# Chasing Hallucinated Value: A Pitfall of Dyna Style Algorithms with Imperfect Environment Models

by

Taher Jafferjee

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Statistical Machine Learning

Department of Computing Science

University of Alberta

# Abstract

In Dyna style algorithms, reinforcement learning (RL) agents use a model of the environment to generate *simulated* experience. By updating on this simulated experience, Dyna style algorithms allow agents to potentially learn control policies in fewer environment interactions than agents that use model-free RL algorithms. Dyna, therefore, is an attractive approach to developing sample efficient RL agents. In many RL problems, however, it is seldom possible to learn a perfectly accurate model of environment dynamics. This thesis explores what happens when Dyna is coupled with an imperfect environment model.

We present the *Hallucinated Value Hypothesis*. We hypothesise that Dyna style algorithms coupled with imperfect environment models may fail to learn control policies if they update $Q$-values of observed states towards values of simulated states. We argue this occurs because the imperfect model may erroneously generate fictitious states that do not correspond to real, reachable states of the environment. These fictitious states may have arbitrary $Q$-values, and temporal difference updates toward them may lead to the propagation of this misleading values through the value function. Consequently, agents may end up incorrectly *chasing hallucinated value*.

We present three Dyna style algorithms that may update real state values toward simulated state values and one which is designed not to. We evaluate these algorithms on *Bordered Gridworld* — a simple setting designed to carefully test the hypothesis. Furthermore, we study whether the hypothesis holds in a

range of standard RL benchmarks: *Cartpole, Catcher*, and *Puddleworld*.

Experimental evidence supports the *Hallucinated Value Hypothesis*. The algorithms which update real state values toward simulated state values struggle to improve their control performance. On the other hand, $n$-step predecessor Dyna, our algorithm which does not perform such updates, seems to be robust to model error on the tested domains. Furthermore, it enjoys speed-ups in learning over its competitors.

# Preface

The work presented in this thesis will be submitted as a paper entitled *Chasing Hallucinated Value: A Pitfall of Dyna style Algorithms With Imperfect Environment Models* to the 34$^{\text{th}}$ AAAI Conference on Artificial Intelligence to be held in New York, New York, USA in February 2020.

*This, too, shall pass.*

– Attar of Nishapur (Allegedly)

# Acknowledgements

Words cannot express my gratitude to my supervisors Professors Michael Bowling and Martha White for putting up with me through the development of this thesis. But, as words are all we have, thank you so much for all your guidance and support Mike and Martha! You're both AWESOME!

I would be remiss if I did not mention Jesse Farebrother who provided me with starter code for DQN. You are a rockstar!

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Legg and Hutter [14] define intelligence as,

> *...[measure of] an agent's ability to achieve goals in a wide range of environments.*

One may claim that achieving a goal in an environment amounts to making decisions which increase the likelihood of that goal materialising. Moreover, reaching goals often requires a sequence of decisions. For instance, for the task of driving home from work, a driver must decide at each choice point on the road about which path to take. On an even finer scale, about how much to depress the accelerator and for how long, how much to change the steering angle, when to brake, and so on.

Reinforcement learning (RL) is a framework for defining and computationally solving sequential decision-making problems. In many RL algorithms, agents solve such problems by estimating the long-term utility of particular actions and defining a *policy* over these estimates. The policy (or *action*, in the jargon of the field) specifies the best action to take at each choice point. One may claim that as RL agents are able to solve sequential decision making problems they are also, therefore, able to attain 'goals in a wide range of environments'. Hence, RL may be a powerful approach through which one may develop intelligent agents. One family of algorithms under the RL umbrella is Dyna style algorithms. Dyna style algorithms are concerned with *explicitly* using a model of the environment in order to expedite solving RL problems.

