-Y. Su. Deep dyna-q: Integrating planning for task-completion dialogue policy learning, 2018. URL https://arxiv.org/abs/1801.06176.

[29] B. Peng, Y. Mu, Y. Guan, S. E. Li, Y. Yin, and J. Chen. Model-based actor-critic with chance constraint for stochastic system, 2021. URL https://arxiv.org/abs/2012.10716.

[30] A. Rutherford, B. Ellis, M. Gallici, J. Cook, A. Lupu, G. Ingvarsson, T. Willi, A. Khan, C. S. de Witt, A. Souly, et al. Jaxmarl: Multi-agent rl environments in jax. *arXiv preprint arXiv:2311.10090*, 2023. URL https://arxiv.org/abs/2311.10090.

[31] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[32] S. Sujit, P. H. M. Braga, J. Bornschein, and S. E. Kahou. Bridging the gap between offline and online reinforcement learning evaluation methodologies, 2023. URL https://arxiv.org/abs/2212.08131.

[33] R. S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bull.*, 2(4):160–163, July 1991. ISSN 0163-5719. doi: 10.1145/122344.122377. URL https://doi-org.proxy-ub.rug.nl/10.1145/122344.122377.

[34] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[35] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller. Deepmind control suite, 2018. URL https://arxiv.org/abs/1801.00690.

[36] T. Tateyama, S. Kawata, and T. Shimomura. Parallel reinforcement learning systems using exploration agents and dyna-q algorithm. In *SICE Annual Conference 2007*, pages 2774–2778, 2007. doi: 10.1109/SICE.2007.4421460.

[37] M. Teye, H. Azizpour, and K. Smith. Bayesian uncertainty estimation for batch normalized deep networks. In J. Dy and A. Krause,

editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4907–4916. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/teye18a.html.

[38] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

[39] J. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. In M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996. URL https://proceedings.neurips.cc/paper_files/paper/1996/file/e00406144c1e7e35240afed70f34166a-Paper.pdf.

[40] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil. Deep reinforcement learning and the deadly triad, 2018. URL https://arxiv.org/abs/1812.02648.

[41] Y. Wan, A. Rahimi-Kalahroudi, J. Rajendran, I. Momennejad, S. Chandar, and H. van Seijen. Towards evaluating adaptivity of model-based reinforcement learning methods, 2022. URL https://arxiv.org/abs/2204.11464.

[42] C. J. C. H. Watkins. *Learning with Delayed Rewards*. Phd thesis, Cambridge University Psychology Department, 1989.

[43] Z. Yang, L. Xing, Z. Gu, Y. Xiao, Y. Zhou, Z. Huang, and L. Xue. Model-based reinforcement learning and neural-network-based policy compression for spacecraft rendezvous on resource-constrained embedded systems. *IEEE Transactions on Industrial Informatics*, 19(1):1107–1116, 2023. doi: 10.1109/TII.2022.3192085.

[44] K. Young and T. Tian. Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments, 2019. URL https://arxiv.org/abs/1903.03176.

[45] L. Zou, L. Xia, P. Du, Z. Zhang, T. Bai, W. Liu, J.-Y. Nie, and D. Yin. Pseudo dyna-q: A reinforcement learning framework for interactive recommendation. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, WSDM '20, page 816–824, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368223. doi: 10.1145/3336191.3371801. URL https://doi-org.proxy-ub.rug.nl/10.1145/3336191.3371801.

A Pseudocodes

Algorithm A.1 PQN ($s, a, s', r,$ target are B-dim vectors)

```
1: $\phi \leftarrow$ initialize regularized Q-network parameters
2: $s \leftarrow$ initial state $s_0 \sim P_0$
3: for each episode do
4:     for each $i \in B$ (in parallel) do
5:         $$a_i \leftarrow \begin{cases} a_i \sim \text{Unif}(\mathcal{A}), & \text{with prob. } \epsilon, \\ \arg\max_{a'} Q_\phi(s_i, a'), & \text{otherwise,} \end{cases}$$
6:         $s'_i, r_i \leftarrow s'_i \sim P_S(s_i, a_i), \quad r_i \sim P_R(s_i, a_i)$
7:         $\text{target}_i \leftarrow r_i + \gamma\, \mathbf{1}(\text{not terminal}) \max_{a'} Q_\phi(s'_i, a')$
8:     end for
9:     $\phi \leftarrow \phi - \alpha \nabla_\phi \sum_{i \in B} \big[\text{StopGrad}(\text{target}_i) \; - \; Q_\phi(s_i, a_i)\big]^2$
10: end for
```

