THESIS

POLICY OPTIMIZATION FOR INDUSTRIAL BENCHMARK USING DEEP REINFORCEMENT LEARNING

Submitted by

Anurag Kumar

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2020

Master's Committee:

Advisor: Charles Anderson

Hamid Chitsaz
Michael Kirby

ABSTRACT

POLICY OPTIMIZATION FOR INDUSTRIAL BENCHMARK USING DEEP REINFORCEMENT LEARNING

Significant advancements have been made in the field of Reinforcement Learning (RL) in recent decades. Numerous novel RL environments and algorithms are mastering these problems that have been studied, evaluated, and published. The most popular RL benchmark environments produced by OpenAI Gym and DeepMind Labs are modeled after single/multi-player board, video games, or single-purpose robots and the RL algorithms modeling optimal policies for playing those games have even outperformed humans in almost all of them. However, the real-world applications using RL is very limited, as the academic community has limited access to real industrial data and applications. Industrial Benchmark (IB) is a novel RL benchmark motivated by Industrial Control problems with properties such as continuous state and action spaces, high dimensionality, partially observable state space, delayed effects combined with complex heteroscedastic stochastic behavior. We have used Deep Reinforcement Learning (DRL) algorithms like Deep Q-Networks (DQN) and Double-DQN (DDQN) to study and model optimal policies on IB. Our empirical results show various DRL models outperforming previously published models on the same IB.

ACKNOWLEDGEMENTS

I want to thank my Advisor, Prof. Chuck Anderson, for his constant guidance and encouragement, without which this thesis work would not be possible. I would also like to thank my committee members, Dr. Hamid Chitsaz and Prof. Michael Kirby, for their constructive advice and invaluable tips. I want to thank Mountain Data Group LLC for allowing me to gain hands-on industry experience in machine learning and reinforcement learning. Special thanks to the Mullaney's, for being a great host and making me feel at home. I would also like to thank my parents, Mr. Arun Kumar and Mrs. Saroj Kumari, and my brother Ayush Kumar, for their constant support throughout my Master's degree.

DEDICATION

I would like to dedicate this thesis to my family and friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In recent years, major advancements in the field of reinforcement learning (RL) [1] can be attributed to the numerous complex benchmarks developed by the research community like OpenAI Gym [2], DeepMind Labs [3], MuJoCo [4] and many more. These benchmarks attempt to solve problems ranging from searching, planning, to optimizing under partial supervision. The ability to learn from the interactions with the environment and continuously improve its performance by the process of receiving rewards and punishments on every action taken enables RL models to train systems to respond to unforeseen environments. Deep Reinforcement Learning (DRL) [5] which combines RL with Artificial Neural Networks (ANN) [6] for value function approximations has led to development of agents with superhuman performance in Atari [7] games. Deep Q-Networks (DQN) [8] and Double-DQN (DDQN) [9] enable us to represent complex high-dimensional data, i.e., state and action space in compact low-dimensional vectors, and therefore can be used for optimizing policies for decision-making problems.

Industrial control systems like steel processing [10], pulp, and paper processing [11], and car manufacturing [12] or power generation with wind or gas turbines [13] have always been an exciting area of interest for the RL researchers. The hope is to develop an intelligent agent that can learn and produce optimal policies for industrial control problems mentioned earlier. However, these DRL algorithms have not been tested enough on real-world problems due to the high risk of failures in these complex and expensive systems. Therefore, there is a huge demand for simulators that have high dimensionality combined with complex heteroscedastic stochastic behavior. The major challenge in developing such a simulator is the limited access to industrial data and applications. There are a few new simulators which have been introduced to the research community like Industrial Benchmark (IB) [14], GEM Electric Motors [15], Micro-Grid [16], AnyTrading by OpenAI Gym [2], and autonomous driving simulators [17] [18] [19]. IB is a novel simulator that can be used to model industrial control problems with similar hardness and complexity as many real-world systems. IB is unique as it is not designed to simulate any specific industrial problem, unlike the others designed for a specific task like controlling electric motors, optimization of micro-grids based on energy conversion by power electric converters, trading securities, and autonomous driving.

In this work, we will use DQN and DDQN to develop optimal policies using the novel IB for discrete action space with different architectures of Q-Network, gradient descent optimizer [20], and uniform and prioritized experience replays.

1.1 Research Question

This work explores the following research questions:

- **[RQ1]** How to use Deep Reinforcement Learning for Policy Optimization for discrete action space on an Industrial Benchmark?
- **[RQ2]** What are the effects of Experience Replay, architectures of Artificial Neural Networks, and other hyper-parameters on optimal policies for the Industrial Benchmark problem?

1.2 Overview of Approach

Industrial Benchmark (IB) [14] problem has a partially observable continuous state space of six random variables. Three of these variables are control variables, which can take inputs from the agent to improve the existing policy. Two other variables which depend on the complete Markov state (Appendix A) of the environment are used to compute the reward for each state transition $(s_t, a_t \longrightarrow (s_{t+1}, a_{t+1}))$ and the last observable variable represents the external load to the environment which cannot be influenced by the agent.

Q-learning based techniques will be used for approximating the optimal action-value function to get the best policy. We will use Double Q-learning to overcome the limitation of maximization bias in the Q-learning. Artificial Neural Networks (ANN) [6] will be used as function approximators in DQN [8] and DDQN [9]. We will use multiple combinations of the observable state as an input to our neural networks. We will analyze the effect of hyper-parameters on the policy by experimenting with different sizes and types of experience replay buffer [21], different neural network architectures, different optimizers [20], and more. We will explore both constant and variable external loads. At a minimum, we will get the best optimal policies with discrete action space and constant external load. We have implemented all of these methods using TensorFlow [22], Keras [23], Keras-RL [24] and IB OpenAI Gym wrapper.

1.3 Research Contributions

Through this research, we demonstrate DRL techniques to produce an optimal policy on IB. This work presents a detailed analysis of the following design choices and hyper-parameters that can be used to improve the RL agent's performance.

1. We show that the external load is directly proportional to the complexity of the environment. So, complex environments require more sophisticated models.

2. We demonstrate that Uniform Experience Replay Buffer performs better than Prioritized Experience Replay Buffer, and we also show that increasing the size of the replay buffer improves and stabilizes the policy.

3. We experimented with SGD and Adam optimizers and evaluated the average fitness value of the policies for different settings.

4. We tried three different input patterns consisting of the observable state to the neural networks.

5. We implemented Neural Networks with four different architectures, mainly based on the number of dense layers included in the network and evaluated the fitness value for the same environment.

6. The update frequency of the Target Model in DDQN was evaluated for three different frequencies.

1.4 Organization

The organization of the rest of this document is as follows. Chapter 2 describes work that is related to our work or is a prerequisite to our methodology. Chapter 3 gives an overview of DRL algorithms, as well as describes DQN and DDQN along with all the design choices that we made. Chapter 4 provides information about our experimental setup, implementation details, and our empirical results. Lastly, our conclusions and future work are described in Chapter 5.

Chapter 2

Prerequisites and Related Work

2.1 Reinforcement Learning

Reinforcement Learning (RL) [1] is one of the sub-domains of Machine Learning, which itself is a sub-domain of Artificial Intelligence. It is classified as a semi-supervised learning technique because the agent is trained without any prior knowledge of the environment, but it does receive rewards (positive/neutral/negative) from its interaction with the environment. The agent needs to discover on its own which actions lead to positive rewards and which do not. Through exploration, an RL agent learns to interact with the environment and improves over time with experience by maximizing (or minimizing) some cumulative reward function.

Like any other learning technique, RL has its challenges that must be considered while making potential trade-offs. One of the major challenges is the trade-off between exploration and exploitation, where an agent needs to decide between trying new actions at the risk of lower reward or using its past experiences at the risk of getting stuck at local maxima (or minima). Another challenge is to predict if a new situation (state, action) is good or bad from its experience. And finally, we also need to consider delayed consequences of the agent's action, i.e., if the present high (or low) reward is a result of the last action or some other action in the past? We will tackle these challenges in Chapter 3.

RL has gained a lot of popularity in recent years with the introduction of great policy optimization techniques like Deep Q-Networks (DQN) [8], Double DQN (DDQN) [9], Continuous DQN [25], Actor-Critic Model [26], and Continuous control [27], and the incredible performances of these algorithms in Atari Games [7], and Go [28].

2.2 Game based RL Environments

Over the years, the RL research community has introduced numerous benchmarks to evaluate the performance of RL algorithms. A large number of these benchmarks are based on board games and video games with state and action space ranging from less than 100 to more than a million. Some of the most popular environments including Cart Pole [29], Mountain Car [30], and Arcade Learning Environment [31] are almost always used to benchmark newer RL algorithms. Another very significant benchmark is the MuJoCo [4], which stands for **Mu**lti-**Jo**int dynamics with **Co**ntact and is a physics engine with very detailed contact simulation designed for model-based control. OpenAI Gym [2] provides a set of challenging benchmarks for continuous control tasks like pushing/sliding/picking blocks, and in-hand object manipulations with a robotic arm for *Multi-Goal Reinforcement Learning* and *Hindsight Experience Replay*. Obstacle Tower Environment [32] is a new benchmark designed to test agents in computer vision, navigational skills, high-level planning, and generalization over experiences. GymGo is an environment for the board game Go which can be used to test advanced hybrid RL algorithms like the one used in AlphaGo [28].

2.3 Other RL Environments

The OpenSim RL [33] is a musculoskeletal reinforcement learning environment that allows researchers to develop and master complex physiologically accurate movement environments for walking, running, and other real-world control problems.

The Gym Electric Motor (GEM) [15] is a benchmark for developing RL agents for controlling electric motors. It can be used to model several variants of DC motors, permanent magnet synchronous motor, power electric converters, and mechanical load as well. It also provides a cascaded PI-controller as a baseline for comparing RL agents.

The OpenModelica Microgrid Gym (OMG) [16] is a benchmark for simulation and control optimization of Micro-and Smart Grids (MSG). It offers plug-and-play controller testing for modeling arbitrary MSG topologies and RL-based controllers. MSGs are very important for integrating renewable energy sources in conventional electricity grids. It is driven by power electric converters due to their high efficiency and flexibility. Therefore RL agents controlling electric converters in microgrids are of great interest to researchers.

All the real-world benchmarks mentioned above are of significant importance to both the research community and the industry. However, they are designed to model a specific control problem, and therefore they may not be useful in developing and testing RL agents from across domains. However, there is another real-world environment that is described in the next section.

2.4 Industrial Benchmark

We have established the need for more real-world benchmarks which can help us develop agents to model manufacturing [12] and processing in the industries [10] [11] [13]. Let us look at the dynamics of IB [14] and understand how it can simulate a generic industrial application.

A central feature of the IB is the continuous, high-dimensional, and partially observable state space. It has both discrete and continuous action space, which is made up of three continuous components and affects three control inputs, namely the velocity v, gain g, and shift h. The known reward function for this benchmark has two components with opposite dependencies on the action. The IB dynamics include stochastic and delayed effects on the state. It is also heteroscedastic as the observable state noise, and probability distributions are determined by the latent variables. We can also set the external load to the environment, which is independent of the effects of the actions.

The observable state control variables v, g, and h at any given time step t are influenced by actions $a_t = (a_1, a_2, a_3)$ where $a_i \in [-1, 1]$ for continuous action space and $a_i \in \{-1, 0, 1\}$ for discrete action space. The external actions $a_t = (a_1, a_2, a_3)$ is actually represented as $a_t = (\Delta v_t, \Delta g_t, \Delta h_t)$ where Δv_t, Δg_t, Δh_t depends on the entire state s_t. The complete state s_t is the Markov state which is only partially observed by the agent. The observable state $o = (v, g, h, p, c, f) \subset s$ where p is the setpoint, c is the consumption and f is the fatigue. Setpoint p simulates the external force, such as the speed of the wind driving a turbine or load of power plant that directly affects the agent but it has no control over the external environment. The variable fatigue f represents the detrimental wear and tear suffered by the system. The resources consumed in the processing or manufacturing like fuel, power, is represented by the variable consumption c. Please refer to Appendix A for the complete state space.

At each time step t, the agent takes an action a_t at current state s_t to produce the next state s_{t+1} which contains c_{t+1} and f_{t+1}. The reward is a function of consumption and fatigue and can be calculated using eq.(2.1).

$$r_{t+1} = -c_{t+1} - 3f_{t+1}, \tag{2.1}$$

The Markov state s_t can be approximated using a sufficient number of historical observations over a time horizon of H. Please refer to the original paper [14] for further details about the sub-dynamics of state variables.

Hein, et al., [34] have used IB to evaluate a batch RL algorithm Particle Swarm Optimization Policy (PSO-P) [35] and compared the results from PSO-P to the results of closed-form control policies derived from the model-based Recurrent Control Neural Network (RCNN) [36] and model-free Neural Fitted Q-Iteration (NFQ) [37]. NFQ performed the worst among the three methods due to its limitations and instability during training. RCNN produced some good performing closed-form policies. PSO-P outperformed the others for almost every setpoint. The main disadvantage of this method is the high computational efforts required for calculating the next action. According to their calculations, it is under 8 seconds, which is still too long for many industrial applications.

Hein, et al., [38] have used IB to autonomously generate Interpretable Fuzzy Controllers using Fuzzy Genetic Programming Reinforcement Learning (FGPRL) and is compared with the results from Fuzzy Particle Swarm Reinforcement Learning (FPSRL). FGPRL can select the relevant feature from a state, determine the size of the fuzzy rule set, and adjust all control parameters on its own in one single step. They used model-based batch reinforcement learning for training policies and then used Monte Carlo rollouts to predict each policy's performance. FPSRL, on the other hand, used particle swarm intelligence to tune the fuzzy policy parameters by selecting only the important features before the training to generate interpretable policies. However, initial heuristic selection and manually creating policy structures are the main limitations of this approach. The advantage of FGPRL is that it searches the full space of fuzzy controllers and automatically determines the essential state variables and number of fuzzy rules.

The fuzzy controllers and the interpretable policies are not directly relevant to our work, but we will use the fitness function (2.3), and the fitness value to compare results against our optimal policies in Chapter 4. Let $\mathcal{R}(s_t, \pi)$ be the sum of all past rewards defined as follows

$$\begin{gathered}\mathcal{R}(s_t, \pi) = \sum_{k=0}^{T-1} \gamma^k r(s_{t+k}, \pi(s_{t+k}), s_{t+k+1}) \\ \text{with } s_{t+k+1} = g(s_{t+k}, a_{t+k})\end{gathered} \tag{2.2}$$

where state $s_{t+1} \in \mathcal{S}$ at time step $(t+1)$ is generated using a transition function $g : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{S}$ with $g(s_t, a_t) = s_{t+1}$, the corresponding reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$, with $r(s_t, a_t, s_{t+1}) = r_{t+1}$ and the optimal policy, $\pi \in \Pi$, where Π is the set of all policies, and $\gamma \in [0, 1]$ is the discount factor. Thus the overall fitness value $\mathcal{F}(\pi)$ is obtained by averaging $\mathcal{R}(s_t, \pi)$ over all the starting states $s_t \in S \subset \mathcal{S}$. Thus, the solution to the problem are policies, $\hat{\pi} \in \underset{\pi \in \Pi}{\arg\max}\, \mathcal{F}(\pi)$, where

$$\mathcal{F}(\pi) = \frac{1}{|S|} \sum_{s_t \in S} \mathcal{R}(s_t, \pi) \tag{2.3}$$

is the fitness function. The fitness values for the best policy developed using FPSRL and FGPRL are $\tilde{\mathcal{F}} = -5700$ and $\tilde{\mathcal{F}} = -5635$, respectively. The fitness values of these models are based on the time horizon H of 100, whereas the fitness values of all the policies that we will produce in Section 4.2 are based on the time horizon H of 1000.

Chapter 3

Methodology

3.1 Reinforcement Learning Algorithms

Let us tackle some of the Reinforcement learning challenges [39] that we listed Section 2.1.

Exploitation vs. Exploration

RL algorithms are required to explicitly explore in order to ensure that its policy is truly optimal. We have multiple techniques to overcome this issue such as *Dynamic Programming* where we can use Bayesian reasoning to calculate the expected future reward for all actions given the current policy is optimal and pick the best action, *Greedy Strategy* where we always pick the action which we know will give the maximum reward, ϵ-$greedy$ strategy where we can randomly select an action with probability ϵ and exponentially decrease ϵ over time. In this work, we will use ϵ-$greedy$ for managing exploitation versus exploration.

Delayed Reward

The RL agent must take into account which actions are good, based on the reward that it will get arbitrarily at some time in the future. One of the ways to solve this problem is to model it as *Markov Decision Process* (MDP). A MDP consists of a set of states $\mathcal{S}$ and actions $\mathcal{A}$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ and a transition function $g : \mathcal{S} \times \mathcal{A} \longrightarrow \Pi(\mathcal{S})$ where $\Pi(\mathcal{S})$ is probability distribution over the set $\mathcal{S}$ (i.e., maps states to probabilities). This approach assumes that we have a model, and we are just solving for an optimal policy.

Problems where the model is not known in advance, require *Model-free Methods*. One of the Model-free techniques is *Q-Learning*. It is an off-policy RL algorithm that takes the best action given the current state using an action-value function, $Q(s_t, a_t)$, which approximates Q^*, the optimal action-value function.

The Q-learning rule is

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$$

where $(s_t, a_t, r_{t+1}, s_{t+1})$ is a tuple describing transition from current state s_t to next state s_{t+1} after performing action a_t and getting a reward r_{t+1}.