## 1.1 A Problem of Dyna style Algorithms

In this thesis, we investigate Dyna [23]. In Dyna, an agent uses a model of its environment to generate *simulated* experience. This experience is treated identically to real experience gained through interaction with the environment and is used to improve the agent's estimates of the long-term utility of particular choices, and, therefore, the agent's control policy. In many domains of interest it is ofttimes very difficult to learn a perfectly accurate model of the environment. For instance, it is difficult to learn a perfect model of the dynamics of even Cartpole — a relatively simple RL problem. Any predictions of *simulated* transition tuples *(state, action, reward, next state, termination)* generated by the model will likely contain some error; the prediction for *next state* given the *state* and *action* may be slightly wrong or even, in extreme cases, be completely fictitious (i.e., a state that does not occur in the environment). Thus, in this thesis, we explore the question,

*What are some consequences of using an imperfect model in Dyna?*

## 1.2 Why This Problem?

A central component of our (human) intelligence is the ability to make use of a model of the world to plan ahead for unrealised scenarios to aid in decision making. The ability to perform 'model-based planning' endows us with the ability to make, loosely speaking, good decisions in situations that we have never been in but foresee happening. Dyna style algorithms mimic this capacity. However, in many RL problems it is difficult to learn a perfectly accurate model of environment dynamics, and so we need algorithms that are robust to model error. This would enable Dyna algorithms to take advantage of approximate models thereby reducing the burden of trying to accurately model environment dynamic. Moreover, in some sense this would make Dyna algorithms more like our intelligence.

Apart from the philosophical argument of developing more human-intelligence-like algorithms, developing Dyna algorithms robust to model imperfections is

one way to mitigate the sometimes large sample requirements of modern RL algorithms that use artificial neural networks (ANNs) as function approximators. The seminal DQN [16] algorithm requires $200,000,000$ interactions with games in the Arcade Learning Environment (ALE) [1] to learn good control policies. In that vein, OpenAI's *Hand* [19] demands an equivalent of 100 years worth of training data to learn human-like hand dexterity, while *Alphastar* [31] used 200 years of Starcraft II experience to reach professional level play. Suffice it to say, RL algorithms combined with ANNs have significant data requirements. When there is a cost to interacting with environments, such large sample requirements may make these algorithms cost prohibitive to be a practical option.

Indeed, large sample requirements even plague classical tabular RL algorithms. It was to ameliorate the data hunger of tabular RL algorithms that Dyna was developed. In many cases, Dyna algorithms are empirically shown to be able to learn improved behaviour within fewer interactions with the environment than their model-free (i.e., algorithms that do not explicitly use models to simulate the environment) counterparts. This work is simply one step toward further extending Dyna to the modern setting: as it is very difficult to develop perfect environment models for the problems solved by modern RL algorithms, understanding pitfalls of Dyna planning with imperfect models is a useful step towards building more sample efficient algorithms.

## 1.3 Outline of this Thesis

We set out to 1) understand the dynamics of Dyna with imperfect models, and 2) find ways around one identified problem. To this end, we develop the *Hallucinated Value Hypothesis* about Dyna with imperfect models: updating real state values to simulated state values leads to problems. This thesis is organised around investigating this hypothesis, and finding alternative algorithms that avoid such updates.

**Chapter 2 - Background** summarises relevant background material. We briefly survey the RL setting and temporal-difference (TD) learning

methods before reviewing Dyna style algorithms. We also overview Deep Q-Networks (DQN), a modern instantiation of $Q$-learning combining RL and deep learning.

**Chapter 3 - The Hallucinated Value Hypothesis** presents the hypothesis around which this thesis is built. We postulate that Dyna style algorithms which update real state values to simulated state values are brittle in the face of model error and that this can catastrophically prohibit learning of a good control policy. Furthermore, we develop intuition about why particular updates cause failure and propose an algorithm that is robust to erroneous simulated experience.

**Chapter 4 - The *Hallucinated Value Hypothesis* in *Bordered Grid-world*** shows results of experiments which test our hypothesis on a simple tabular domain.

**Chapter 5 - ...and in *The Wild*** presents results of running our Dyna algorithms on more complex domains not amenable to a tabular representation of the value function. We eliminate contrivances of our experiments from Chapter 4 and show evidence that our hypothesis still holds.

**Chapter 6 - Conclusions & Future Work** recapitulates the central hypothesis of this thesis. Moreover, we offer suggestions about potentially fruitful future research directions in further developing Dyna with imperfect models.

## 1.4 Contributions

The focus of this thesis is Dyna coupled with an imperfect environment model. To this field of study, we contribute the following:

We present the ***Hallucinated Value Hypothesis*** (Chapter 3). We claim that Dyna planning updates which result in the values of *real* states being updated towards *simulated* states may detrimentally impact learning.