Algorithm A.2 Dyna-Q

```
1: $Q \leftarrow$ initialize Q-function
2: $\mathcal{M} \leftarrow$ initialize model (empty or parametric)
3: $s \leftarrow s_0 \sim P_0$
4: for each episode do
5:     for each real step do
6:         $$a \leftarrow \begin{cases} \text{random action from } \mathcal{A}, & \text{with prob. } \epsilon, \\ \arg\max_{a'} \; Q(s, a'), & \text{otherwise,} \end{cases}$$
7:         $s', \, r \; \leftarrow$ environment dynamics from $(s, a)$
8:         $$Q(s,a) \; \leftarrow \; Q(s,a) \; + \; \alpha\Big[r \; + \; \gamma \max_{a'} Q(s', a') \; - \; Q(s,a)\Big]$$
9:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{(s, a, r, s')\}$     /* Update the model */
10:        for $k = 1$ to $K$ do
11:            $(\hat{s}, \hat{a}) \; \leftarrow$ sample from $\mathcal{M}$
12:            $(\hat{r}, \hat{s}') \; \leftarrow \; \mathcal{M}(\hat{s}, \hat{a})$
13:            $$Q(\hat{s}, \hat{a}) \; \leftarrow \; Q(\hat{s}, \hat{a}) \; + \; \alpha\Big[\hat{r} \; + \; \gamma \max_{a'} Q(\hat{s}', a') \; - \; Q(\hat{s}, \hat{a})\Big]$$
14:        end for
15:        $s \leftarrow s'$
16:    end for
17: end for
```

B More Hyperparameter Search Plots

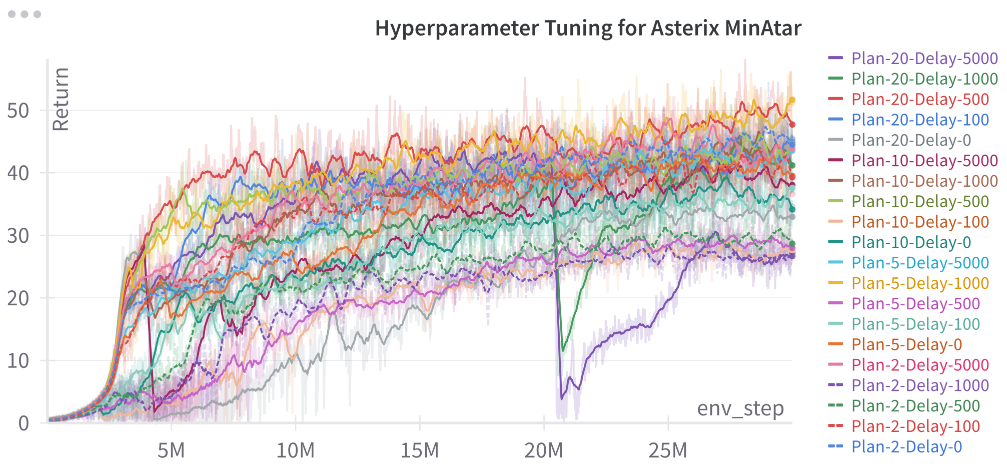


Figure B.1: Diagram of hyperparameter grid search performed on Asterix. Chosen parameters: PS = 5 and PD = 1000.