Algorithm 1 Q-Learning with ϵ-greedy exploration

```
1:  procedure Q-LEARNING($\epsilon, \alpha, \gamma$)
2:      Initialize $Q(s,a)$ for all $s \in S$, $a \in A$ arbitrarily except $Q(terminal, \cdot)$
3:      Initialize $\epsilon = 1$ and it is decay, $\varepsilon \in [0, 1)$
4:      $\pi \leftarrow \epsilon$-greedy policy w.r.t. $Q$
5:      for each episode do
6:          Set $s_1$ as the starting state
7:          $t \leftarrow 1$
8:          loop until episode terminates
9:              Select action $a_t = \begin{cases} \text{random action,} & \text{with probability } \epsilon \\ \max_a Q(s_t, a), & \text{otherwise} \end{cases}$
10:             Take action $a_t$ and observe reward $r_{t+1}$ and next state $s_{t+1}$
11:             $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
12:             $\pi \leftarrow \epsilon$-greedy policy w.r.t. $Q$ (policy improvement)
13:             $t \leftarrow t + 1$
14:         end loop
15:         $\epsilon \longleftarrow \varepsilon \times \epsilon$
16:     end for
17:     return $Q, \pi$
18: end procedure
```

Generalization

As the state and action space increases with the complexity of the problem, it becomes impractical and inefficient to store each transition in a table-like structure. Thus, we need to modify Q-learning to generalize approximation while storing the experience information in a more compact format. We can use Artificial Neural Networks (ANN) [6] for estimating Q values by using state and action as input. ANN enables automatic feature extraction, which allows us to successfully learn optimal policies directly from the high-dimensional observable state as input.

Batch RL [40] is a sub-domain of RL which uses dynamic programming for learning action-value functions. In general, an RL agent selects an action based on the current state of the environment using some policy, performs that action, observes the reward and new state of the environment, and updates its policy based on the reward. This process is very inefficient and converges very slowly to the optimal policy. We can overcome these deficiencies by using Batch RL, where the agent does not interact with the environment directly; instead, it receives a fixed sample of transitions $(s_j, a_j, r_j, s_{j+1}) \ \forall \ j \in \mathcal{N}$, from the environment prior to the learning. Now, the agent cannot make assumptions about the samples or the sampling process; therefore, the agent's objective changes from learning the optimal policy to learning the best policy for the given dataset. The action-value function is updated synchronously on the entire batch of transitions.

Deep RL [5] (DRL) is another sub-domain of RL that combines RL with deep learning which uses powerful function approximation and compact low dimensional representation of high dimensional state and action space using deep neural networks. Addressing the curse of dimensionality in RL allowed us to solve decision-making problems with autonomous agents, some of the prime examples are superhuman performances in games like Atari [7] and Go [28]. DRL algorithms can be categorized as *Value-Based* methods like DQN [8], NFQ [37], DDQN [41], Duel DQN [42], *Policy-Based* methods like Vanilla Policy Gradient (VPG) [43], Trusted Region Policy Optimization (TRPO) [44], Proximal Policy Optimization (PPO) [45], and *Actor-Critic* methods like Deterministic Policy Gradient (DPG) [46], Deep DPG (DDPG) [27], and Asynchronous Advantage Actor-Crtic (A3C) algorithm [26].

We will use Value-Based DRL techniques, which incorporate the concept of Batch RL. We will also explore the effects of batch size and sampling heuristics on policy optimization using IB in Chapter 4.

In this work, Q-learning based techniques are used for approximating the optimal action-value function to get the best policy. We have also used Double Q-learning to overcome the limitation of maximization bias in the Q-learning. As IB has large and continuous state and action values, we will use ANN as function approximators in DQN and DDQN.

3.1.1 DQN

Deep Q-Networks [8] is an RL technique based on Q-learning and motivated by the success of deep neural networks. It uses temporal difference to update the neural network, directly from sample of transitions $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ generated from the interactions with the environment. DQN uses *experience replay buffer* to store the transitions e_t's performed by the agent over multiple episodes in a data-set $\mathcal{D}$. The agent uses ϵ-greedy policy to select an action, and then it stores the transition in the replay buffer. The complete algorithm of DQN is described in Algorithm 2.

The advantages of using DQN over Q-learning are data efficiency as we use the same transitions multiple times during training. Randomly selected samples reduce the correlation between the samples and therefore reduces variance, and the value function approximator is more stable because of the experience replay.

Algorithm 2 Deep Q-Learning

```
1: Initialize replay memory $\mathcal{D}$ with a fixed capacity
2: Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
3: Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$
4: for episode $m = 1, ..., M$ do
5:     Reset to initial state $s_1$
6:     for time step $t = 1, ..., T$ do
7:         Select action $a_t = \begin{cases} \text{random action,} & \text{with probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}^-), & \text{otherwise} \end{cases}$
8:         Take action $a_t$ and observe reward $r_{t+1}$ and next state $s_{t+1}$
9:         Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $D$
10:        Sample uniformly a random minibatch of N transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
11:        if episode ends at step $j + 1$ then
12:            $y_j = r_{j+1}$
13:        else
14:            $y_j = r_{j+1} + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}^-)$
15:        end if
16:        Perform a stochastic gradient descent step on $J(\mathbf{w}) = \frac{1}{N}\sum_{j=1}^{N}(y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$
   w.r.t. parameters $\mathbf{w}$
17:        Every $C$ steps reset $\mathbf{w}^- = \mathbf{w}$
18:    end for
19: end for
```

3.1.2 Double-DQN

One of the significant drawbacks of Q-learning is that the estimated values can suffer from maximization bias because it uses the maximum over the estimated values as an estimate of the maximum value, as seen in line 11 Algorithm 1 and lines 7 and 14 in Algorithm 2. In order to avoid this bias, we need to decouple maximizing the Q value and estimating the max Q value for a given state, and this can be achieved by using two independent unbiased estimates, Q_1 and Q_2. We can use one to select the maximum value with some probability p and the other to estimate the value of the maximum with probability $1 - p$. This method is known as Double Q-learning [41], which is described in Algorithm 3.

Algorithm 3 Double Q-Learning

```
1:  procedure DOUBLE Q-LEARNING($\epsilon, \alpha, \gamma$)
2:      Initialize $Q_1(s, a), Q_2(s, a)$ for all $s \in S$, $a \in A$, set $t \leftarrow 0$
3:      $\pi \leftarrow \epsilon$-greedy policy w.r.t. $Q_1 + Q_2$
4:      loop
5:          Sample action $a_t$ from policy $\pi(s_t)$
6:          Take action $a_t$ and observe reward $r_{t+1}$ and next state $s_{t+1}$
7:          if $u \sim U(0, 1) > 0.5$ then
8:              $Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1}, a')) - Q_1(s_t, a_t))$
9:          else
10:             $Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1}, a')) - Q_2(s_t, a_t))$
11:         end if
12:         $\pi \leftarrow \epsilon$-greedy policy w.r.t. $Q_1 + Q_2$ (policy improvement)
13:         $t \leftarrow t + 1$
14:     end loop
15:     return $\pi, Q_1 + Q_2$
16: end procedure
```

Double Q-learning can eliminate sub-optimal actions more quickly than normal Q-learning and thus reduces the training timing significantly.

DQN also suffers from maximization bias as it uses the same network values for selecting and estimating the action, as seen in line 14 Algorithm 2. The same idea of decoupling action selection and action estimation can also be applied to DDQN. This can be achieved by using the target network in the DQN as a second value function. Van Hasselt, et al., [9] proposed to evaluate the greedy policy using the online network and estimate the value using the target network and keep the same update policy as shown in Algorithm 4.

Algorithm 4 Double Deep Q-Learning

```
1: Initialize replay memory $\mathcal{D}$ with a fixed capacity
2: Initialize action-value function $\hat{q}$ with random weights w
3: Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$
4: for episode $m = 1, ..., M$ do
5:     Reset to initial state $s_1$
6:     for time step $t = 1, ..., T$ do
7:         Select action $a_t = \begin{cases} \text{random action,} & \text{with probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}), & \text{otherwise} \end{cases}$
8:         Take action $a_t$ and observe reward $r_{t+1}$ and next state $s_{t+1}$
9:         Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
10:        Sample uniformly a random minibatch of N transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
11:        if episode ends at step $j + 1$ then
12:            $y_j = r_{j+1}$
13:        else
14:            $y_j = r_{j+1} + \gamma \hat{q}(s_{j+1}, \arg\max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}), \mathbf{w}^-)$
15:        end if
16:        Perform a stochastic gradient descent step on $J(\mathbf{w}) = \frac{1}{N}\sum_{j=1}^{N}(y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$
    w.r.t. parameters $\mathbf{w}$
17:        Every $C$ steps reset $\mathbf{w}^- = \mathbf{w}$
18:    end for
19: end for
```

3.2 Supplement or Additional Features

3.2.1 Experience Replay

Experience replay [21] has had a significant impact on the DRL algorithms and has become standard in most RL algorithms. The main reason for its popularity is that it generates uncorrelated data for online learning of DRL techniques while improving the data efficiency as the same transitions get utilized multiple times. Another approach for generating uncorrelated data is to use multiple workers, but that changes the problem settings. In this work, we will be using uniform and prioritized experience replay buffers.

Uniform Replay Buffer

A uniform replay buffer is a queue of the last N transitions stored in memory. It is used to draw a set of sample transitions using a uniform random distribution where each transition is equally likely to get picked.

Prioritized Replay Buffer

Even though uniform experience replay improves the performance and data efficiency of the DRL algorithms, it replays the transitions at the same frequency as before without considering their significance. Schaul, et al., [47] developed an algorithm (refer to Algorithm 5) to prioritize experiences so that we can replay important experiences more frequently. This method is more effective and efficient than the uniform replay. We will be using this same algorithm for our experiments.

The prioritization of experiences is achieved by replaying transitions with high expected learning, measured in terms of the magnitude of temporal-difference error, more frequently. This introduces bias due to stochastic prioritization which is corrected using *importance sampling* (IS) [48] weights

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}$$

We normalize the IS weights by $1/\max_i w_i$ for stability reasons. So the normalized IS weights is

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta} \frac{1}{\max_i w_i}$$

These normalized IS weights can now be used in Q-learning along with the TD-error as shown in line 13 Algorithm 5.

Algorithm 5 Double DQN with proportional prioritization

```
 1: Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $n$, exponents $\alpha$ and $\beta$, budget $T$.
 2: Initialize replay memory $\mathcal{D} = \emptyset$, $\Delta = 0$, $p_1 = 1$
 3: Observe $s_1$ and choose $a_1 \sim \pi_\theta(s_1)$
 4: for $t = 1, ..., T$ do
 5:     Observe $s_{t+1}, r_{t+1}, \gamma_{t+1}$
 6:     Store transition $(s_t, a_t, r_{t+1}, \gamma_{t+1}, s_{t+1})$ in $\mathcal{D}$ with maximal priority $p_t = \max_{i<t} p_i$
 7:     if $t$ mod $K = 0$ then
 8:         for $j = 1$ to $k$ do
 9:             Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:             Compute importance-sampling weight $w_j = (NP(j))^{-\beta} / \max_i w_i$
11:             Compute TD-error $\delta_j = r_j + \gamma_j Q_{target}(s_j, \arg\max_a Q(s_j, a)) - Q(s_{j-1}, a_{j-1})$
12:             Update transition priority $p_j \longleftarrow |\delta_j|$
13:             Accumulate weight-change $\Delta \longleftarrow \Delta + w_j \delta_j \nabla_\theta Q(s_{j-1}, a_{j-1})$
14:         end for
15:         Update weight $\theta \longleftarrow \theta + \eta\Delta$, reset $\Delta = 0$
16:         From time to time copy weights into target network $\theta_{target} \longleftarrow \theta$
17:     end if
18:     Choose action $a_t \sim \pi_\theta(s_t)$
19: end for
```

3.2.2 Optimizer

Optimization techniques [20] are the core component of all machine learning algorithms. As all learning algorithms attempt to build an optimal model to learn a specific objective function from a large source labeled/unlabeled data, optimization techniques are responsible for the success or failure of these algorithms. There are three main categories of optimization techniques: first-order optimization techniques based on gradient descent, high-order optimization techniques based on Newton's method, and heuristic derivative-free optimization techniques based on coordinate descent.

Gradient descent based methods like Stochastic Gradient Descent (SGD) [49], and Nesterov Accelerated Gradient Descent (NAG) [50], and adaptive learning rate methods like AdaGrad [51], AdaDelta [52] or RMSProp [53], and Adam [54] are some of the commonly used optimizers in DRL problems.

Gradient Descent techniques iteratively update the variables in the opposite direction of the gradients of the objective function. The update helps the objective function to converge to the optimal value. This approach guarantees global optima if the objective function is a convex problem. It often takes longer to converge when the variables are closer to the optimal values. In this work, we will use SGD and Adam for optimizing our DRL models.

SGD

The Stochastic Gradient Descent (SGD) [49] is an unbiased estimate of the real gradient and is computed using a randomly selected sample to update the gradient per iteration, instead of using all the samples at every iteration. Thus, the SGD is independent of the sample size and can achieve sub-linear convergence speed. It can also be used for online learning methods. However, the random selection causes the gradient direction to oscillate, which is a major drawback. This challenge can be easily handled by using a mini-batch of 50-256 independent identically distributed samples instead of using a single sample. This reduces the variance in the gradient, stabilizes the convergence, and improves the optimization speed.

The learning rate is an important hyper-parameter for SGD that needs to be defined manually. A small learning rate will significantly slow down the convergence, and a large learning rate will overshoot and oscillate at the optima. We can use Adam to overcome this challenge that uses adaptive learning.

Adam

Adaptive moment estimation (Adam) [54] is an advanced gradient descent method that uses an adaptive learning rate in combination with momentum-based methods. It dynamically adjusts the learning rate of each parameter based on a historical gradient from the previous time steps. The update is formulated as follows:

The exponential decaying average of gradients m_t is

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.1}$$

where g_t is the gradient of parameter θ at time t. It also uses exponential moving average to calculate the second-order cumulative momentum,

$$V_t = \sqrt{\beta_2 V_{t-1} + (1 - \beta_2)(g_t)^2} \tag{3.2}$$

where β_1, β_2 are exponential decay rates. The final update to the parameter θ is

$$\theta_{t+1} = m_t + \eta \frac{\sqrt{1 - \beta_2}}{1 - \beta_1} \frac{m_t}{V_t + \varepsilon} \tag{3.3}$$

where η is the initial learning rate. The default values of β_1, β_2 and ε are 0.9, 0.999 and 10^{-8}.

Adam is suitable for all problems with high dimensional state and action space, and it is also the most stable optimizer.

Chapter 4

Evaluation And Results

4.1 Experimental Setup

We performed several experiments using DQN and DDQN for policy optimization on IB. These experiments were performed on a single computer powered by 4-core Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz with 32GB of RAM and NVIDIA GeForce GTX 1050 Ti Mobile. We have used Tensorflow [22], Keras [23], Keras-RL [24] for implementing our DRL models with ANN [6] as function approximator with ReLU [55] activation function, and IB OpenAI Gym wrapper for generating IB data.

We use DDQN as our default policy optimization technique. The default values of hyper-parameters across all our experiments are as follows:

- Discount Factor $\gamma = 0.99$
- Replay Memory Size = 50,000
- Minimum Replay Memory Size = 1000
- Mini-batch Size = 64
- Update Target Model Every = 5 Episode
- Number of Episodes = 1000

The default settings for exploration are as follows:

- Epsilon $\epsilon = 1$
- Epsilon Decay $\varepsilon = 0.99$
- Minimum Epsilon = 0.001

The default settings for IB are

- Setpoint = 100
- Setpoint Type = Constant
- Reward Type = Classic
- Action Type = Discrete
- Time Horizon H = 1000 (number of steps per episode)

The default configuration of ANN used is as follows:

- Input Layer = Observable state
- First Hidden Layer = Dense Layer with 124 units
- Second Hidden Layer = Dense Layer with 56 units
- Activation Function for Hidden Layers = ReLU
- Output Layer = Dense Layer with 27 (Number of unique discrete actions) units
- Activation Function for Output Layer = Linear
- Optimizer = Adam
- Learning Rate $\eta = 0.001$
- Loss Function = Mean Squared Error

We will specify all the required hyper-parameters in each experiment where we do not use the default settings.

Figure 4.1 shows continuous input actions (a_1, a_2, a_3), and the corresponding outputs, the cost and the reward as a function of time steps over three independent runs. The reward function is just the negative of the cost function. For practical purposes, we have reduced the magnitude of the reward by a factor of 100 to reduce high variance due to large reward values. Figure 4.2 shows the other observable variables (v, g, h, p, c, f) of the state space of the IB corresponding to the same actions shown in Figure 4.1, where v is velocity, g is gain, h is shift, p is setpoint, c is consumption, and f is fatigue.