We experimentally explore this hypothesis on a carefully constructed problem (Chapter 4), and on benchmark RL problems (Chapter 5).

Building upon the prior contribution, we introduce ***n-step predecessor Dyna*** (Chapter 3), and algorithm that is designed not to update real state values to simulated state values. We show that this algorithm seems to be more robust to model error than algorithms which update real state values to simulated state values (Chapters 4 and 5).

# Chapter 2

# Background

We summarise background material relevant to the contributions in this thesis. We begin by introducing the reinforcement learning (RL) setting. Then, we describe temporal difference (TD) learning algorithms (including a discussion of the DQN algorithm). Following this, we overview model-based RL, and in particular Dyna style algorithms. Finally, we briefly review a method to model environment transition dynamics. Readers familiar with RL may want to jump to Section 2.5 where model learning is discussed or skip this chapter entirely.

## 2.1 The Reinforcement Learning Setting

The RL setting involves a decision-making agent interacting with an environment. This setting is often expressed by Markov Decision Processes (MDPs). MDPs consist of a set of states of the environment, $\mathcal{S}$, a set of permissible actions, $\mathcal{A}$, and a reward function $r$ such that $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$; the reward function indicates the 'pleasure' or 'pain' of taking particular actions in particular states by emitting positive or negative reward. MDPs allow flexible choices for $\mathcal{S}, \mathcal{A}, r$: they can be discrete or continuous. Moreover, the environment's transition dynamics are modelled by a transition function $p$. This function defines the probability of transitioning from state $s$ to $s'$ given an action $a$ executed by the agent in $s$, that is, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$). In addition, a 'discount factor' scalar $\gamma \in [0, 1]$ is fixed as well. The discount factor indicates to what extent rewards received sooner are valued over rewards received later. Thus, the RL setting is defined by the tuple $(\mathcal{S}, \mathcal{A}, r, p, \gamma)$.

A notion of time is manifest in the agent's interaction with the environment. As shown in Figure 2.1, at each time step $t$, the agent is situated in some state $s_t \in \mathcal{S}$ (the$s_t$ indicates the agent is in state $s$ at time $t$). The agent possess a policy $\pi : \mathcal{S} \mapsto \mathcal{A}$ defining how to act in each state. The agent selects an action $a_t \in \mathcal{A}$ according to $\pi$ and executes it in the environment resulting in the agent transitioning to a new state $s_{t+1} \in \mathcal{S}$ (according to the distribution given by the function $p(s_t, a_t, s_{t+1})$). Furthermore, the agent receives two feedback signals: a scalar reward $r_t \in R$ indicating the utility of taking $a_t$ in $s_t$, and a Boolean termination signal $\tau_{t+1}$. In the continuing setting (RL has two settings: episodic and continuing) $\tau_t$ is always `False` (i.e., an episode never ends).

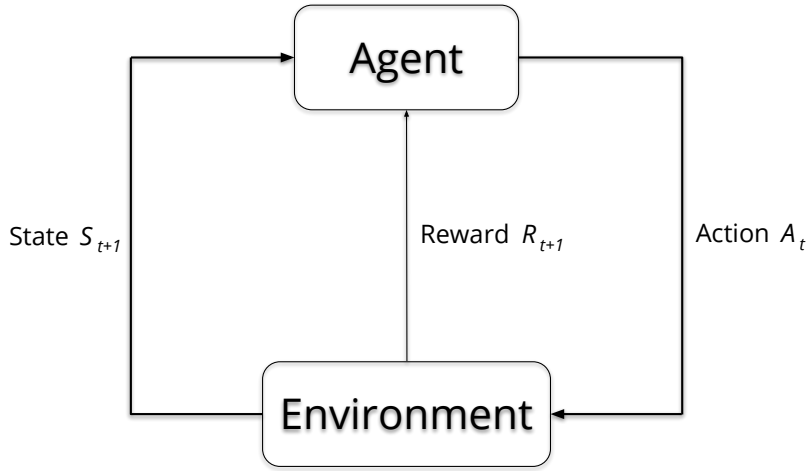



Figure 2.1: **The Reinforcement Learning Setting**. An agent starts off situated in some state $S_0 = s_0$ (note the capitalised $S_0$ indicates the agent's start state is some random variable). Applying some policy $\pi$ to choose actions, the agent executes an action $A_0 = a_0$ (again, capitalisation of $A_0$ indicates the agent's action choices are probabilistic). Consequently, the agent is now in state $S_1 = s_1$ and the environment also emits a probabilistically distributed reward signal $R_1 = r_1$ designating the utility of taking action $a_0$ in state $s_0$. This cycle repeats until the termination signal $\tau_t$ (not shown) is `True`. Once the episode has ended (in the episodic setting), the agent restarts in some start state $S_0 = s_0^{'}$.