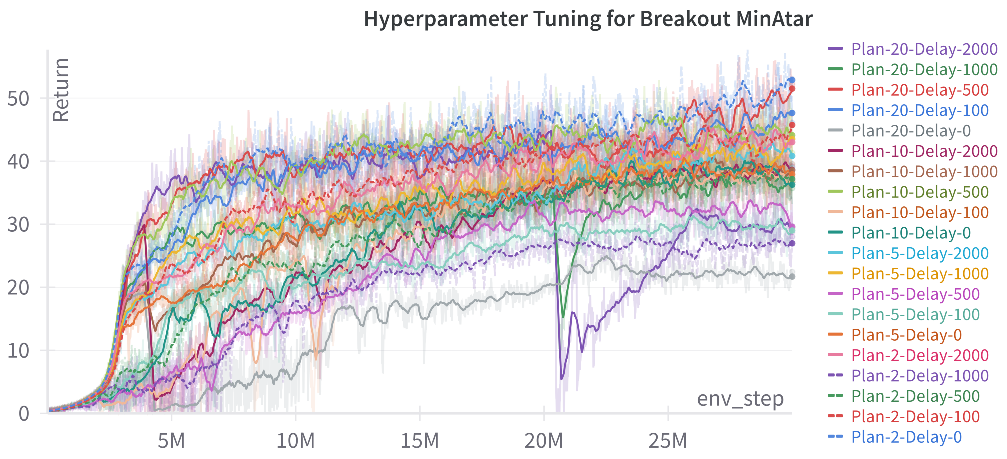


Figure B.2: Diagram of hyperparameter grid search performed on Breakout. Chosen parameters: PS = 20 and PD = 500.

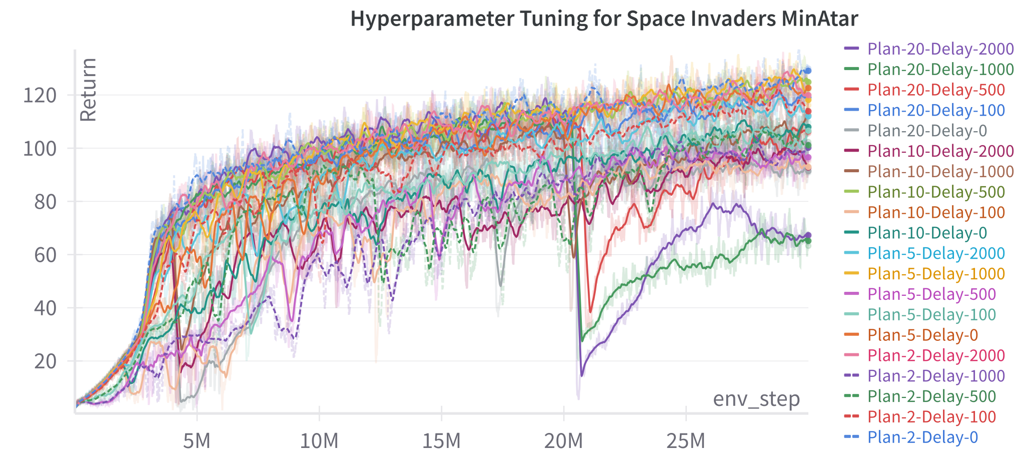


Figure B.3: Diagram of hyperparameter grid search performed on Space Invaders. Chosen parameters: PS = 2 and PD = 1000.

C Hyperparameters

Table C.1: Hyperparameters used for the PDQN experiments on MinAtar environments.