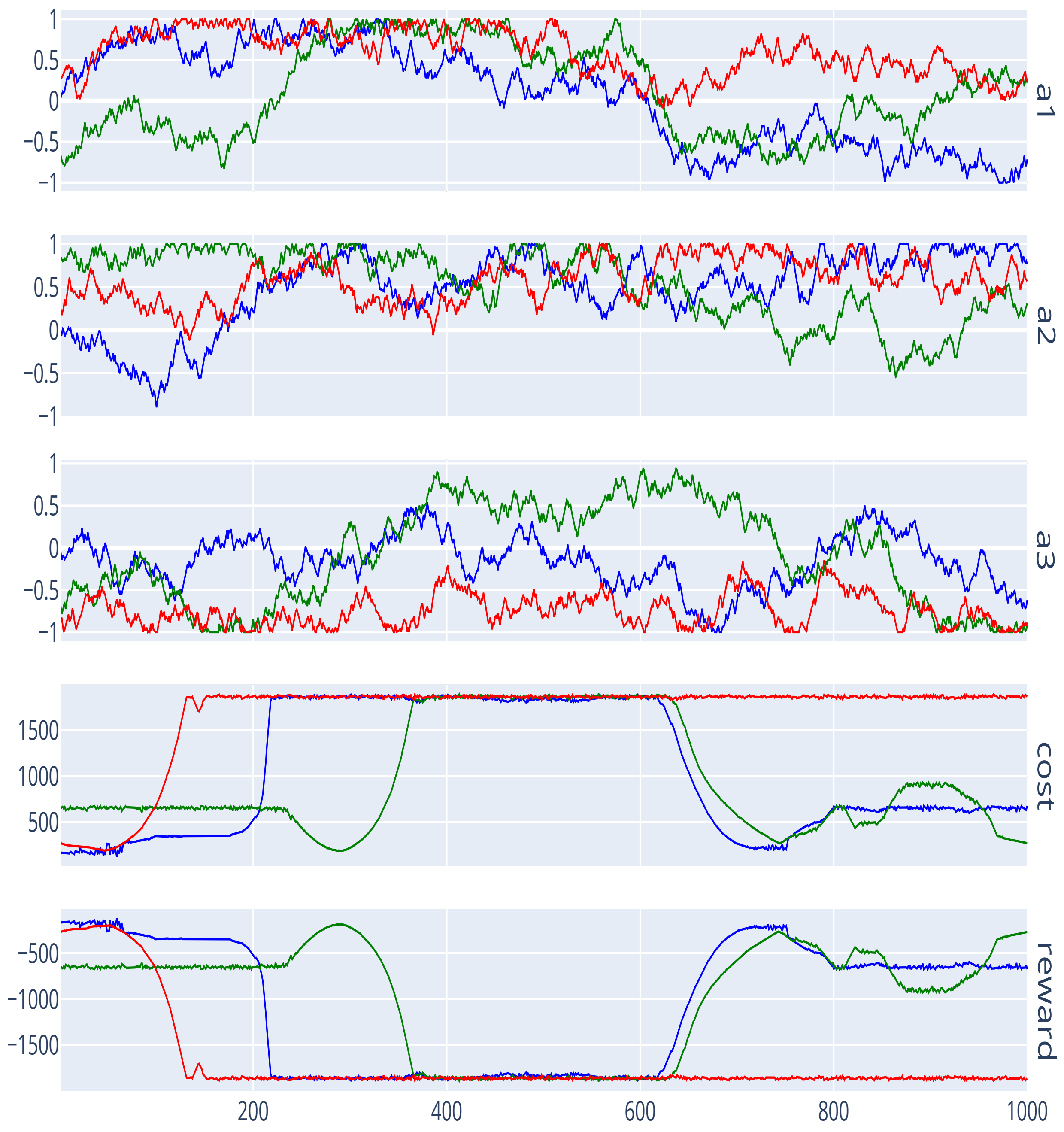

Figure 4.1: Action space and Reward as a function of time steps over 3 independent runs

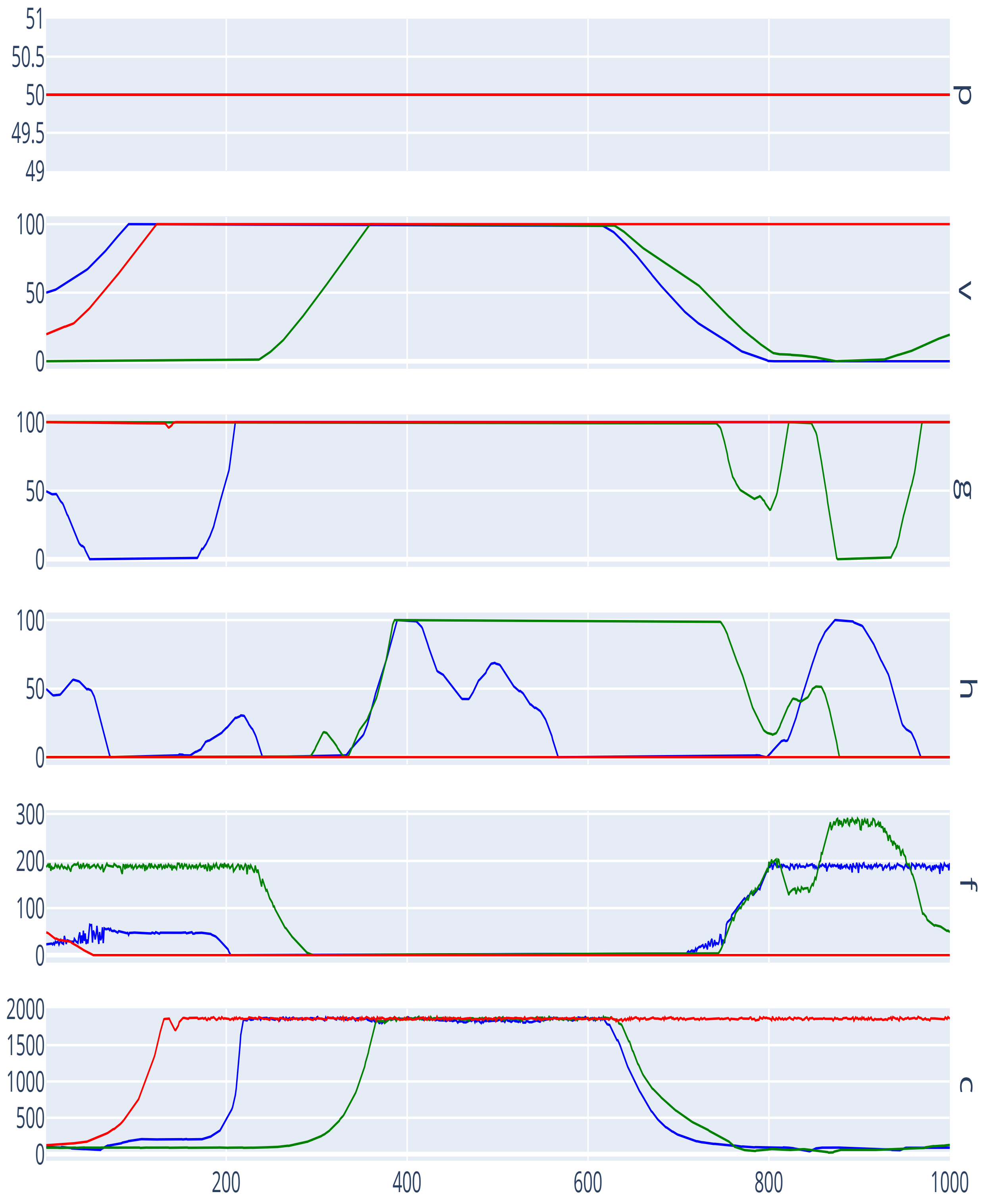

Figure 4.2: Observable state space as a function of time steps over 3 independent runs

4.2 Empirical Results

This section presents the results from the numerous experiments performed using IB. Each subsection contains a plot of *Rewards per Episode Vs. Episode Number* and a corresponding table is summarizing the statistics for each run. We compare the different models based on the fitness value, which is calculated using equation (2.3). The model with a bigger (smaller magnitude) fitness value is a better model. The plots for all the experiments present the rewards per episode of a single best run for some hyper-parameter and average over five runs for others. We have summarized the statistics of the best models from all the experiments at the end in Section 4.2.10.

4.2.1 Setpoint

As mentioned earlier in Section 2.4, Setpoint p represents the external load that cannot be influenced by the actions. In this experiment, we are trying to understand and visualize the differences between setpoints. Figure 4.3 shows the rewards per episode for ten different setpoints, which decrease in value (increase in cost) with an increasing setpoint. It also signifies the complexity of the control system that we are trying to simulate. An IB environment with setpoint $p = 10$ is a very easy environment to model compared to an environment with setpoint $p = 100$. We have summarized the results from all the setpoints in Table 4.1.

Similarly, Figure 4.4 shows the rewards per episode for variable setpoint. A variable setpoint changes the external load in a heteroscedastic manner. It makes the IB environment even more complex and difficult to model. The results are summarized in Table 4.2. The results for variable setpoints were recorded over 1000 episodes compared to 100 in the fixed setpoint to capture the variability.

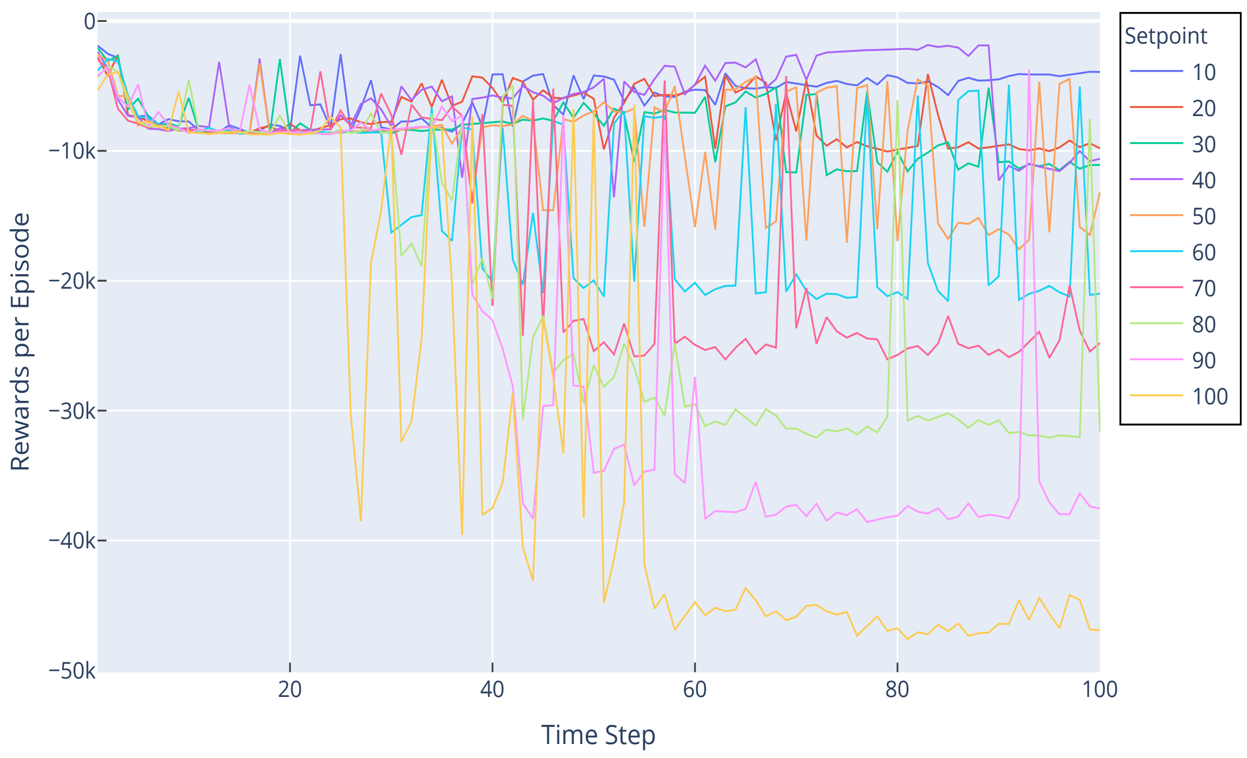

Figure 4.3: Rewards per Episode for Setpoint P=(10, 20,...,100)

Table 4.1: Summary of Sum of All Rewards for Fixed Setpoint

Setpoint	Mean	Std. Dev	Min	Max
10	-5735.14	1744.13	-8598.07	-1898.13
20	-7367.93	2037.27	-10054.58	-2615.39
30	-8362.81	2201.67	-11873.94	-2073.98
40	-5956.25	2995.14	-13568.36	-1847.97
50	-9397.29	4184.45	-17582.52	-2550.22
60	-13769.28	6522.78	-21567.60	-2919.22
70	-16872.53	8679.07	-26061.33	-2965.23
80	-20888.54	10805.02	-32091.53	-3461.00
90	-24217.23	14097.87	-38581.09	-3581.38
100	-30718.85	17172.2	-47592.58	-3909.40

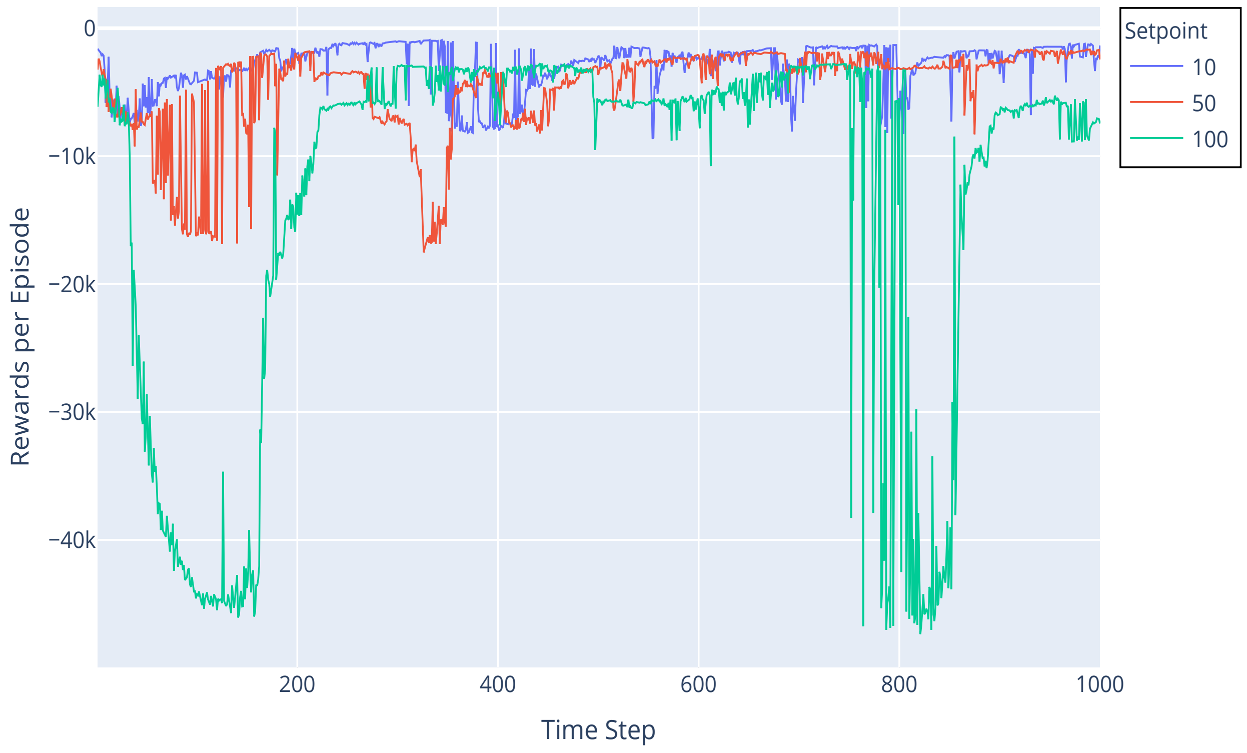

Figure 4.4: Rewards per Episode for Variable Setpoint P=(10, 50, 100)

Table 4.2: Summary of Sum of All Rewards for Variable Setpoint

Setpoint	Mean	Std. Dev	Min	Max
10	-2973.36	1935.64	-8638.81	-918.55
50	-4607.81	3620.18	-17524.25	-1647.63
100	-12735.16	14406.51	-47367.28	-2743.87

4.2.2 Mini-batch Size

The mini-batch size determines the number of samples that will be drawn from the replay buffer for training and optimizing the policy. We have analyzed its effect on both uniform and prioritized replay buffers. Figure 4.5 shows the average rewards per episode across three different mini-batch sizes for DDQN with uniform replay buffer (URB). We have summarized the results in Table 4.3, and we can see that mini-batch of 128 samples is better suited for DDQN with uniform replay buffer.

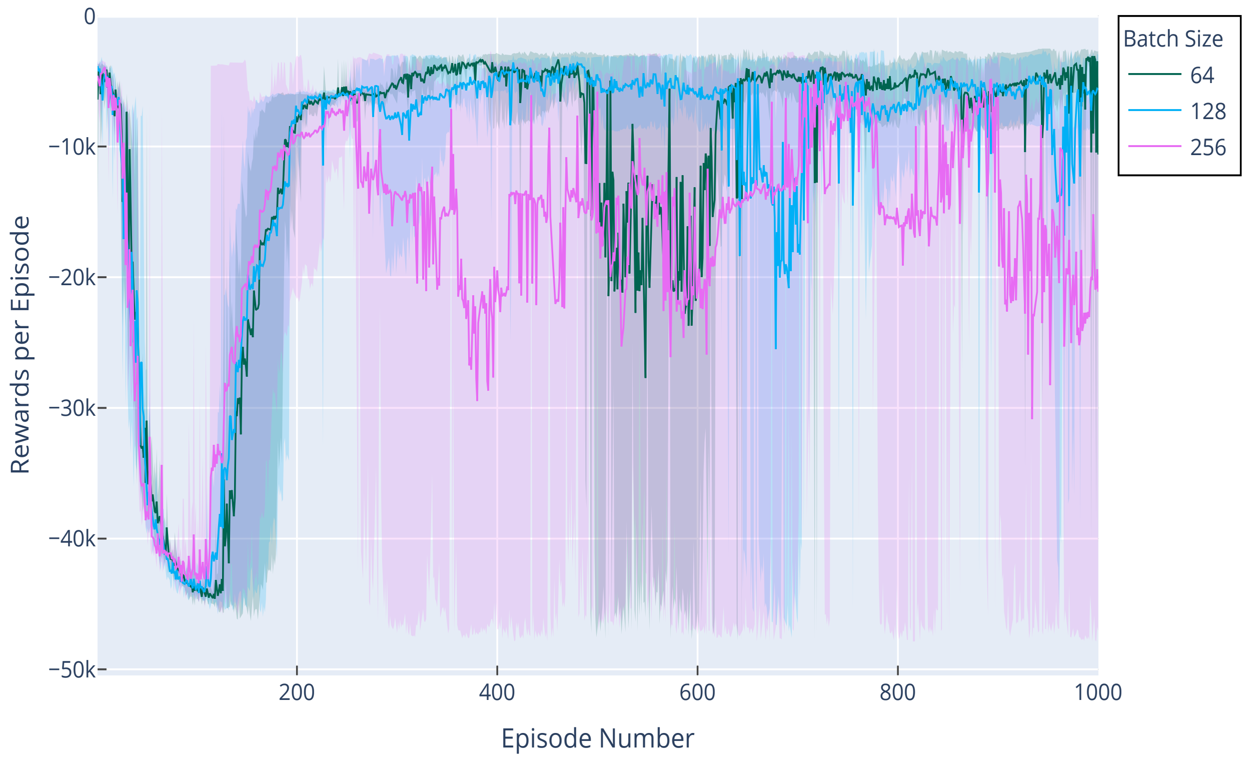

Figure 4.5: Average Rewards per Episode for DDQN with URB: Setpoint P=100, Mini-batch=(64, 128, 256)

Table 4.3: Summary of Sum of All Rewards Vs. Mini-batch Size

Setpoint	Batch Size	Mean	Std. Dev.	Min	Max
100	64	-11032.31	11156.79	-44600.57	-3074.04
100	128	**-10741.57**	10458.93	-44158.97	-3570.59
100	256	-16710.54	8801.76	-43425.13	-3921.79

Similarly, Figure 4.6 shows the average rewards per episode for DDQN with a prioritized replay buffer (PRB). We have summarized the results from prioritized replay buffer in Table 4.4, and we can see that a mini-batch of 64 samples outperforms others. In this experiment, we get better performance from uniform replay than prioritized replay in general.