In this setting, the agent's objective is to maximise the cumulative reward collected. This is formalised by the notion of *return*. Return (of a reward sequence $R_0, R_1, R_2, R_3, ...$) is defined as:

$$G_t \doteq \mathbb{E}\Big[\sum_{0}^{\infty} \gamma^i R_{t+i}\Big] \tag{2.1}$$

Intuitively, this formula characterises the expected discounted long term accumulation of reward. The $\gamma$ term in the return ensures that this sum is bounded (as $r$ is a bounded function). Given that we have a definition of return – something that we would like to optimise – how can we use it to aid learning of control policies (i.e., policies which tell agents how to act in a given environment) in autonomous agents? One answer is provided by *value functions*.

Suppose an agent is situated in state $s$. From state $s$, the agent's interaction with the environment is the following trajectory $s_0 = s, a_0, r_0, s_1, a_1, r_1, \ldots$. The $a_i$ are selected according to policy $\pi$, and the return is defined by the $r_i$ of this trajectory. Let the value function $V_\pi(s)$ be a function from states, $s \in \mathcal{S}$, to $\mathbb{R}$, i.e., $V_\pi : \mathcal{S} \mapsto \mathbb{R}$. This function defines the return that may be accumulated from $s$ if an agent follows a policy $\pi$ to select actions in $s$ and all subsequent states.

The value function, thus, defines the 'goodness' of following a particular policy $\pi$ from $s$. Some policies may results in higher $V(s)$, and others in equal or lower $V(s)$. And, by extension, in some settings (tabular and linear), some policies may result in higher values of $V(s)$ for all $s \in \mathcal{S}$ and some policies in lower values of $V(s)$ for all $s \in \mathcal{S}$. To maximise the return, we want policies which give us the highest values of $V(s)$ for the most states $s \in S$.

An extension of the value function to incorporate particular actions $a$ selected in $s$ can be made. This function, called the *state-action* or $Q$-value function, $Q_\pi(s, a)$, defines the return from $s$ if action $a$ is taken in $s$ and policy $\pi$ is followed to select actions thereafter. Indeed, with an accurate state-action value function, it becomes trivial for an agent to choose actions to maximise the return (as per the agent's current estimates) from any state: simply pick $\max_a Q_\pi(s, a)$. The $Q$-value function, thus, allows agents to learn control policies $\pi$.

Having defined the RL setting and value functions, we now define the RL problem:

"The agent's sole objective is to maximise the total reward it receives

> over the long run... The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future."
>
> Sutton and Barto, 2018 [25]

Succinctly, the problem we seek to solve is how best to endow agents with the ability to alter their behaviour such that they maximise the return they receive from the distribution of start states of the MDP. In the next section, we introduce temporal difference methods, one solution to the reinforcement learning problem.

## 2.2 Temporal Difference Learning

We illustrated that a notion of time is manifest in the RL setting. Temporal difference (TD) [22] algorithms exploit information contained in consecutive time-steps to learn the value function for $\pi$. Consider an experience trajectory $s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_n, a_n, r_n$. How can an agent learn the value of state $s_0$?