Hyperparameter	**Value**
Algorithm (ALG_NAME)	PDQN
Total timesteps	3×10^7
Timesteps for decay	3×10^7
Number of environments (NUM_ENVS)	128
Number of steps per update (NUM_STEPS)	32
Exploration ϵ start (EPS_START)	1.0
Exploration ϵ finish (EPS_FINISH)	0.05
Exploration ϵ decay (EPS_DECAY)	0.1
Planning steps (PLANNING_STEPS)	2, 5, 10 or 20
Planning delay updates (PLANNING_DELAY_UPDATES)	0, 100, 500, 1000 or 5000
Use planning-specific ϵ (USE_PLANNING_EPS)	False
Planning ϵ start (EPS_START_PLAN)	1.0
Planning ϵ finish (EPS_FINISH_PLAN)	0.05
Planning ϵ decay (EPS_DECAY_PLAN)	0.1
Number of minibatches (NUM_MINIBATCHES)	32
Number of epochs per update (NUM_EPOCHS)	2
Normalization type (NORM_TYPE)	layer_norm
Learning rate (LR)	5×10^{-4}
Learning rate linear decay (LR_LINEAR_DECAY)	True
Maximum gradient norm (MAX_GRAD_NORM)	10
Discount factor (γ)	0.99
GAE parameter (λ)	0.65
Environment (ENV_NAME)	One of: Asterix-MinAtar, Breakout-MinAtar, Freeway-MinAtar, SpaceInvaders-MinAtar
Environment arguments (ENV_KWARGS)	{}
Test during training	True
Test interval (TEST_INTERVAL)	0.05
Test number of environments (TEST_NUM_ENVS)	128
Test ϵ (EPS_TEST)	0.0 (greedy policy)
Number of seeds (NUM_SEEDS)	1 (specified in `config.yaml`)
Random seed (SEED)	0 (specified in `config.yaml`)

D Implementation Details

`Python` is used as the main programming language, and the main library used is `JAX`. MinAtar is loaded through the `Gymnax` library. *WandB* (or *Weights and Biases*) tool was used as the main empirical testing, but also for doing the hyperparameter grid search and plotting of said hyperparameter searches. `Pandas` and `Numpy` were used to read the CSV's of the runs and make the plots and get the means and standard deviations.

E More Results and Performance Plots

Table E.1: Results: Baseline vs PDQN (Target)

	Asterix	Breakout	Freeway	Space Invaders
Baseline	73.10 ± 0.16	98.99 ± 1.33	39.72 ± 7.89	165.90 ± 1.26
PDQN	72.96 ± 0.21	63.41 ± 6.76	65.80 ± 0.25	167.00 ± 1.14

Table E.2: Results: Baseline vs PDQN (Behavior)

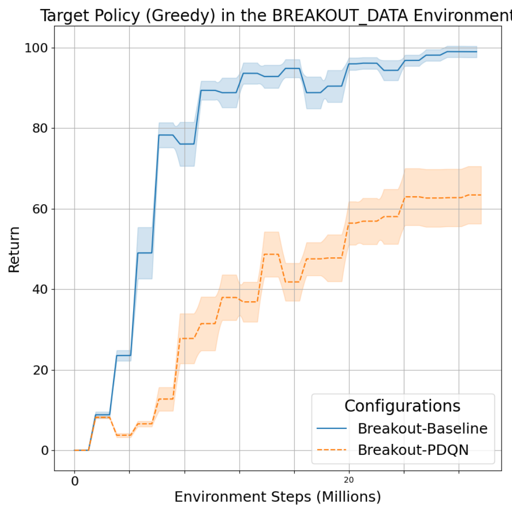
	Asterix	Breakout	Freeway	Space Invaders
Baseline	9.39 ± 0.59	42.73 ± 1.66	60.75 ± 0.15	123.39 ± 1.57
PDQN	50.43 ± 0.49	28.42 ± 1.11	60.23 ± 0.07	125.39 ± 2.06

Table E.3: Statistical comparison between PDQN and PQN across MinAtar environments using *p*-values and Cohen's *d* effect sizes.

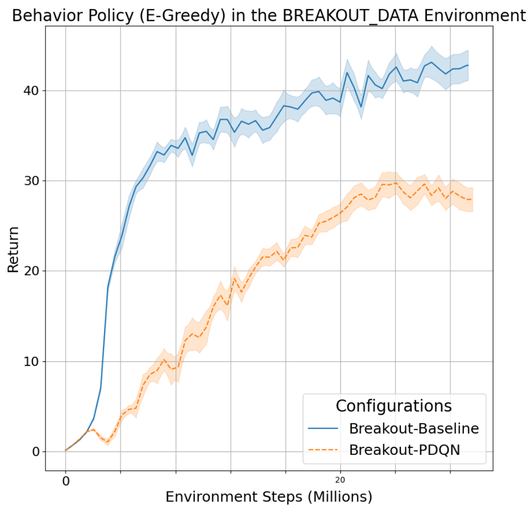
Environment	**Policy Type**	***p*-value**	**Cohen's *d***	**Effect Size**
Asterix	Target	0.6095	0.23	Small
	Behavior	0.2120	-0.58	Medium
Breakout	Target	0.0007	2.19	Large
	Behavior	4.74e-06	3.04	Large
Freeway	Target	0.0120	-1.40	Large
	Behavior	0.0128	1.29	Large
Space Invaders	Target	0.5470	-0.27	Small
	Behavior	0.4300	-0.36	Small