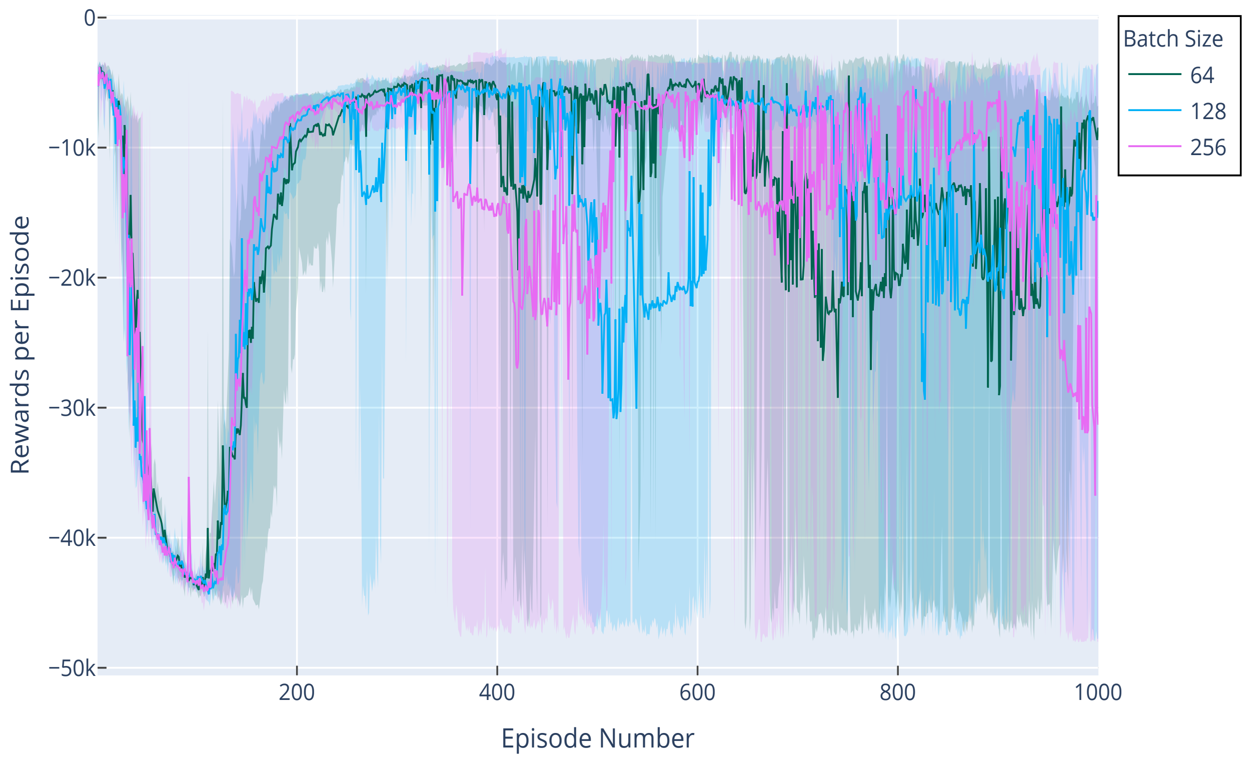

Figure 4.6: Average Rewards per Episode for DDQN with PRB: Setpoint P=100, Mini-batch=(64, 128, 256)

Table 4.4: Summary of Rewards Sum of All Rewards with Prioritized Replay Vs. Mini-batch Size

Setpoint	Batch Size	Mean	Std. Dev.	Min	Max
100	64	**-14214.56**	10299.48	-44012.59	-3819.32
100	128	-14870.42	10438.81	-44344.08	-3918.77
100	256	-14659.20	10415.07	-44150.09	-3803.85

4.2.3 SGD Optimizer

Section 3.2.2 gives an overview of the optimizer [20] and the numerous options [49] [54] [51] [52] [53] that we have at our disposal. In general, Adam [54] is considered the best optimizer for DRL problems; therefore, we have used it as our default optimizer. In this experiment, we are using SGD [49] as optimizer with momentum $m = 0.9$, and learning rates $\eta = 0.001$, and $\eta = 0.0001$, as shown in Figure 4.7 and Figure 4.8, respectively. The DDQN models are used to understand the effects of mini-batch size when combined with SGD, and we have summarized the results in Table 4.5 and Table 4.6.

In general, the models with learning rate $\eta = 0.001$ outperforms all the models with learning rate $\eta = 0.0001$, and the best model among the better models is the one with mini-batch of 512 samples.

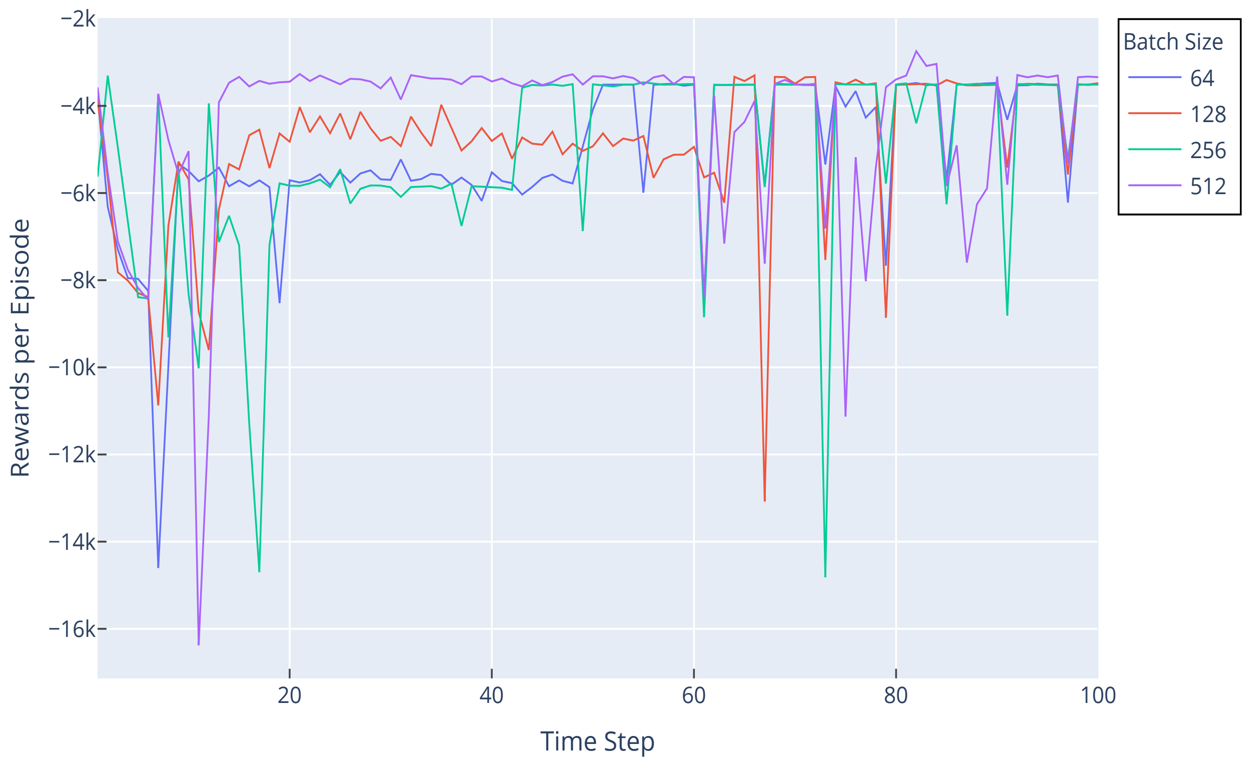

Figure 4.7: Rewards per Episode for DDQN using SGD optimizer and URB: Setpoint P=100, Learning Rate $\eta = 0.001$, Mini-batch=(64, 128, 256, 512)

Table 4.5: Summary of Sum of All Rewards using SDG with $\eta = 0.001$ Vs. Mini-batch Size

Setpoint	Batch Size	Mean	Std. Dev.	Min	Max
100	64	-5038.18	1724.48	-14603.27	-3471.33
100	128	-4886.08	1705.72	-13077.59	-3299.84
100	256	-5124.71	2239.50	-14816.15	-3305.11
100	512	**-4395.18**	2095.30	-16378.02	-2744.79

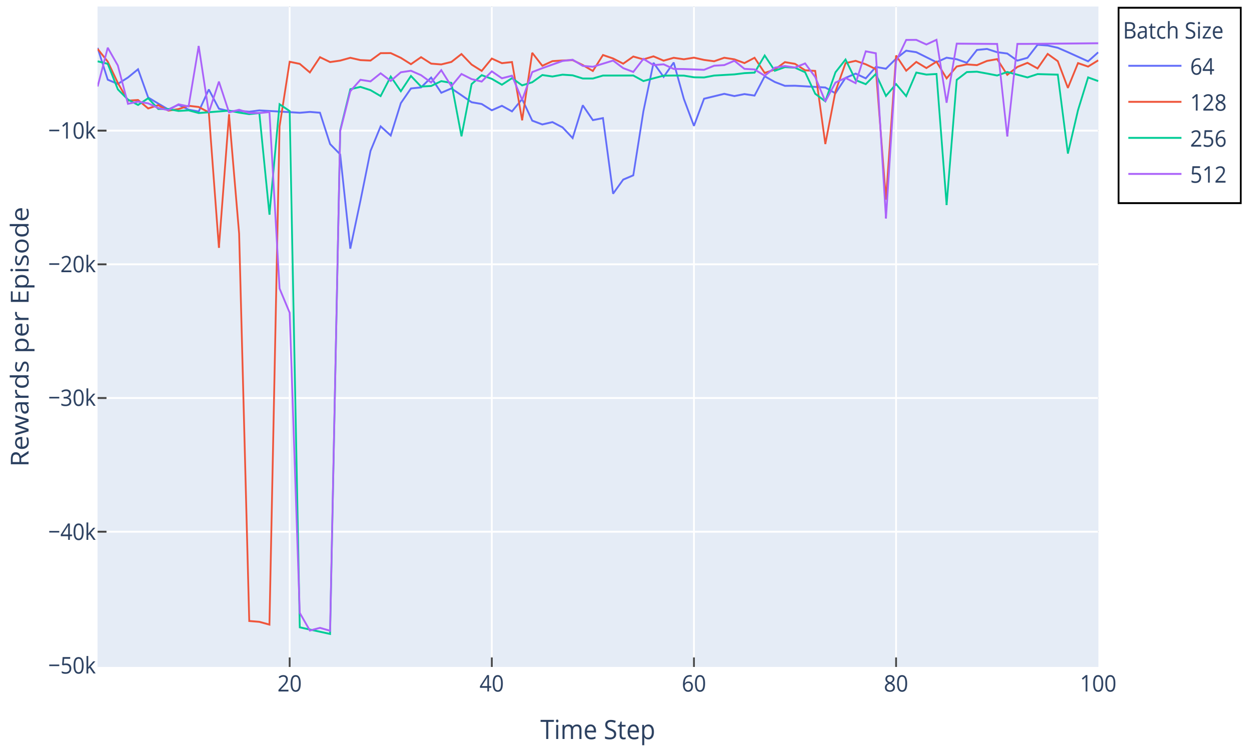

Figure 4.8: Rewards per Episode for DDQN using SGD optimizer and URB: Setpoint P=100, Learning Rate $\eta = 0.0001$, Mini-batch=(64, 128, 256, 512)

Table 4.6: Summary of Sum of All Rewards using SDG $\eta = 0.0001$ Vs. Mini-batch Size

Setpoint	Batch Size	Mean	Std. Dev.	Min	Max
100	64	-7383.91	2679.34	-18823.70	-3576.57
100	128	**-7057.65**	7437.44	-46946.59	-3887.79
100	256	-8449.20	8193.44	-47644.25	-4379.80
100	512	-7765.93	8627.84	-47407.39	-3206.80

4.2.4 ANN Architectures

In Section 3.1, we mentioned the need for generalization, and we also mentioned that we would use ANN to achieve that. In this experiment, we will explore the effects of the number of hidden dense layers in ANN on the performance of DDQN models. A model with zero dense layers is a linear model. The number of units in the first, second, and third dense layers is 124, 56, and 124, respectively, and they were selected randomly.

Figure 4.9 shows the average reward per episode for DDQN with uniform replay buffer (URB) models across several dense layers. Figure 4.10 shows the same for DDQN models with a prioritized replay buffer (PRB).

The results from each set of models are summarized in Table 4.7 and Table 4.8. In general, the model with more dense layers performs better in both settings. The models with uniform replay have outperformed their counterparts with prioritized replay. Based on these results, we can claim that ANN with uniform replay performs better than the same ANN with prioritized replay.

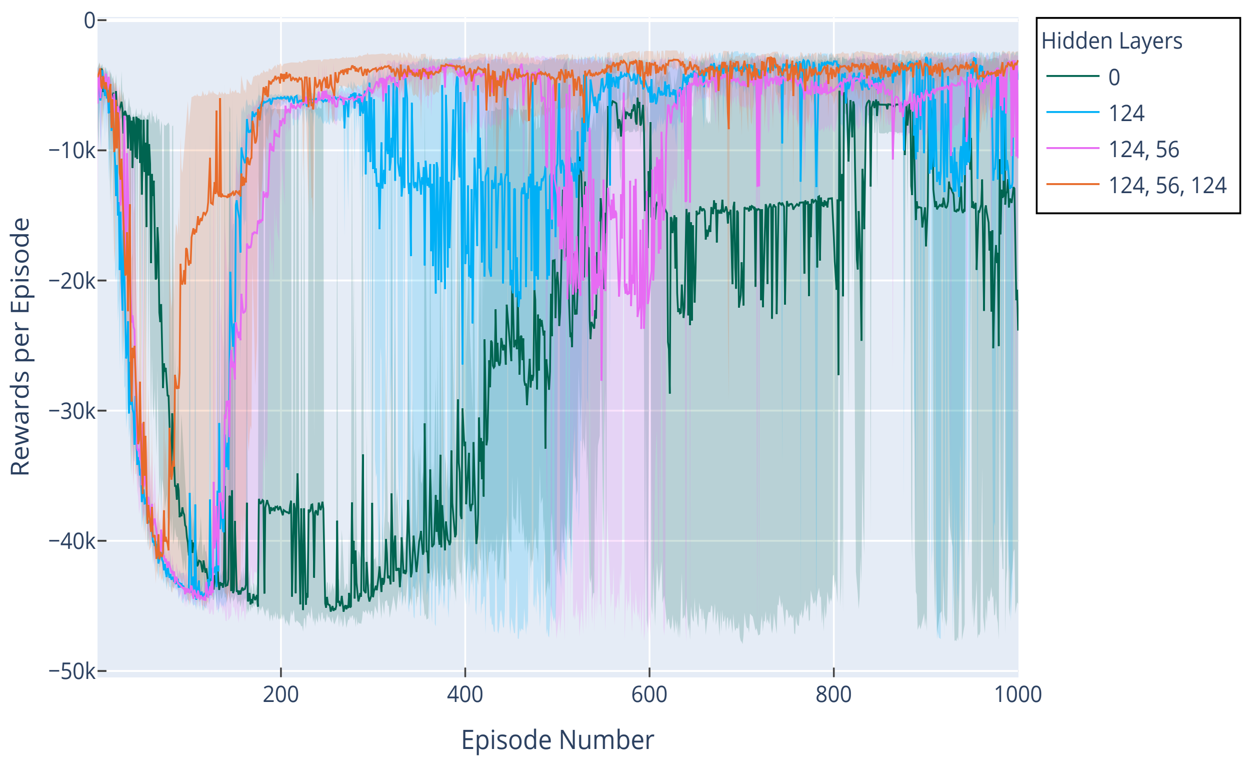

Figure 4.9: Average Rewards per Episode for DDQN with URB: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.7: Summary of Sum of All Rewards for DDQN Vs. Number of Dense Layers

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-23655.09	13246.99	-45443.12	-3723.10
100	1	-11430.57	11065.34	-44529.65	-2838.93
100	2	-11032.31	11156.79	-44600.57	-3074.04
100	3	**-6686.92**	7524.28	-41407.52	-2880.93

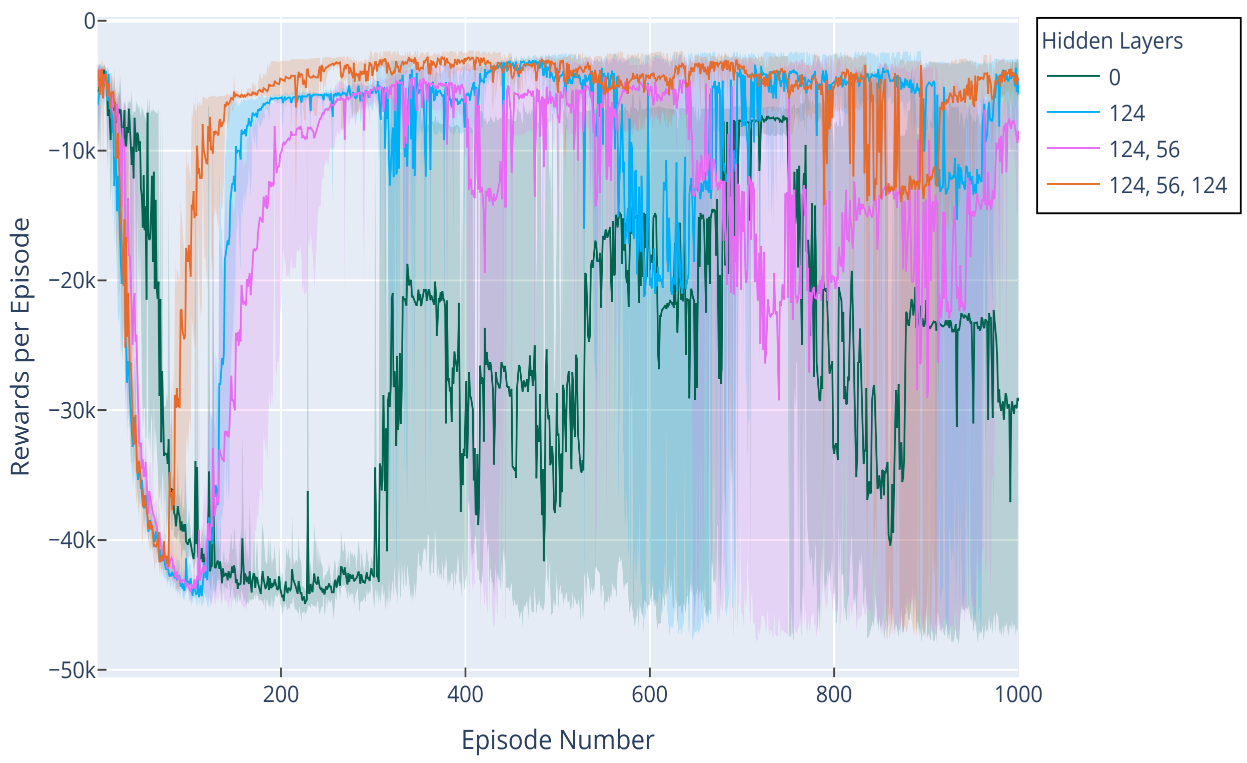

Figure 4.10: Average Rewards per Episode for DDQN with PRB: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.8: Summary of Sum of All Rewards for DDQN with PRB Vs. Number of Dense Layers

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-26904.02	11240.68	-44917.72	-3779.61
100	1	-10736.12	10714.12	-44410.98	-3042.04
100	2	-14214.56	10299.48	-44012.59	-3819.32
100	3	**-7202.13**	7910.00	-42092.31	-2770.99

4.2.5 DQN

We have mostly used DDQN in all the experiments for reasons mentioned in Section 3.1. In this experiment, we will use DQN to learn the policy for the IB problem. DQN uses the same Q-net to select the best action and to estimate the maximum value function for that action. In theory, DQN models will suffer from maximization bias and will be unstable to model problems like IB. The same hypothesis can be confirmed from Figure 4.11 and Figure 4.12. The results for DQN models are summarized in Table 4.9 and Table 4.10.