If we look at this trajectory in reverse, from $s_n$ to $s_0$, one way to learn state values becomes apparent. We defined value as the return from a given state. Since $s_n$ is a final state before termination, the return is simply equal to $r_n$, thus we can update the value (in the value function) of $s_n$ to reflect that a sample (as $r$ is a probabilistic function and therefore $r_n$ is one sample of $R_n$) of the long term return thereon is $r_n$. Moving on, the return from $s_{n-1}$ consists of two terms: $r_{n-1}$ and all future reward (appropriately discounted). Notice, we have already encapsulated 'all future reward' into the value of $s_n$. Thus, we can replace 'all future reward' with the value of $s_n$. The value of $s_{n-1}$ can therefore be updated towards $r_{n-1}$ and the value of what is called the *bootstrap* state $V(s_n)$, i.e., towards $r_{n-1} + \gamma V(s_n)$. Continuing on in this manner, we obtain values for each state. Indeed, this is the central idea of temporal difference learning, and the value function update for each state becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.2}$$

for some learning rate $\alpha$. It is important to note that the value function ought to reflect the long-term *expected* return from any state. Thus, any single trajectory of experience is unlikely to be a sufficient sample from which an agent may learn an accurate value function. In practice, a set of diverse trajectories through the environment are required to learn the true expected long-term return from any state.

Temporal difference learning is not constrained to learn state values by updating over a single time-step. $n$-step TD updates update the value of a state to the discounted sum of rewards $n-1$ steps into the future along with the *bootstrap* value of the state $n$ steps into the future. The $n$-step TD update is:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big(\sum_{i=0}^{n-1} \gamma^i R_{t+i} + \gamma^n V(S_{t+n}) - V(S_t)\Big) \tag{2.3}$$

In fact, algorithms using $n$-step TD updates sometimes converge faster than algorithms using 1-step TD updates. This important fact speaks to the desirability of our algorithm, $n$-step predecessor Dyna.

Given that we have a method to learn a value function, how can an agent learn a control policy? In the following subsection we describe $Q$-learning, an algorithm to learn optimal control policies under certain settings.

### 2.2.1 $Q$-learning

In the previous section, we described how TD learning can facilitate the learning of a value function. Importantly, we defined it in terms of learning values of some policy $\pi$. In many RL problems, we seek to *learn* a policy telling the agent how to act such that it maximises the return. This is called learning a *control* policy. One algorithm which enables us to learn such a policy is $Q$-learning. $Q$-learning is a *fundamental* RL algorithm that we use throughout this thesis. $Q$-learning [32] is an off-policy TD *control* algorithm guaranteed to learn the state-action value function of the optimal policy (in the tabular and linear settings) on data generated under any behaviour policy (hence the term *off-policy*). That is, the converged action-value function $Q_\pi$ learned using $Q$-learning is equivalent to $Q_*$ (the value function of the optimal

policy), independent of the policy followed to select actions (with the caveat that convergence requires all state-action pairs to be selected infinitely often). Ipso facto, $Q$-learning is also able to learn an optimal *control* policy, $\pi_*$, as this policy can be inferred from the values of $Q_*$ by simply picking $\max_a Q(s,a)$ for all states $s$ . The TD update of 1-step tabular $Q$-learning is of the form:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{2.4}$$

while the TD update of $n$-step tabular $Q$-learning is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\rho_{t:n}\Big[\sum_{i=0}^{n-1} \gamma^i R_{t+i} + \gamma^n \max_a Q(S_{t+n}, a) - Q(S_t, A_t)\Big] \tag{2.5}$$

Importantly, in multi-step $Q$-learning we must 'weight' the update target $\sum_{t=0}^{n-1} R_{t+i} + \gamma^n \max_a Q(S_{t+n}, a) - Q(S_t, A_t)$ by importance sampling ratio $\rho_{t:n} = \prod_{i=0}^{n-1} \frac{\pi(A_{t+i}|S_{t+i})}{b(A_{t+i}|S_{t+i})}$ where $\pi(A_{t+i}|\cdot)$ represents the probability of taking actions under the optimal policy while $b(A_{t+i}|\cdot)$ represents the probability of taking actions under the behaviour policy. This is needed to account for the fact that actions are selected according to the distribution defined by $b$, while the expected values of a different policy $\pi_*$ are learned.