E.1 Model Loss Plots

To enhance clarity and interpretability, the Y-axes of all model loss plots are displayed on a logarithmic scale.

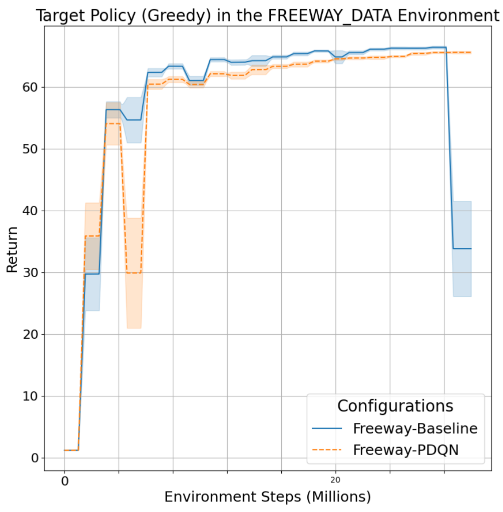


(a) **Breakout Target Plot.**

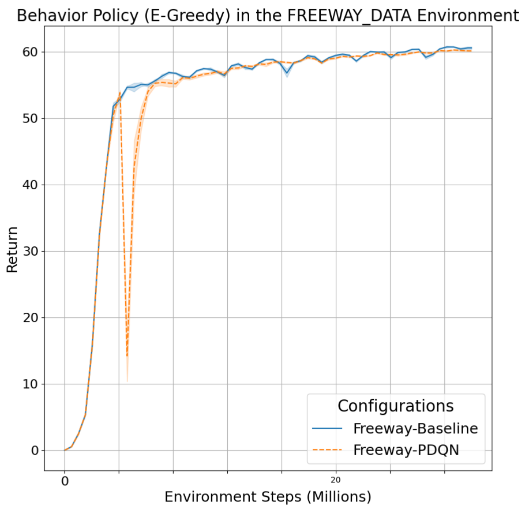


(b) **Breakout Behavior Plot.**

Figure E.1: Blue (full) line represents the baseline (PQN), while the orange (dotted) line represents PDQN. The plots represent the learning curves of both algorithms on the Breakout environment averaged over 10 runs with PS = 20 and PD = 500. Here we notice that PDQN performs much worse than PQN, and at around 24 million steps, PDQN seems to have reached a plateau, whereas PQN shows signs of improvement.



(a) Freeway Target Plot.



(b) Freeway Behavior Plot.

Figure E.2: Blue (full) line represents the baseline (PQN), while the orange (dotted) line represents PDQN. The plots represent the learning curves of both algorithms on the Freeway environment averaged over 10 runs with PS = 2 and PD = 1000. After 4 million steps, PDQN has a drastic drop in performance, but it adjusts itself back within approximately 1 million steps. At 8 million steps, both algorithms show slow, but steady increase in performance. PQN, however, shows signs of runs with bad updates, leading to a sudden drop in performance, at around 27 million steps (observable in plot a).