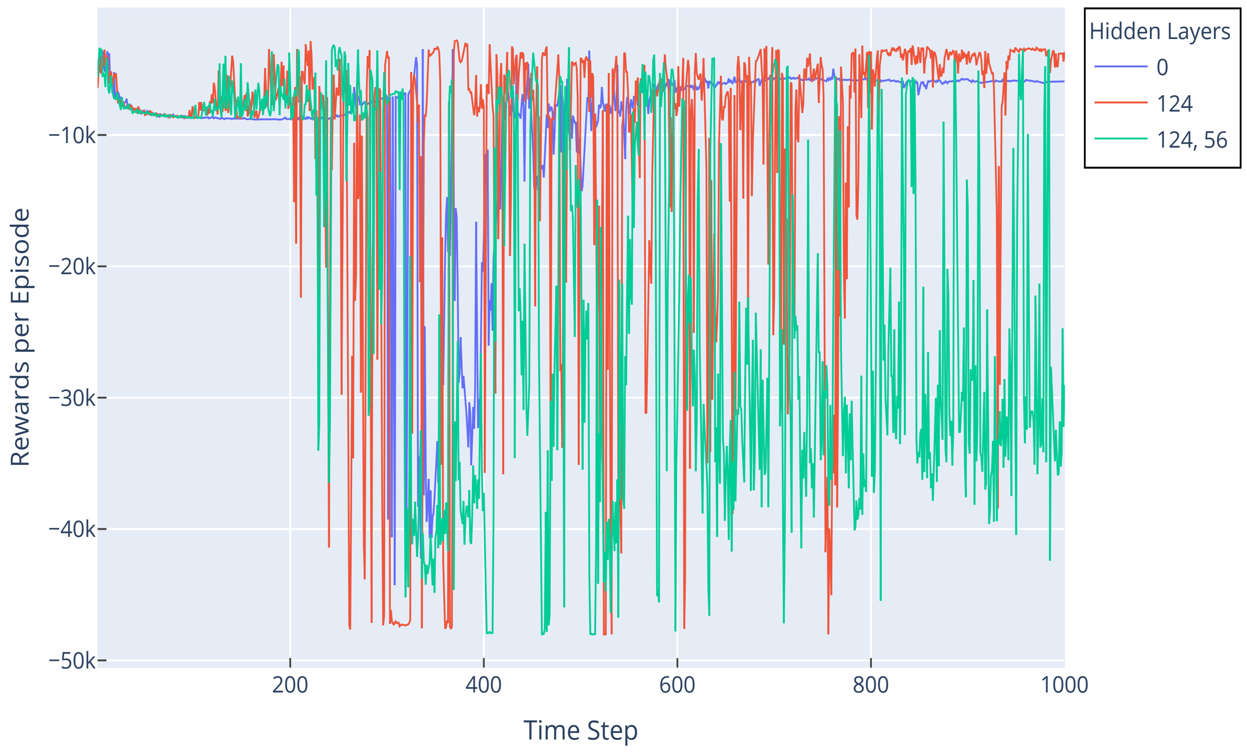

Figure 4.11: Rewards per Episode for DQN with URB: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56)

Table 4.9: Summary of Sum of All Rewards for DQN Vs. Number of Dense Layers

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	**-8979.82**	6576.28	-44268.75	-3461.81
100	1	-10613.48	11078.02	-48034.36	-2800.96
100	2	-21168.30	13957.35	-48024.85	-3147.80

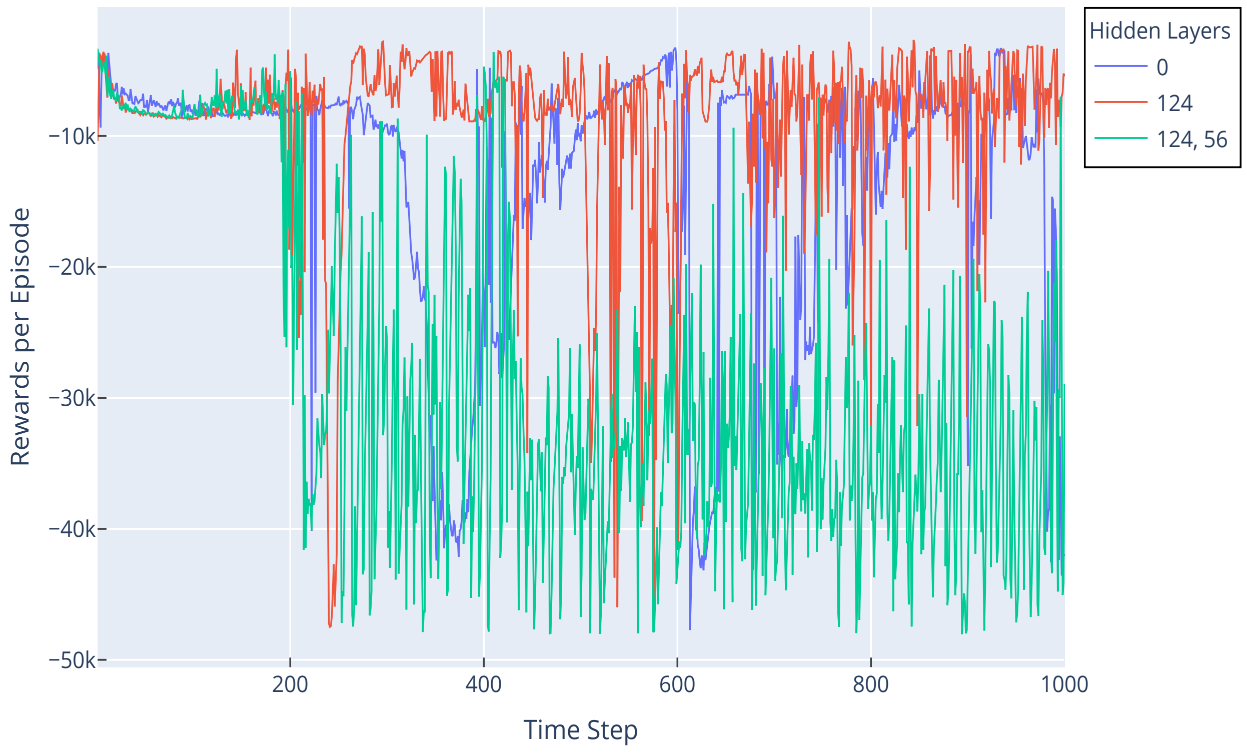

Figure 4.12: Rewards per Episode for DQN with PRB: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56)

Table 4.10: Summary of Sum of All Rewards for DQN with PRB Vs. Number of Dense Layers

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-13293.53	10441.78	-47723.53	-3258.22
100	1	**-8657.39**	6710.42	-47483.94	-2669.50
100	2	-28215.85	13179.32	-48036.62	-3332.44

4.2.6 DDQN with Target Model updated every episode

In all the DDQN models that have seen so far, we were updating the target model every five episodes to ensure stability. We suspect that the target model is not able to keep up with the variations in the environment. Therefore, in this experiment, we decided to update the target model in every episode. Figure 4.13 and Figure 4.14, shows the average reward per episode for uniform and prioritized replay respectively. The results are summarized in Table 4.11 and Table 4.12, and we can see the significant improvement in results compared to the previous results.

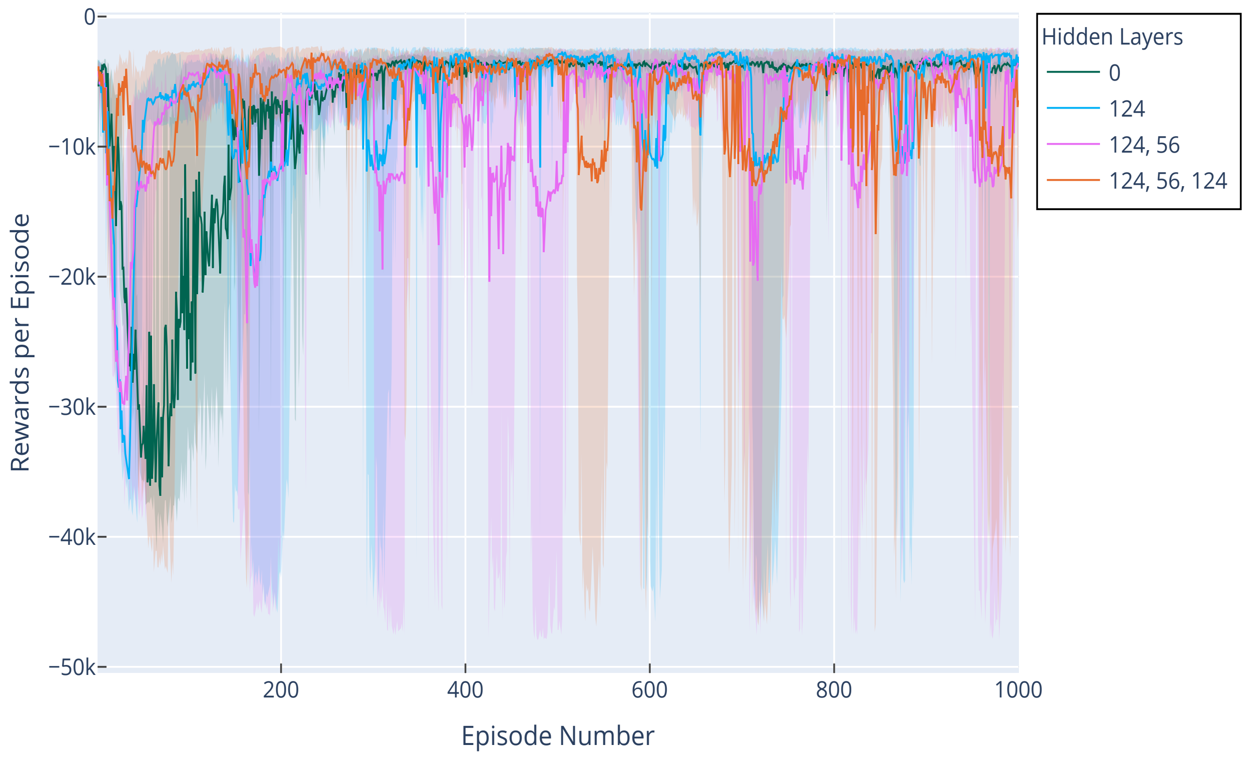

Figure 4.13: Average Rewards per Episode for DDQN with URB and Target Model updated every episode: Setpoints P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.11: Summary of Sum of All Rewards for DDQN with Target Model updated every episode

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-6704.16	6860.30	-36848.46	-3153.34
100	1	**-6032.01**	4859.15	-35556.74	-2667.84
100	2	-7984.18	4881.11	-29846.85	-2873.51
100	3	-6055.18	3034.30	-16727.48	-2786.02

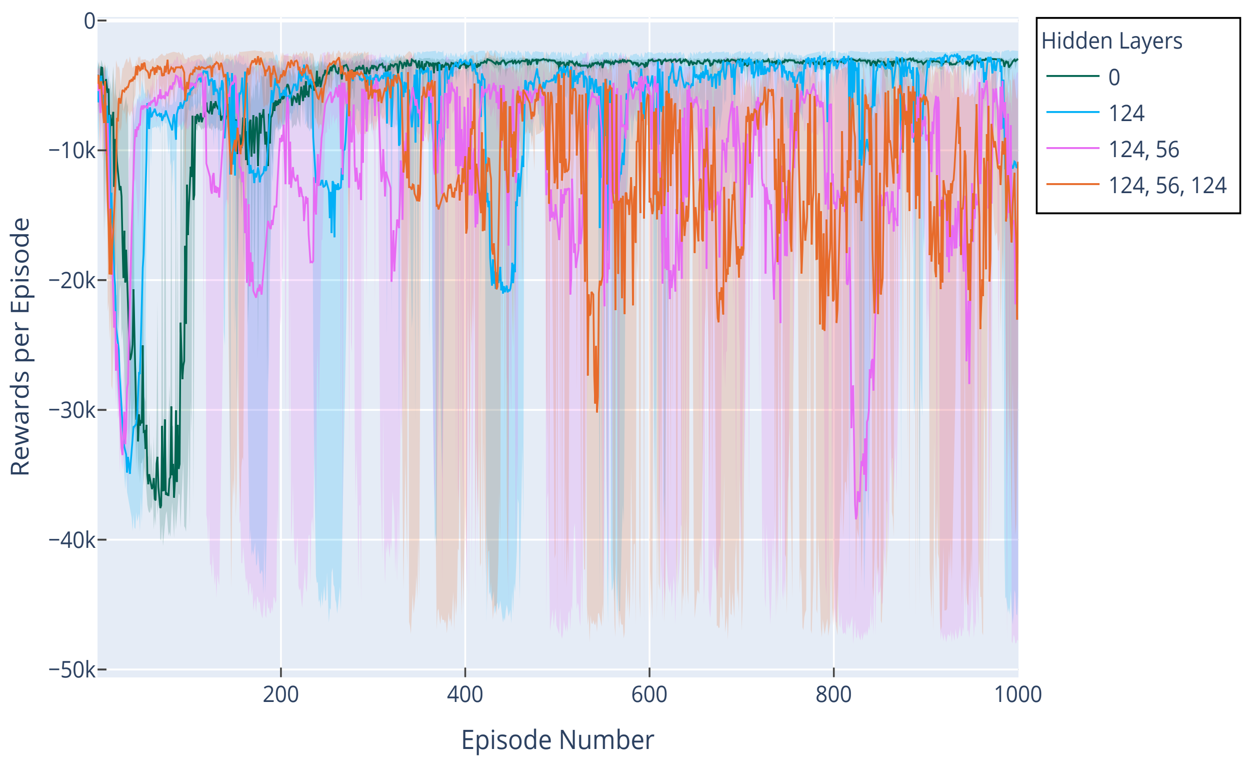

Figure 4.14: Average Rewards per Episode for DDQN with PRB and Target Model updated every episode: Setpoints P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.12: Summary of Sum of All Rewards for DDQN with PRB and Target Model updated every episode

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	**-5906.40**	7186.25	-37560.60	-2930.66
100	1	-6873.84	5571.45	-34928.66	-2545.41
100	2	-11265.57	6346.57	-38431.29	-4036.28
100	3	-9660.24	5554.69	-30199.65	-2746.15

4.2.7 DDQN with Target Model updated every other episode

Similar to our last experiment, we also wanted to see if updating the target model every other episode further improves the performance or not. Figure 4.15 and Figure 4.16 shows the results from DDQN models with different number of dense layers for both types of replay buffers. We can observe from the summaries of these models, as seen in Table 4.13 and Table 4.14 that models update every other episode also performs better than models updated every five episodes. It is evident that the target model must be updated more frequently than our default policy.