In the case of a linear function approximator, rather than directly updating values in a state-action value table, the $Q$-learning algorithm learns the values of weights $\boldsymbol{\theta}$. Given a feature generator $\boldsymbol{\phi}$, the 1-step TD update for $Q$-learning with a linear function approximator is:

$$\boldsymbol{\theta^{i+1}} \leftarrow \boldsymbol{\theta^i} + \alpha\Big[R_{t+1} + \gamma \max_a(\boldsymbol{\phi}(S_{t+1}, a)^T \boldsymbol{\theta^i}) - \boldsymbol{\phi}(S_t, A_t)^T \boldsymbol{\theta^i}\Big]\boldsymbol{\phi}(S_t, A_t) \tag{2.6}$$

while the $n$-step update is:

$$\boldsymbol{\theta^{i+1}} \leftarrow \boldsymbol{\theta^i} + \alpha\rho_{t:n}\Big[\sum_{i=0}^{n-1} R_{t+i} + \gamma^n \max_a(\boldsymbol{\phi}(S_{t+n}, a)^T \boldsymbol{\theta^i}) - \boldsymbol{\phi}(S_t, A_t)^T \boldsymbol{\theta^i}\Big]\boldsymbol{\phi}(S_t, A_t) \tag{2.7}$$

### 2.2.2 Deep $Q$-Networks (DQN)

Introduced by Mnih *et al.* [16], Deep Q-Network (DQN) demonstrated "...human-level control through deep reinforcement learning" on the Arcade

Learning Environment (ALE) [1]. DQN is a model-free reinforcement learning algorithm that utilises an artificial neural network to approximate the $Q$-value function. The state of the environment, $s$, is fed as input to the network. Following this, the network outputs the state-action values for all actions in that state: $f(s) = Q(s, a)\ \forall a$. In order to marry neural networks with reinforcement learning algorithms *experience replay buffers* and *target networks* are required. We elaborate on these algorithmic innovations as well as the loss function in the following.

**Network Architecture**

The original DQN network architecture was designed to process images from the ALE. The experiments in this thesis only use environments where states are represented by feature vectors, thus we adapt the nework architecture for our purposes. As depicted in Figure 2.2, in this thesis we use a network architecture consisting of two hidden layers each of 256 hidden units with the first hidden layer possessing `ReLU` activation. The output layer is composed of estimations of $Q(s, a)\ \forall a \in \mathcal{A}$. The units are initialised with *Glorot* initialisation[4]; the network parameters $\boldsymbol{\theta}$ are optimised with the Adam optimiser [13].

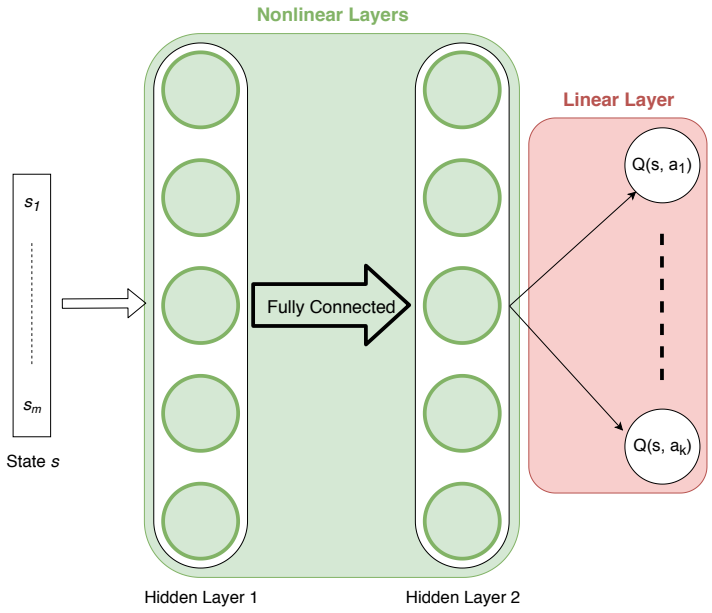



Figure 2.2: **DQN Architecture.**

**Loss Function & Update Rule**

The network weights, $\boldsymbol{\theta}$, are updated by the semi-gradient $Q$-learning update. That is, the loss function minimised is:

$$L_i(\boldsymbol{\theta_i}) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta_i^-}) - Q(s, a; \boldsymbol{\theta_i}\big)^2\Big] \tag{2.8}$$

where mini-batch training samples are drawn uniformly from the pool stored in the experience replay buffer as indicated by $(s, a, r, s') \sim U(D)$. Furthermore, $\boldsymbol{\theta_i^-}$ indicates a network with an asynchronous copy of the agent network weights. Important details (e.g., modifying this loss with the *Huber* loss) about the optimisation procedure not mentioned here may be found in the work of Mnih *et al.* [16]. Notice, with each update to the network parameters, the term $\max_{a'} Q(s', a'; \boldsymbol{\theta_i^-})$ changes. That is, the neural network is required to approximate a function with a *moving* target. As this is particularly challenging for neural networks, *experience replay buffers* and *target networks* are required to stabilise training in DQN.