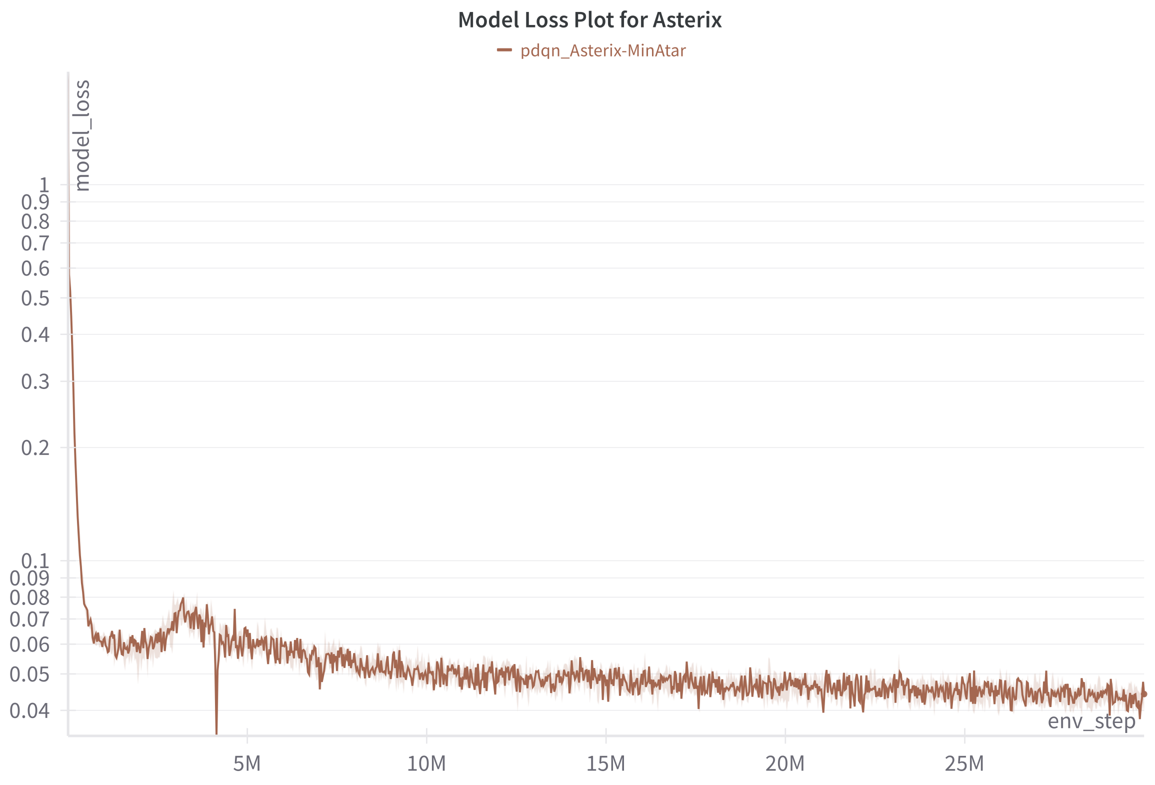


Figure E.3: Model Loss (Y axis) over Environment Time steps (X axis) plot for Asterix MinAtar