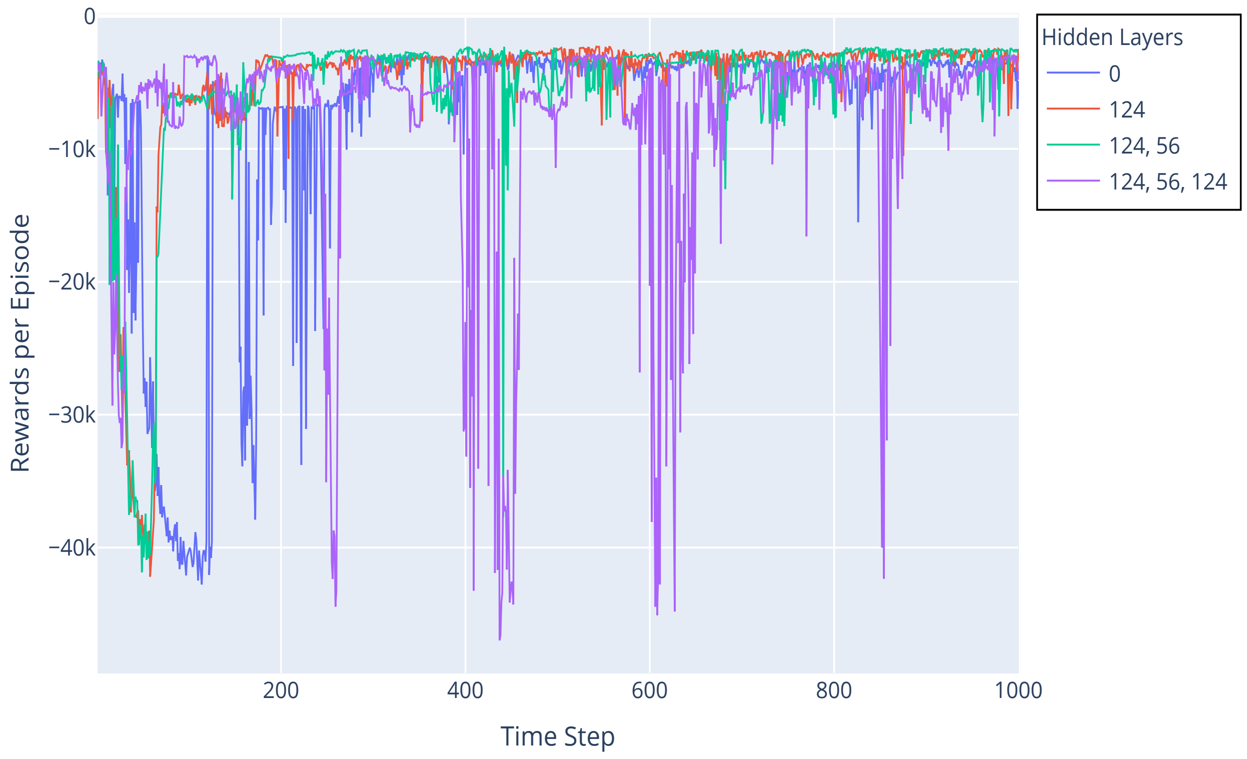

Figure 4.15: Rewards per Episode for DDQN with URB and Target Model updated every alternate episode: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.13: Summary of Sum of All Rewards for DDQN with Target Model updated every alternate episode

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-7727.23	9702.57	-42780.71	-2797.13
100	1	**-5096.49**	6577.57	-42189.23	-2305.73
100	2	-5404.14	6633.26	-41854.39	-2323.57
100	3	-8128.60	8745.39	-46977.44	-2843.71

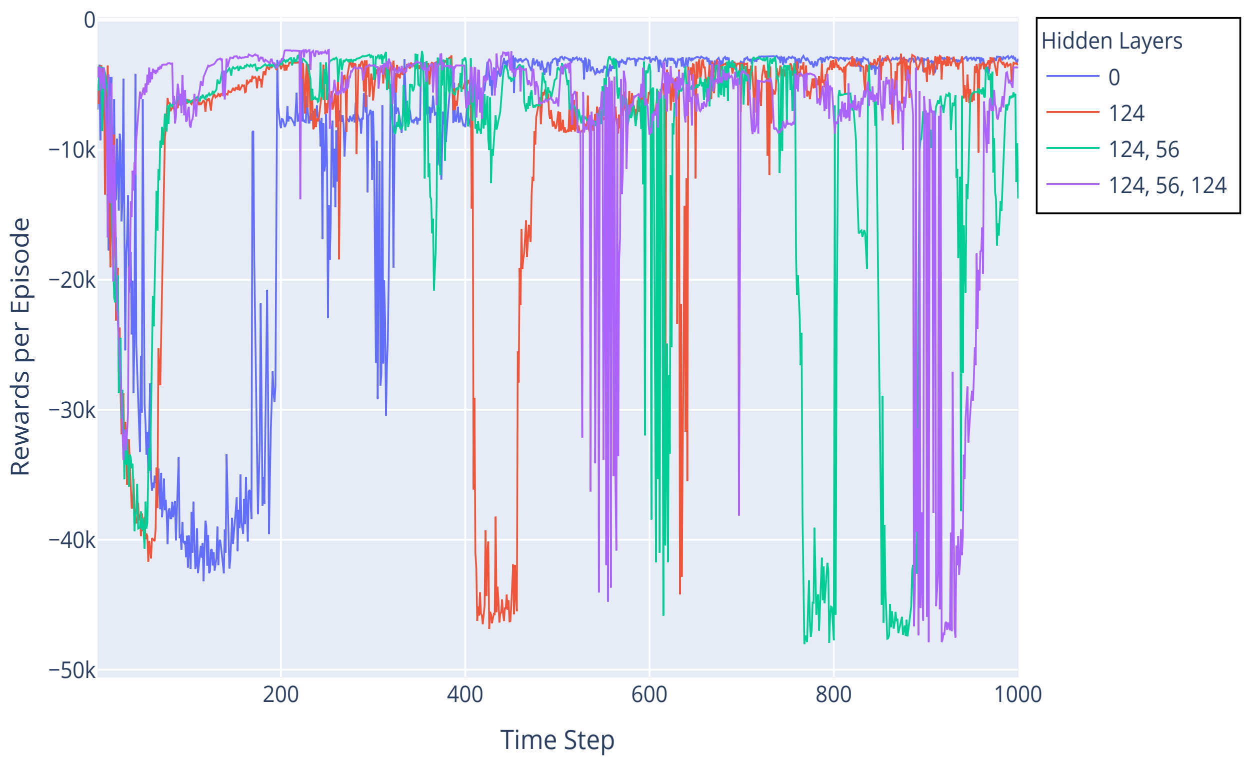

Figure 4.16: Rewards per Episode for DDQN with PRB and Target Model updated every alternate episode: Setpoint P=100, NN Hidden layers=(0, 1=124, 2=56, 3=124)

Table 4.14: Summary of Sum of All Rewards for DDQN with PRB and Target Model updated every alternate episode

Setpoint	Num of Dense Layer	Mean	Std. Dev	Min	Max
100	0	-9831.50	12039.82	-43198.96	-2719.47
100	1	-8919.87	10929.45	-46863.78	-2617.66
100	2	-10791.03	12377.70	-48027.71	-2409.93
100	3	**-7939.46**	9433.53	-47875.71	-2328.19

4.2.8 Experience Replay Buffer

We have discussed experience replay buffers in detail in Section 3.2.1. Here, we will evaluate the effects of replay buffer size on the performance for both uniform and prioritized replay buffers.

Figure 4.17 and Figure 4.18 shows the results for DDQN with uniform and prioritized replay buffers, respectively. The replay buffer size of 100,000 produced the best results in both cases, as seen in summaries of the models described in Table 4.15 and Table 4.16.

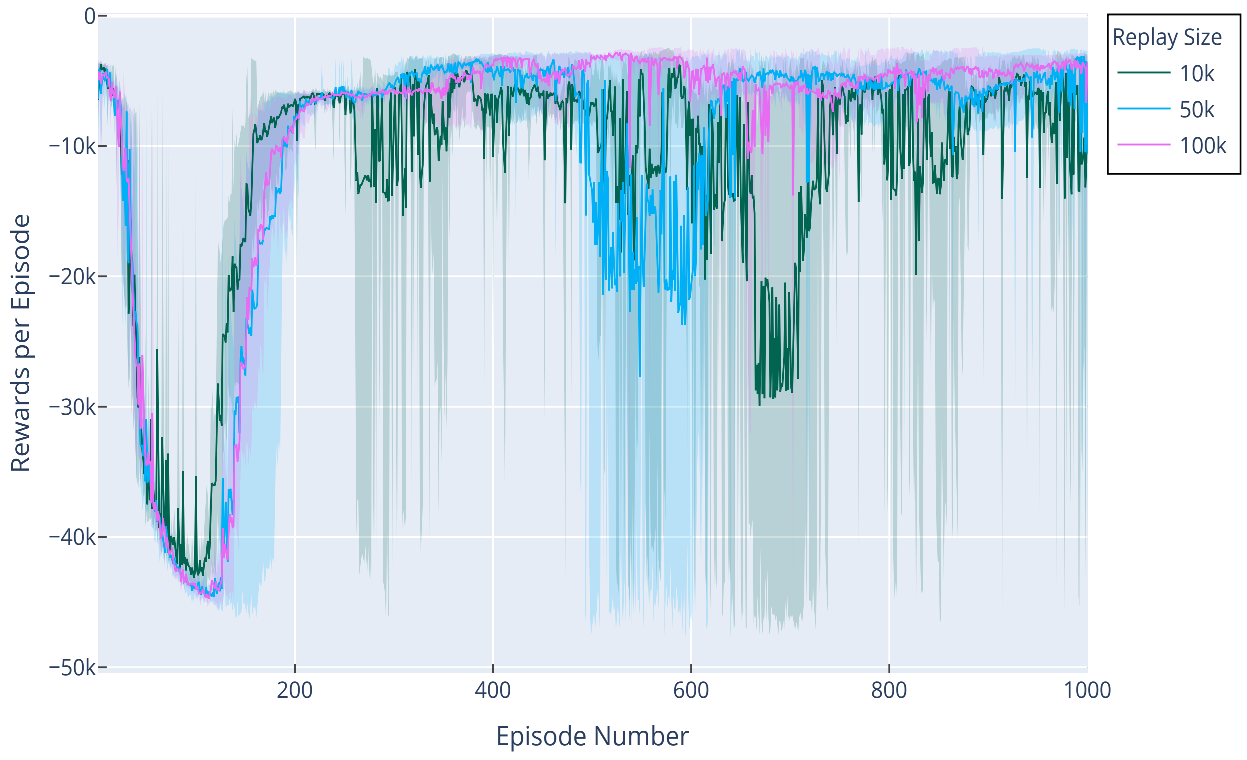

Figure 4.17: Average Rewards per Episode for DDQN with URB Size=(10k, 50k, 100k), Setpoint P=100

Table 4.15: Summary of Sum of All Rewards for DDQN Vs. Size of Uniform Replay Buffer

Setpoint	Replay Buffer Size	Mean	Std. Dev	Min	Max
100	10k	-12129.97	9546.47	-43143.38	-3735.83
100	50k	-11032.31	11156.79	-44600.57	-3074.04
100	100k	**-9236.51**	11094.01	-44740.94	-2824.23

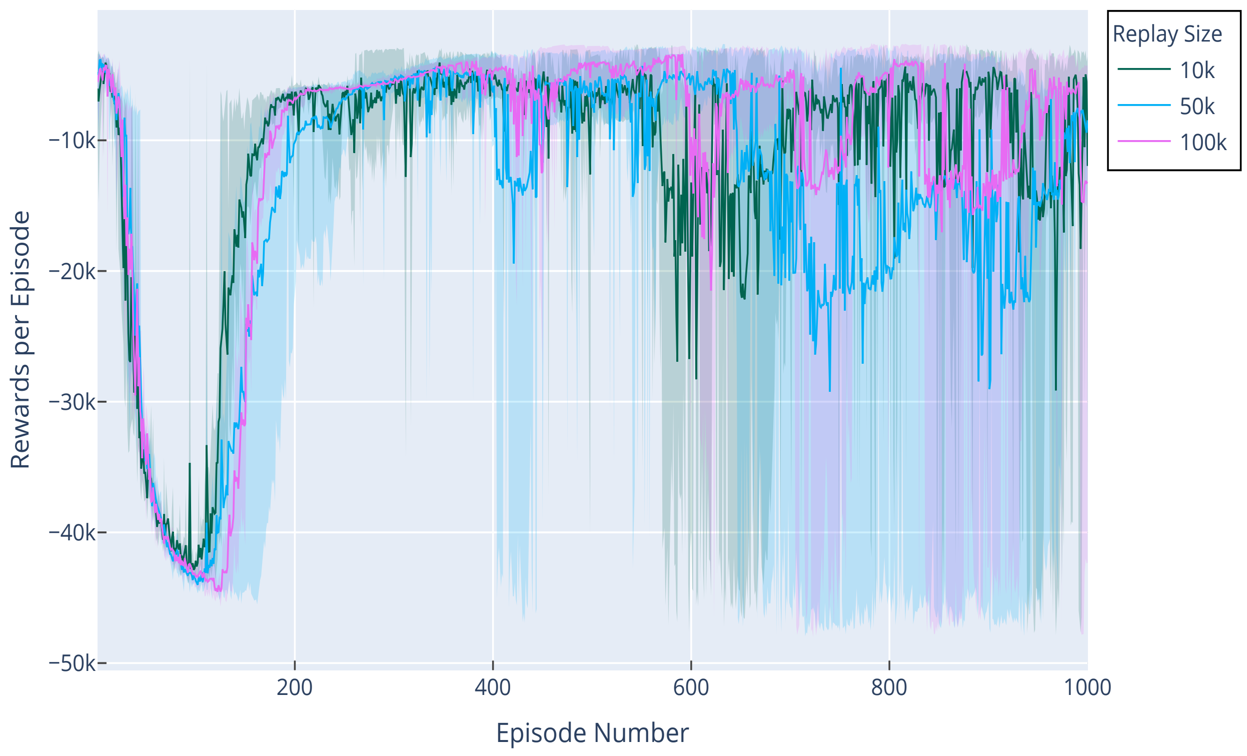

Figure 4.18: Average Rewards per Episode for DDQN with PRB Size=(10k, 50k, 100k), Setpoint P=100

Table 4.16: Summary of Sum of All Rewards for DDQN Vs. Size of Prioritized Replay Buffer

Setpoint	Replay Buffer Size	Mean	Std. Dev	Min	Max
100	10k	-11582.78	9489.59	-42867.10	-4048.51
100	50k	-14214.56	10299.48	-44012.59	-3819.32
100	100k	**-11189.28**	10869.74	-44542.95	-3408.89

4.2.9 Input States

We also experimented with three different formats of input to ANN. Figure 4.19 shows the normal input format, which consists of N random transitions drawn from the experience replay. Figure 4.20 shows the sequential input format with sequential window $k = 3$, where a sequential window is formed three consecutive transitions, drawn randomly from the experience replay. Figure 4.21 shows the third input format, which again consists of N random samples drawn from the experience replay, but each sample is a combination of the current transition combined with the last two transitions forming a serial window of $k = 3$ states.

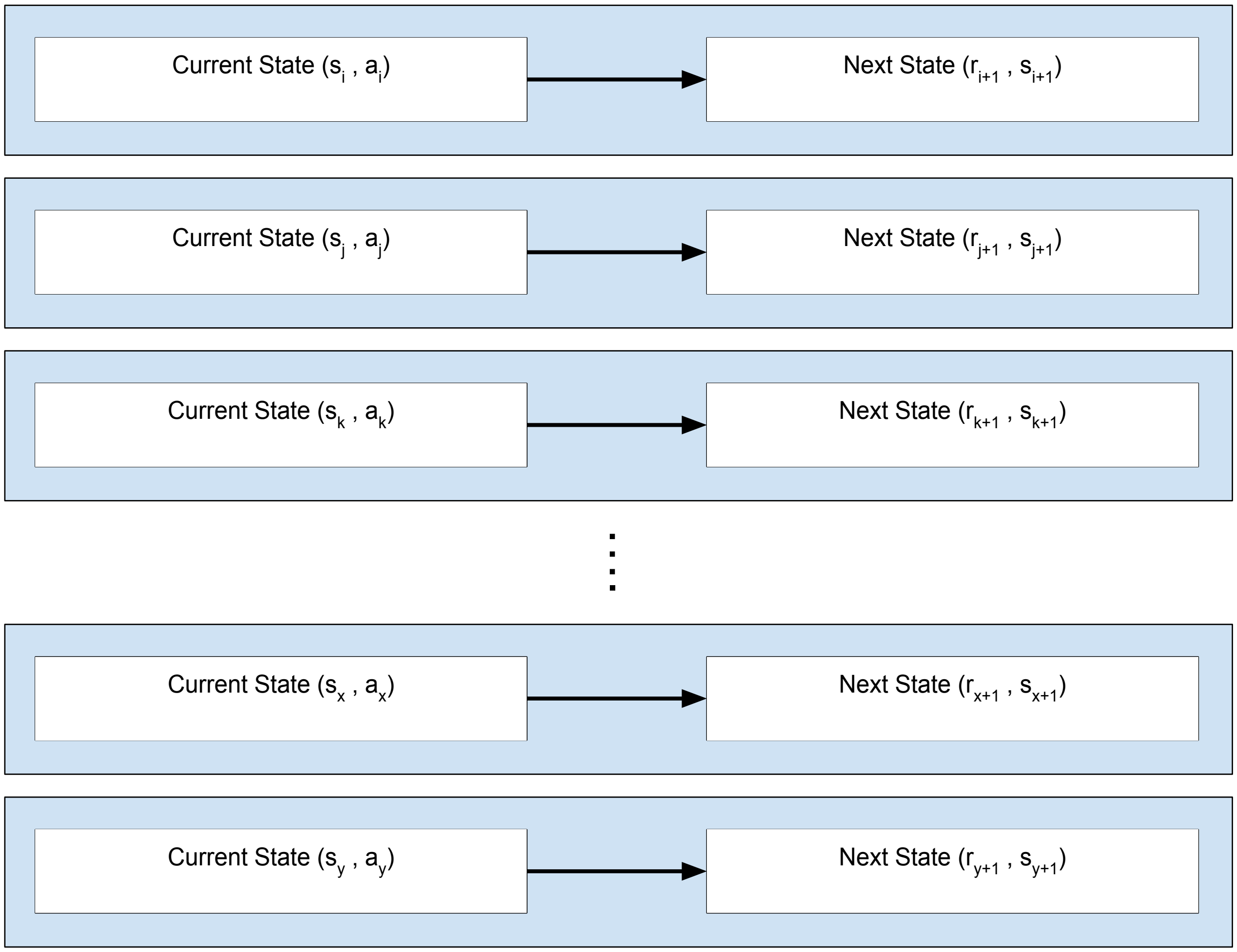
Current State (s_i , a_i)
Next State (r_{i+1} , s_{i+1})
Current State (s_j , a_j)
Next State (r_{j+1} , s_{j+1})
Current State (s_k , a_k)
Next State (r_{k+1} , s_{k+1})
Current State (s_x , a_x)
Next State (r_{x+1} , s_{x+1})
Current State (s_y , a_y)
Next State (r_{y+1} , s_{y+1})

Figure 4.19: Design of Normal Input State

The rationale for testing different input formats is to understand if representing the same data in another format can provide more insight into the problem or not. In case of the sequential state, ANN is trained on mini-batch containing sample of transitions $(s_{i+t-1}, s_{i+t}, s_{i+t+1}) \,\forall\, i \in N$, which introduces little bit of correlation while maintaining generalization. We are considering sequential states because we hypothesize that it will produce more stable policies than normal state.

On the other hand, in long state, three previous states are used to form a single input to the ANN to produce the next state and the reward like $(s_{i+t-2}, s_{i+t-1}, s_{i+t}, a_{i+t}) \longrightarrow (r_{i+t+1}, s_{i+t+1})\ \forall\ i \in N$. The ANN now receives three times the number of inputs compared to normal or sequential state. We hypothesize that the new transformed state as input will provide more information which should further improve the policy.

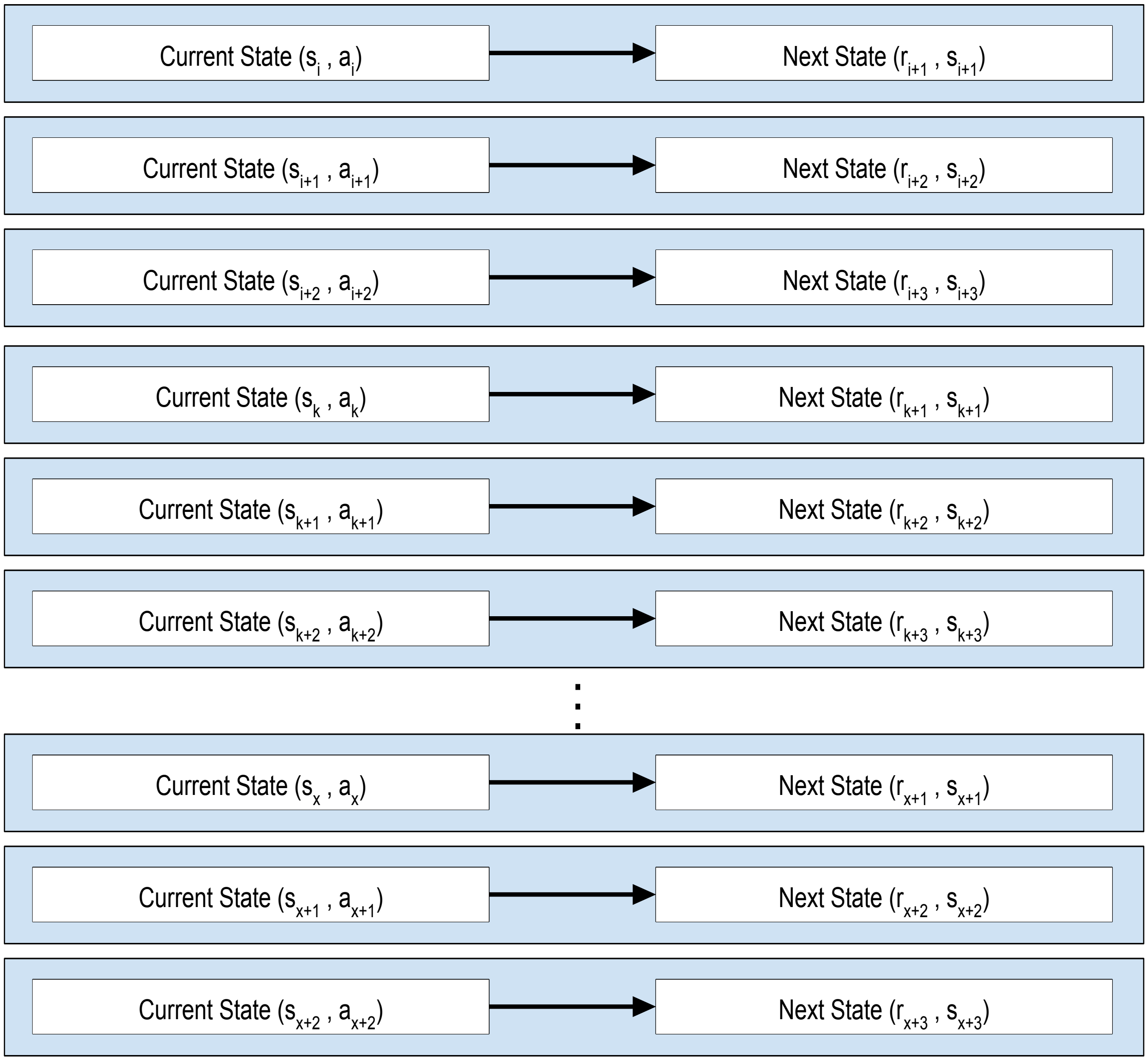
Current State (s_i , a_i)
Next State (r_{i+1} , s_{i+1})
Current State (s_{i+1} , a_{i+1})
Next State (r_{i+2} , s_{i+2})
Current State (s_{i+2} , a_{i+2})
Next State (r_{i+3} , s_{i+3})
Current State (s_k , a_k)
Next State (r_{k+1} , s_{k+1})
Current State (s_{k+1} , a_{k+1})
Next State (r_{k+2} , s_{k+2})
Current State (s_{k+2} , a_{k+2})
Next State (r_{k+3} , s_{k+3})
Current State (s_x , a_x)
Next State (r_{x+1} , s_{x+1})
Current State (s_{x+1} , a_{x+1})
Next State (r_{x+2} , s_{x+2})
Current State (s_{x+2} , a_{x+2})
Next State (r_{x+3} , s_{x+3})

Figure 4.20: Design of Sequential Input State with window k=3

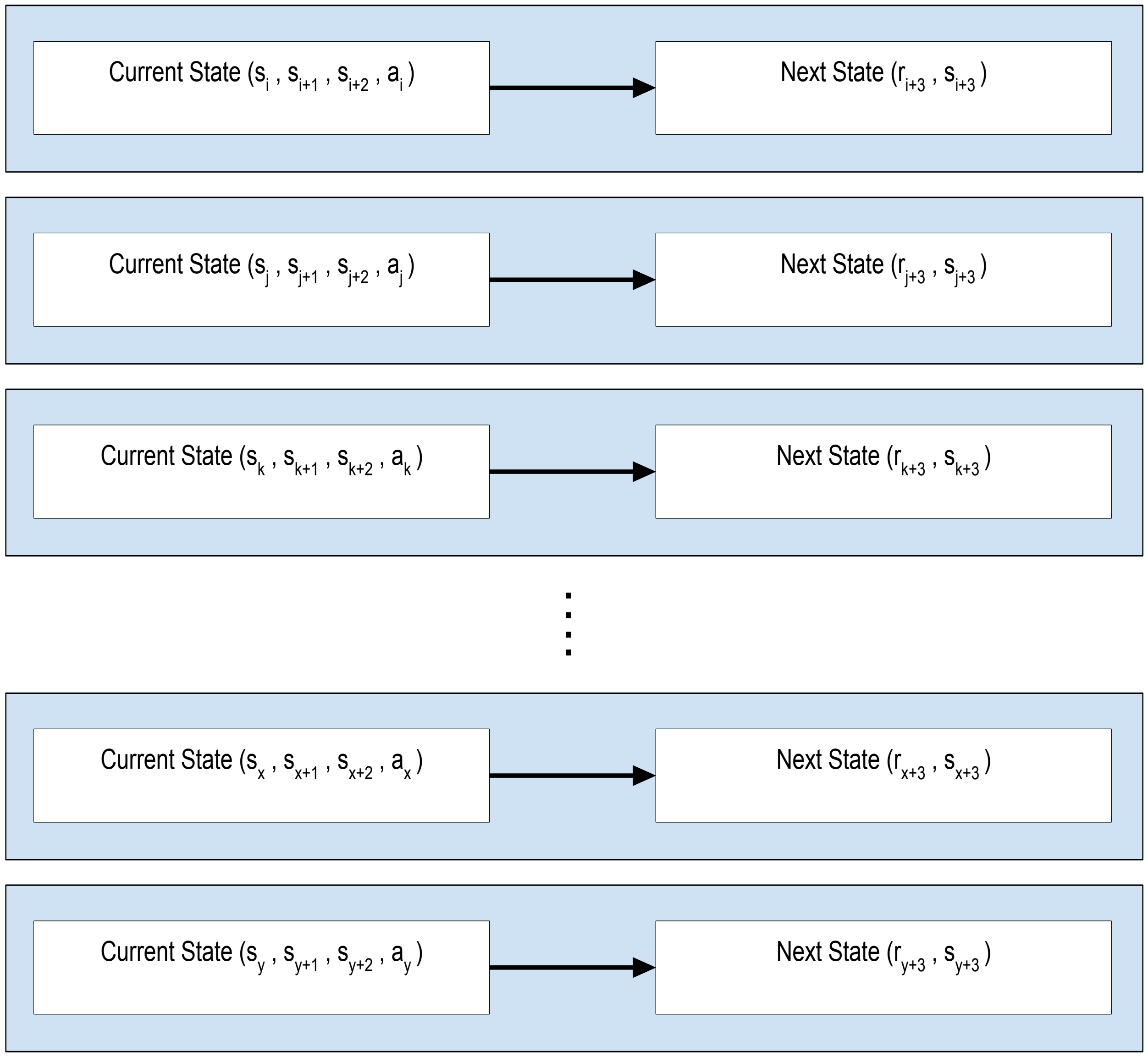
Current State (s_i , s_{i+1} , s_{i+2} , a_i)
Next State (r_{i+3} , s_{i+3})
Current State (s_j , s_{j+1} , s_{j+2} , a_j)
Next State (r_{j+3} , s_{j+3})
Current State (s_k , s_{k+1} , s_{k+2} , a_k)
Next State (r_{k+3} , s_{k+3})
Current State (s_x , s_{x+1} , s_{x+2} , a_x)
Next State (r_{x+3} , s_{x+3})
Current State (s_y , s_{y+1} , s_{y+2} , a_y)
Next State (r_{y+3} , s_{y+3})

Figure 4.21: Design of Long Input State with overlap k=3

Figure 4.22 shows the reward per episode for serial windows of size 1, 3, and 5 for setpoint 10 and 100, and the results are summarized in Table 4.17. We can see from the results that they are significantly better in comparison to the normal state, which is the same as the sequential state with a sequential window, k = 1. The results support our hypothesis that the sequential state produces more stable and better policies. Figure 4.23 shows the rewards per episode for sequential state across different mini-batch sizes and its results are summarized in Table 4.18.

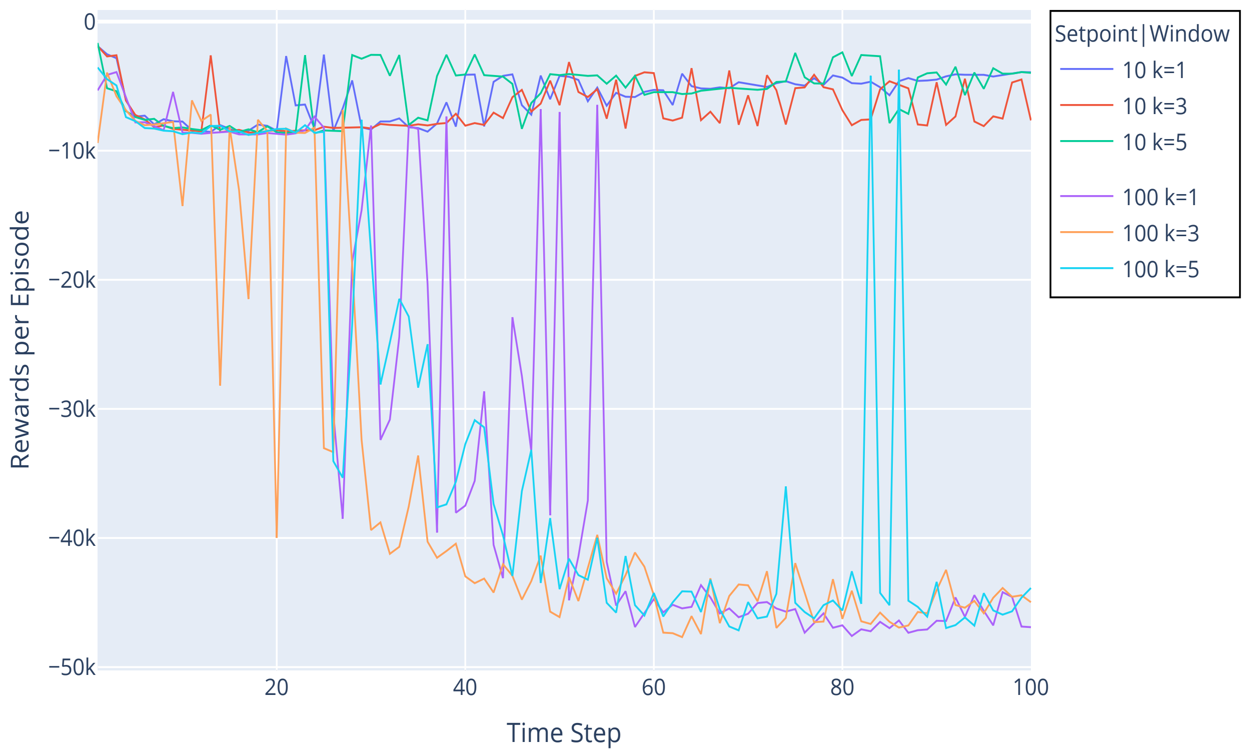

Figure 4.22: Rewards per Episode: Setpoint P=(10 & 100), Sequential States k=(1, 3, 5)

Table 4.17: Summary of Rewards for k Sequential States

Setpoint	Window	Mean	Std. Dev	Min	Max
10	1	-5735.14	1744.13	-8598.07	-1898.13
10	3	-6744.74	1762.94	-8606.56	-1895.30
10	5	**-5371.68**	1999.33	-8656.13	-1655.21
100	1	**-30718.85**	17172.20	-47592.58	-3909.40
100	3	-35004.10	15102.42	-47676.62	-3949.97
100	5	-31425.28	16177.17	-47155.55	-3552.49

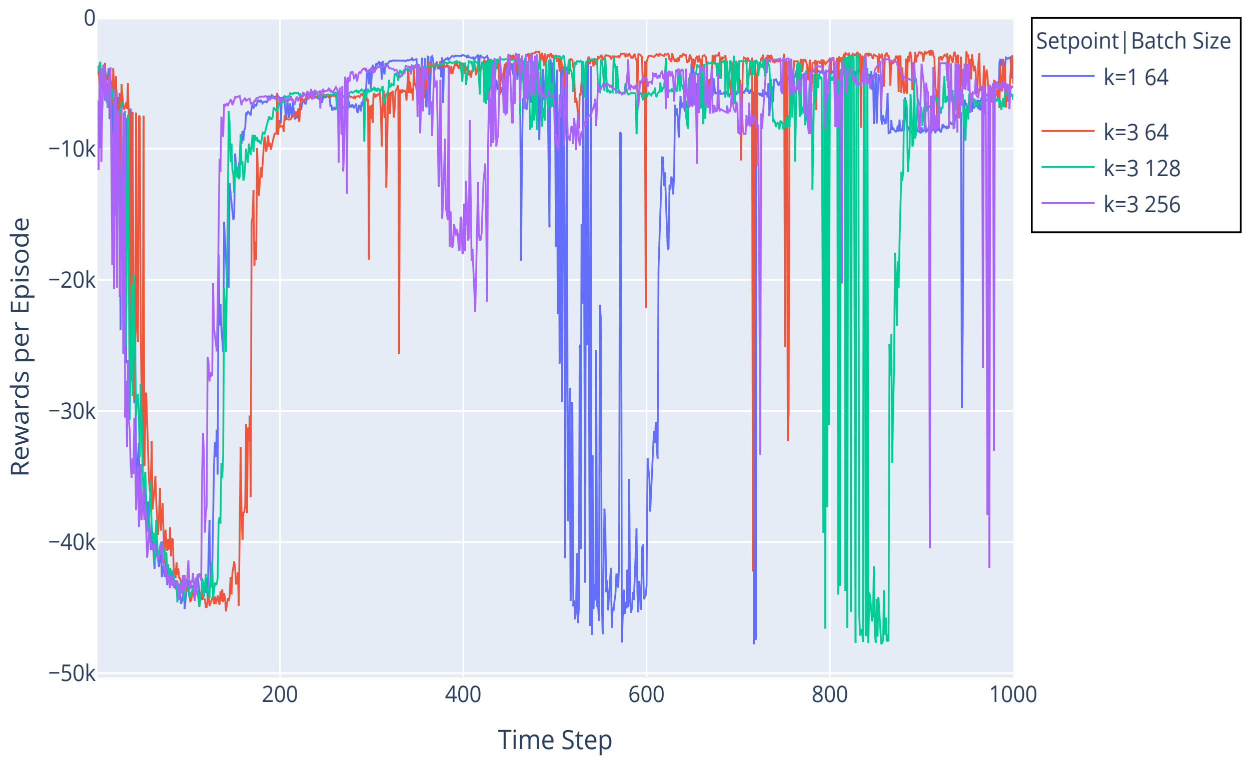

Figure 4.23: Rewards per Episode: Setpoint P=100, Sequential States k=(1 & 3), Mini-batch=(64, 128, 256)

Table 4.18: Summary of Rewards for Sequential States Vs. Mini-batch Size

Setpoint	Window	Batch Size	Mean	Std. Dev	Min	Max
100	1	64	-12909.95	13983.14	-47778.67	-2821.35
100	3	64	**-9128.12**	12219.18	-45280.54	-2467.03
100	3	128	-11021.44	12988.45	-47764.92	-2855.04
100	3	256	-9680.52	10534.58	-44472.53	-2726.99

Figure 4.25 shows the rewards per episode for the long state against the normal state, and Figure 4.25 shows the rewards per episode for a long state with different ANN architecture against the

normal state with the default setting. The results of the experiments are summarized in Table 4.19 and Table 4.20, respectively. We can see that the long state does not improve the fitness value.