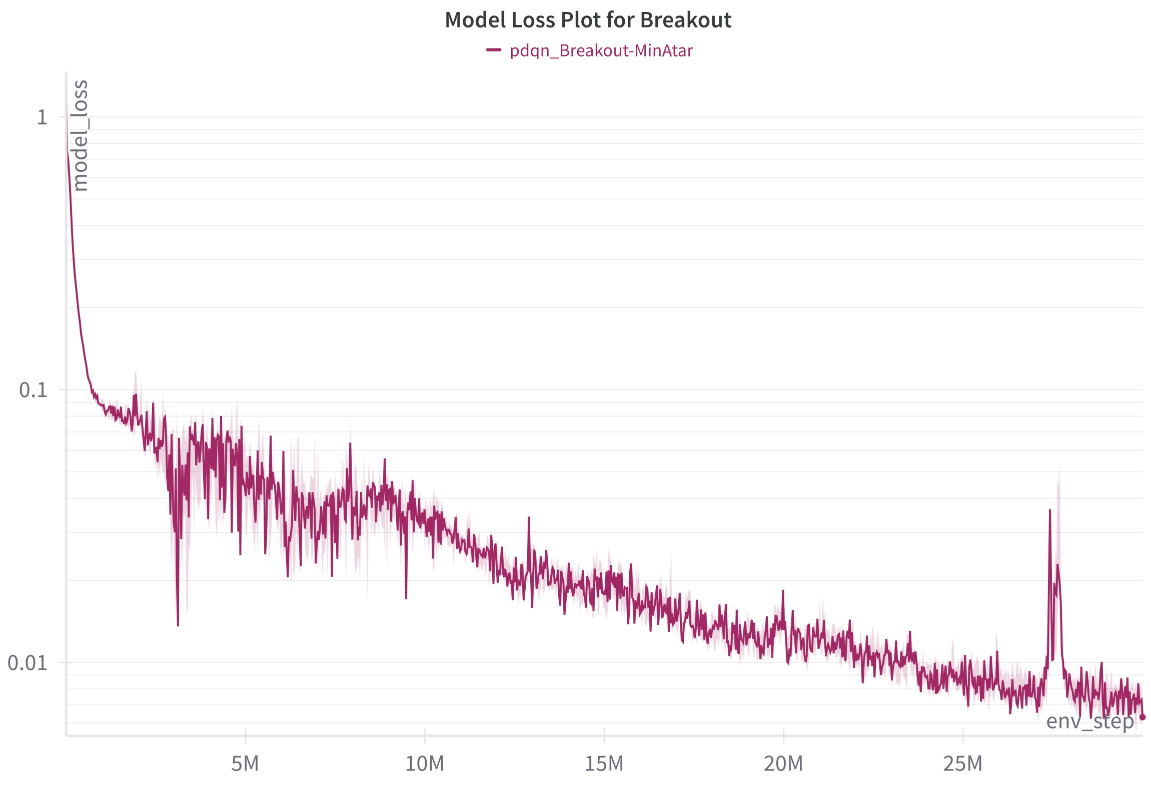


Figure E.4: Model Loss plot (Y axis) over Environment Time steps (X axis) for Breakout MinAtar