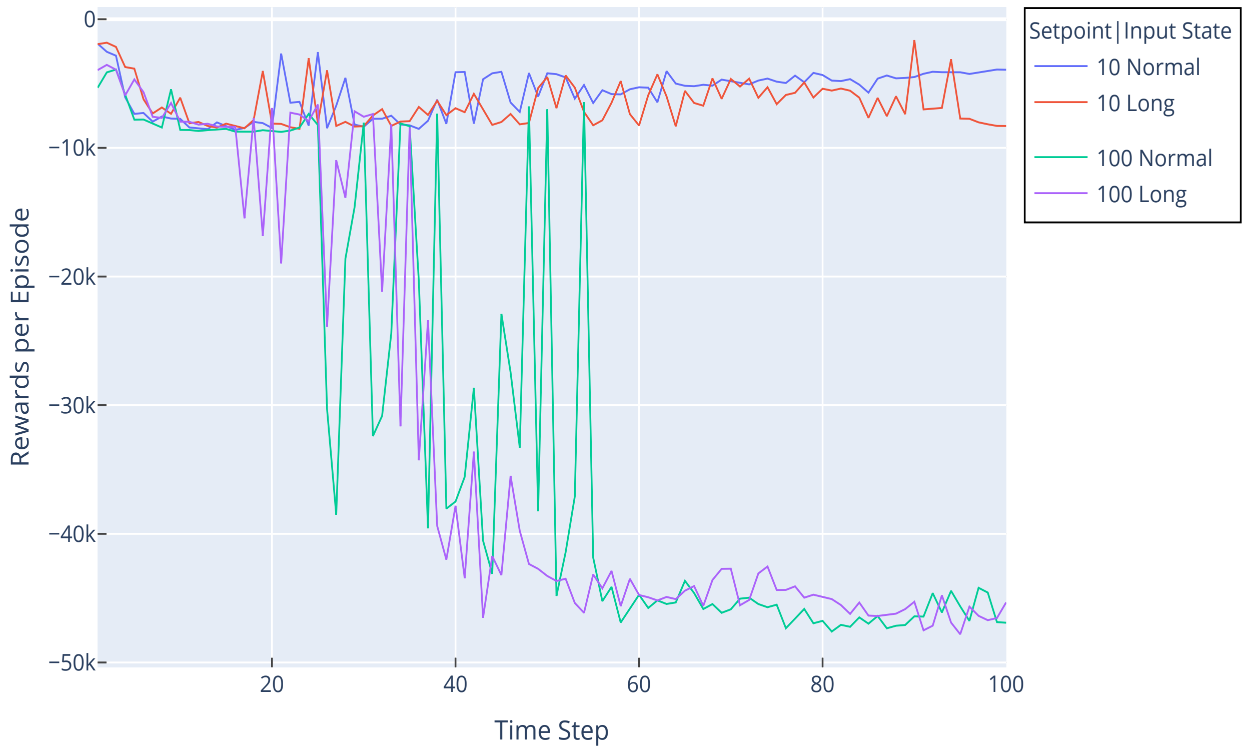

Figure 4.24: Rewards per Episode: Setpoints P=(10 & 100), Long State k=3

Table 4.19: Summary of Long State Vs. Normal State

Setpoint	State Type	Mean	Std. Dev.	Min	Max
10	Normal	**-5735.14**	1744.13	-8598.07	-1898.13
10	Long	-6538.21	1698.62	-8518.45	-1622.97
100	Normal	**-30718.85**	17172.20	-47592.58	-3909.40
100	Long	-31841.07	16938.53	-47803.97	-3548.36

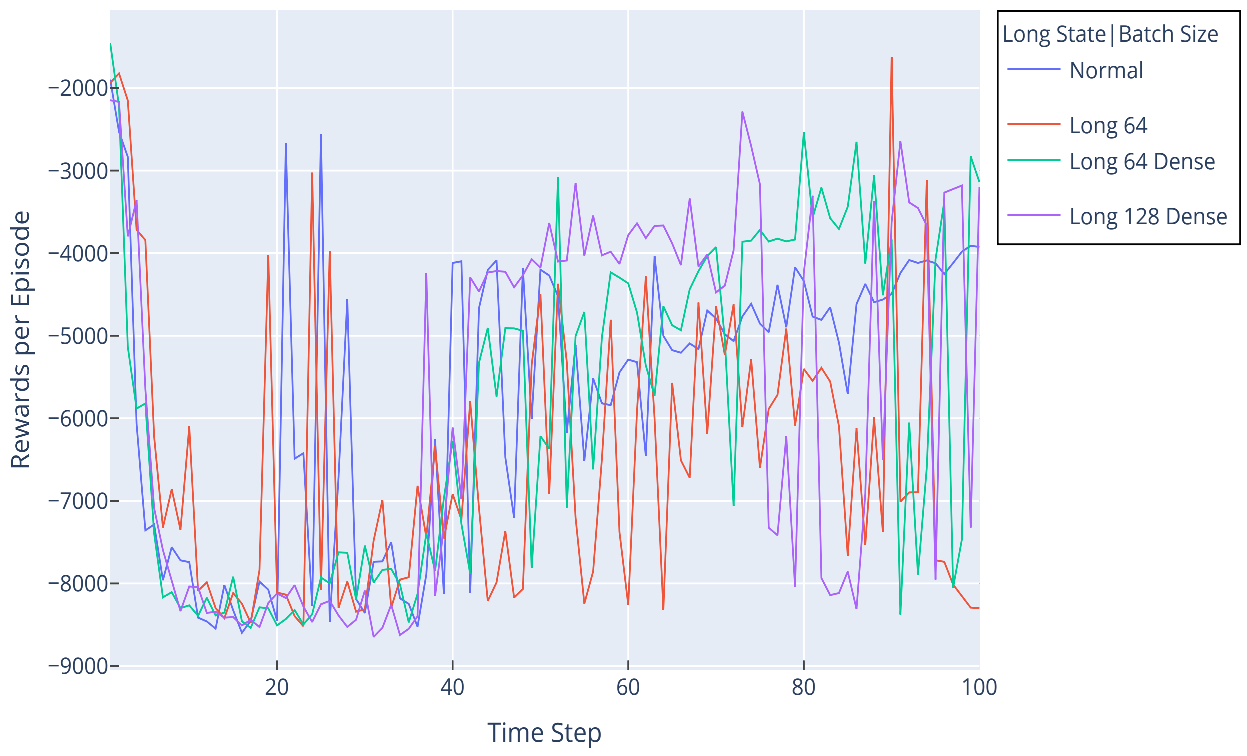

Figure 4.25: Rewards per Episode: Setpoint P=10, Long State k=3, Mini-batch=(64, 128), NN Hidden layers=(1=124,2=56,3=56)

Table 4.20: Summary of Long State Vs. Neural Network Layers Vs. Mini-batch Size

Setpoint	State/ Batch/ Dense_Layer Size	Mean	Std. Dev.	Min	Max
10	Normal	**-5735.14**	1744.13	-8598.07	-1898.13
10	Long 64	-6538.21	1698.62	-8518.45	-1622.97
10	Long 64 Dense	-5939.05	2001.46	-8538.23	-1457.47
10	Long 128 Dense	-5769.73	2222.4	-8647.88	-2148.54

4.2.10 Summary

Finally, Table 4.21 summarizes the best models from all the experiments that produced relevant results along with an Oracle (the known reward function), and a fixed random model with no learning/training. All models that have performed better than the random model without any learning have been highlighted with bold text. Overall, we can say that prioritized replay does not perform better than uniform replay for DDQN with at least one dense layer. The performance of DQN is reasonable but not significant. The most important observation is the target model update frequency. The DDQN model, with the target model, updated every episode outperforms all the other models in similar settings. As mentioned in Section 2.4, the fitness values for the best policy developed using FPSRL and FGPRL [38] are $\tilde{\mathcal{F}} = -5700$ and $\tilde{\mathcal{F}} = -5635$, respectively. Please note that the fitness values of these models are based on the time horizon T of 100, whereas the fitness values of all the policies that we have produced and summarized Table 4.21 are based on the time horizon T of 1000. Even though it is not an apples-to-apples comparison, but our models have outperformed the best of FPSRL and FGPRL [38].

Table 4.21: Summary of all the best models

Model Name	**Setpoint**	**Mean**	**Std. Dev**	**Min**	**Max**
Oracle	100	-4838.15	1835.29	-20093.91	-2856.39
Random with no learning	100	-8760.45	540.68	-8914.76	-3650.82
DQN with 0 dense layer	100	-8979.82	6576.28	-44268.75	-3461.81
DQN with PRB and 1 dense layer	100	**-8657.39**	6710.42	-47483.94	-2669.50
DDQN with batch-size=128	100	-10741.57	10458.93	-44158.97	-3570.59
DDQN with PRB and batch-size=64	100	-14214.56	10299.48	-44012.59	-3819.32
512 mini sdg (100 episodes)	100	**-4395.18**	2095.30	-16378.02	-2744.79
DDQN with 3 dense layers	100	**-6686.92**	7524.28	-41407.52	-2880.93

Table 4.21 continued from previous page

Model Name	**Setpoint**	**Mean**	**Std. Dev**	**Min**	**Max**
DDQN with PRB and 3 dense layers	100	**-7202.13**	7910.00	-42092.31	-2770.99
DDQN with 1 dense layers and target update every episode	100	**-6032.01**	4859.15	-35556.74	-2667.84
DDQN with PRB, 0 dense layer and target update every episode	100	**-5906.40**	7186.25	-37560.60	-2930.66
DDQN with 1 dense layer and target update every other episode	100	**-5096.49**	6577.57	-42189.23	-2305.73
DDQN with PRB, 3 dense layers and target update every other episode	100	**-7939.46**	9433.53	-47875.71	-2328.19
DDQN with URB=100k	100	-9236.51	11094.01	-44740.94	-2824.23
DDQN with PRB=100k	100	-11189.28	10869.74	-44542.95	-3408.89
DDQN with window=3 batch-size=64	100	-9128.12	12219.18	-45280.54	-2467.03
FPSRL	10-100	-5700*	-	-	-
FGPRL	10-100	-5635*	-	-	-

Chapter 5

Conclusions and Future Work

This study evaluates the performance of DRL algorithms DQN and DDQN on the novel Industrial Benchmark problem. We conduct numerous experiments to develop a deeper understanding of the IB problem. We showed using our empirical results that (1) the external load which is controlled by the setpoint is directly related to the complexity of the simulated problem and requires sophisticated models, (2) uniform replay buffer performs better than prioritized replay buffer for non-linear models, (3) both SGD and Adam optimizers can be used to generate good policies with careful tuning of hyper-parameters, (4) different patterns of the observable state as input did not improve the performance in comparison to using the complete observable state as input, (5) the number of dense layers affected both the fitness value and the stability of the model, and (6) the update frequency of the Target model in DDQN had a significant impact on all the models. Our results show that DDQN with the target model updated every episode tends to generate the best policy for the IB problem.

5.1 Future Work

Our experiments and results successfully highlight the significance of IB as a benchmark for RL algorithms. It utilizes the external load and the control variables to simulate a real-world industrial control system and eliminates the problem of a lack of quality industrial data for researchers. Furthermore, IB can simulate the same environment for both discrete and continuous actions enables us to make an apples-to-apples comparison of RL algorithms irrespective of the action type they support.

In this work, we have focused on the discrete action space of IB, and so there is a lot to be explored in the realms of continuous action space. The results from DQN and DDQN may be optimized further by using more complex ANN [56] for functional approximation. Smarter policy optimization techniques like Proximal Policy Optimization (PPO) [45], Trust Region Policy Optimization (TRPO) [44], and Actor-Critic model [26], can be used to generate more efficient, optimal and generalized policies. The continuous control policies, which use Continuous-DQN (CDQN) with Normalized Acceleration Function (NAF) [25] can also incorporate actor-critic models [27], which would improve the performance of our models both in terms of stability and accuracy. This work will encourage other researchers to include the realistic IB in future comparative studies of reinforcement learning algorithms for involving real-world problems.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

[3] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *Computing Research Repository*, abs/1708.04782, 2017.

[4] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[6] J. Zurada. *Introduction to Artificial Neural Systems*. West Publishing Co., USA, 1992.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[9] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[10] Martin Schlang, Björn Feldkeller, Bernhard Lang, Thomas Poppe, and Thomas Runkler. Neural computation in steel industry. In *1999 European Control Conference (ECC)*, pages 2922–2927. IEEE, 1999.

[11] TA Runkler, E Gerstorfer, M Schlang, E Jünnemann, and J Hollatz. Modelling and optimisation of a refining process for fibre board production. *Control engineering practice*, 11(11):1229–1241, 2003.

[12] Stephan A Hartmann and Thomas A Runkler. Online optimization of a color sorting assembly buffer using ant colony optimization. In *Operations Research Proceedings 2007*, pages 415–420. Springer, 2008.

[13] Anton Maximilian Schaefer, Daniel Schneegass, Volkmar Sterzing, and Steffen Udluft. A neural reinforcement learning approach to gas turbine control. In *2007 International Joint Conference on Neural Networks*, pages 1691–1696. IEEE, 2007.

[14] D. Hein, S. Depeweg, M. Tokic, S. Udluft, A. Hentschel, T. A. Runkler, and V. Sterzing. A benchmark environment motivated by industrial control problems. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017.

[15] Arne Traue, Gerrit Book, Wilhelm Kirchgässner, and Oliver Wallscheid. Towards a reinforcement learning environment toolbox for intelligent electric motor control. *arXiv preprint arXiv:1910.09434*, 2019.

[16] Henrik Bode, Stefan Heid, Daniel Weber, Eyke Hüllermeier, and Oliver Wallscheid. Towards a scalable and flexible simulation and testing environment toolbox for intelligent microgrid control. *arXiv preprint arXiv:2005.04869*, 2020.

[17] Edouard Leurent. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

[18] J. Chen, B. Yuan, and M. Tomizuka. Model-free deep reinforcement learning for urban autonomous driving. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2765–2771, 2019.

[19] Jianyu Chen, Shengbo Eben Li, and Masayoshi Tomizuka. Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning. *arXiv preprint arXiv:2001.08726*, 2020.

[20] S. Sun, Z. Cao, H. Zhu, and J. Zhao. A survey of optimization methods from a machine learning perspective. *IEEE Transactions on Cybernetics*, pages 1–14, 2019.

[21] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *Deep Reinforcement Learning Symposium, NIPS*, 2017.

[22] Martín Abadi, Michael Isard, and Derek G. Murray. A computational model for tensorflow: An introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, page 1–7, New York, NY, USA, 2017. Association for Computing Machinery.

[23] François Chollet et al. Keras. https://keras.io, 2015.

[24] Matthias Plappert. keras-rl. https://github.com/keras-rl/keras-rl, 2016.

[25] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *Proceedings of the 33rd International Conference*

on International Conference on Machine Learning - Volume 48, ICML'16, page 2829–2838. JMLR.org, 2016.

[26] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[27] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[28] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[29] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Syst. Man Cybern.*, 13(5):834–846, 1983.

[30] Andrew William Moore. Efficient memory-based learning for robot control. Technical Report UCAM-CL-TR-209, University of Cambridge, Computer Laboratory, November 1990.

[31] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[32] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization

challenge in vision, control, and planning. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 2684–2691. ijcai.org, 2019.

[33] Łukasz Kidziński, Sharada P. Mohanty, Carmichael F. Ong, Jennifer L. Hicks, Sean F. Carroll, Sergey Levine, Marcel Salathé, and Scott L. Delp. Learning to run challenge: Synthesizing physiologically accurate motion using deep reinforcement learning. In Sergio Escalera and Markus Weimer, editors, *The NIPS '17 Competition: Building Intelligent Systems*, pages 101–120, Cham, 2018. Springer International Publishing.

[34] Daniel Hein, Steffen Udluft, Michel Tokic, Alexander Hentschel, Thomas A Runkler, and Volkmar Sterzing. Batch reinforcement learning on the industrial benchmark: First experiences. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4214–4221. IEEE, 2017.

[35] Daniel Hein, Alexander Hentschel, Thomas A Runkler, and Steffen Udluft. Reinforcement learning with particle swarm optimization policy (pso-p) in continuous state and action spaces. *International Journal of Swarm Intelligence Research (IJSIR)*, 7(3):23–42, 2016.

[36] A. M. Schaefer, S. Udluft, and H. Zimmermann. A recurrent control neural network for data efficient reinforcement learning. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 151–157, 2007.

[37] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[38] Daniel Hein, Steffen Udluft, and Thomas A Runkler. Generating interpretable fuzzy controllers using particle swarm optimization and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1268–1275, 2018.

[39] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[40] Sascha Lange, Thomas Gabel, and Martin Riedmiller. *Batch Reinforcement Learning*, pages 45–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[41] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[42] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1995–2003. JMLR.org, 2016.

[43] Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[44] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1889–1897. JMLR.org, 2015.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

[47] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[48] A. Rupam Mahmood, Hado P van Hasselt, and Richard S Sutton. Weighted importance sampling for off-policy learning with linear function approximation. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3014–3022. Curran Associates, Inc., 2014.

[49] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[50] Y. NESTEROV. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady AN USSR*, 269:543–547, 1983.

[51] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[52] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[53] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[54] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[55] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[56] Alexander Hans and Steffen Udluft. Efficient uncertainty propagation for reinforcement learning with limited data. In *International Conference on Artificial Neural Networks*, pages 70–79. Springer, 2009.

Appendix A

State Description

The IB state space is only partially observable. The observation vector which consists of observable variables does not fulfill the Markov property. The observation vector o_t at time t comprises current values of velocity v_t, gain g_t, shift h_t, setpoint p_t, consumption c_t, and fatigue f_t.

A state space that consists of 20 values fulfills the Markov property with the minimum number of variables. The complete Markov state comprises of the observation vector (v_t, g_t, h_t, p_t, c_t, and f_t), and some latent variables of the sub-dynamics. Please refer to the original paper [14] for more details about the sub-dynamics of the latent variables.

Table A.1: IB Markovian State [14].

		text name or description	symbol
Markovian state	Observables	setpoint	p_t
		velocity	v_t
		gain	g_t
		shift	h_t
		consumption	c_t
		fatigue	f_t
		operational cost at $t-1$	θ_{t-1}
		operational cost at $t-2$	θ_{t-2}
		operational cost at $t-3$	θ_{t-3}
		operational cost at $t-4$	θ_{t-4}
		operational cost at $t-5$	θ_{t-5}
		operational cost at $t-6$	θ_{t-6}
		operational cost at $t-7$	θ_{t-7}
		operational cost at $t-8$	θ_{t-8}
		operational cost at $t-9$	θ_{t-9}
		1st latent variable of mis-calibration	δ
		2nd latent variable of mis-calibration	ψ
		3rd latent variable of mis-calibration	ϕ
		1st latent variable fatigue	μ^{v}
		2nd latent variable fatigue	μ^{g}