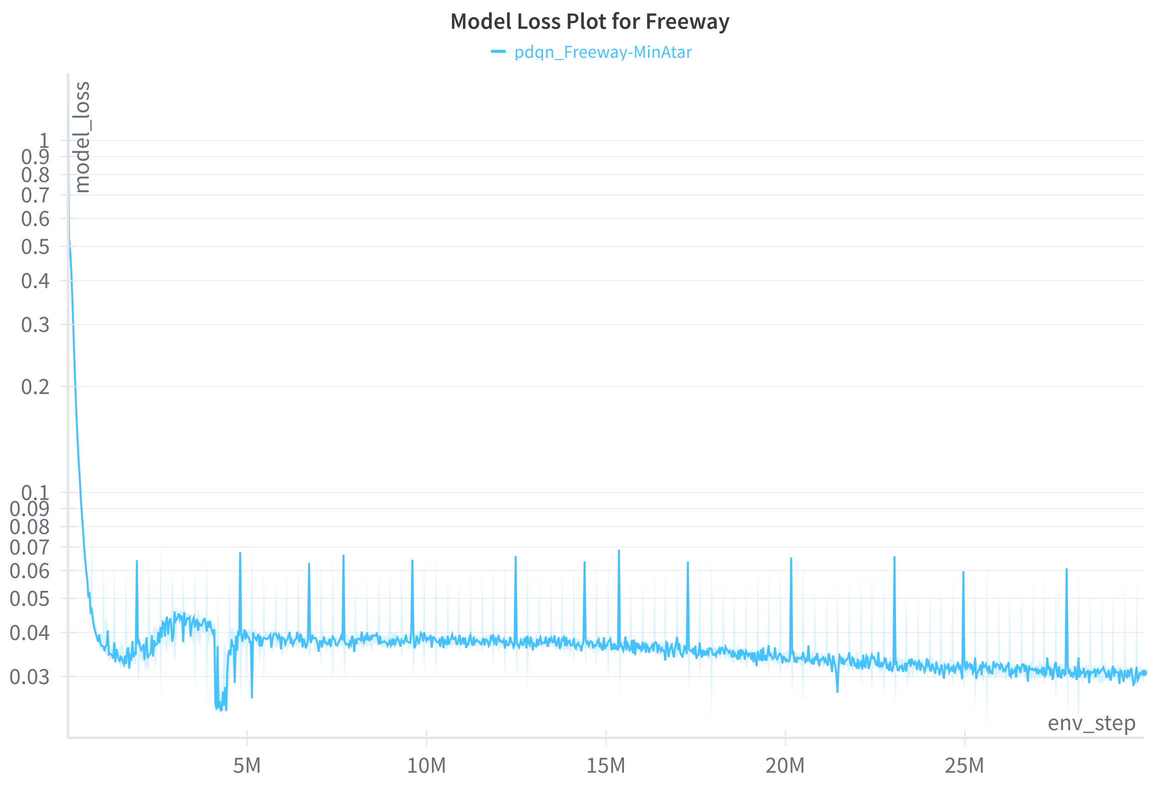


Figure E.5: Model Loss plot (Y axis) over Environment Time steps (X axis) for Freeway MinAtar

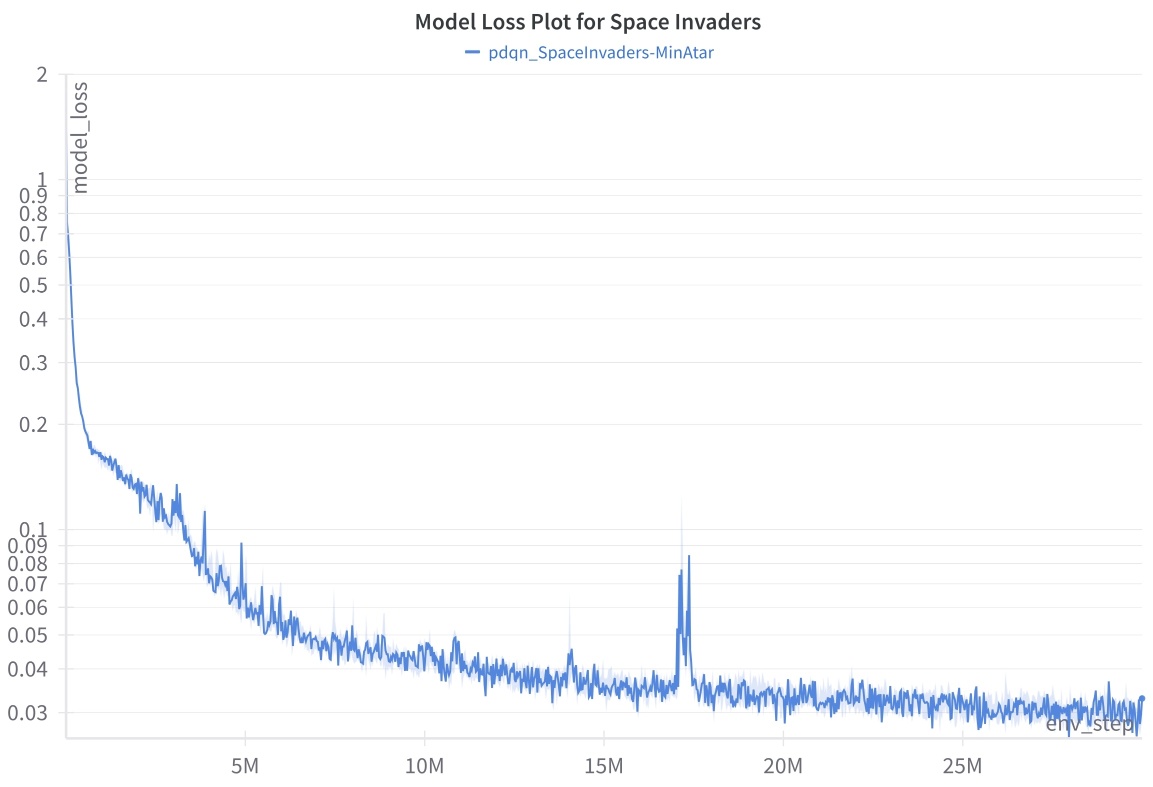


Figure E.6: Model Loss plot (Y axis) over Environment Time steps (X axis) for Space Invaders MinAtar