**Experience Replay Buffer**

DQN utilises an experience replay buffer [15] to store the $N$ most recent environment transitions $(s_t, a_t, r_t, s_{t+1}, \tau_{t+1})$ experienced by the agent. This buffer is required to be large enough to store multiple episodes of transitions as the neural network expects to train on independent and identically distributed (IID) data samples. Multiple episodes, each of which is run by an agent with a different policy (due to updates to the value function), allow for the data to be roughly IID.

**Target Networks**

The target network is an asynchronous copy of the agent network — the weights of the agent network are copied to the target network at a predetermined frequency. As seen in 2.8 the target network calculates the Q-values, $Q(s', a'; \boldsymbol{\theta_i^-})$, of the next-state of a given transition. Target networks are required so that value estimates remain fixed for a certain period of time thereby giving the network a fixed function to approximate.

This thesis is on model-*based* RL, however, so far, we have only elaborated

on background material related to model-*free* RL. In the next section, we describe the basic ideas and algorithms of model-based RL that we expand upon in this thesis.

## 2.3 Model-based Reinforcement Learning

Model-based reinforcement learning refers to a family of RL algorithms which *explicitly* utilise transition dynamics *models* of the environment. The agents may utilise the models to perform model-based updates to the value function (as in Dynamic Programming), perform decision-time planning, or to generate simulated experience and improve value function estimates as described in the Dyna framework (Section 2.4).

As the environment which an agent operates in can either be deterministic or stochastic, the environment models themselves come in a variety of forms. *Expectation* models (which we use), given an input state $s$ and an action $a$, output a single state $s'$ – the *expected* next state – and a single estimation of the reward $r$, the *expected* reward for the transition from $s$ to $s'$ given $a$. On the other hand, *distribution* or *sample* models produce, for $s, a$, all possible $s'$ and $r$ or a single $s'$ and $r$, respectively. Indeed, as one may immediately notice, in expectation models, the expected state may not correspond to any actually realisable state. However, despite this shortcoming, they are much easier to learn than other variants of environment models, and, so, enjoy widespread use.

The focus of this thesis is on Dyna planning algorithms which improve the value function using expectation models.

## 2.4 Dyna

Dyna is a framework for 'Integrated Planning, Acting, and Learning' [24]. Figure 2.3 summarises the way an agent utilises its real experience in the environment to simultaneously learn a model of the environment, improve its value function estimates, and, *importantly*, uses the model to generate simulated experience to further update the value function. A crucial premise

of the Dyna framework is that real experience and simulated experience are treated identically. That is, simulated experience is treated as though it actually happened, and value function updates using simulated experience are performed in the same manner as value function updates using real experience. Indeed, it is this fact that warrants this thesis examining the impact of imperfect models coupled with Dyna algorithms.

A fundamental distinction between the Dyna framework presented here and the algorithms we investigate is that we do not seek to learn the environment models online. Rather, we presume that we have been given imperfect (learned or otherwise) environment models prior and develop ways to use these model effectively. Our motivation for this choice is that we seek to understand the effects of planning in Dyna style algorithms when the model is incorrect. Learning a model online (i.e., as the agent explores the world) introduces another level of complexity which may make it difficult to isolate the precise effects of the inaccurate model. Of course, one may argue that learning the model online may, in the long run, largely reduce model error. While this may be true for parts of the state space regularly visited by the agent, it may cause the model to become more inaccurate in other parts of the state space thereby introducing error.

As seen in Figure 2.3, Dyna is a flexible framework that permits a variety of specific implementations. For instance, as previously mentioned, one choice is the type of environment model to use. In this thesis, we focus on the following two factors in the Dyna design space:

1. The direction of environment dynamics simulation by the model — forwards or backwards in time.

2. One-step or multi-step TD updates using the model's simulated transitions.

In Chapter 3, we expand on this design space and briefly summarise prior work in the 'quadrants' delineated by this design space. Finally, in the last section of background material that follows, we describe one way to learn environment models.

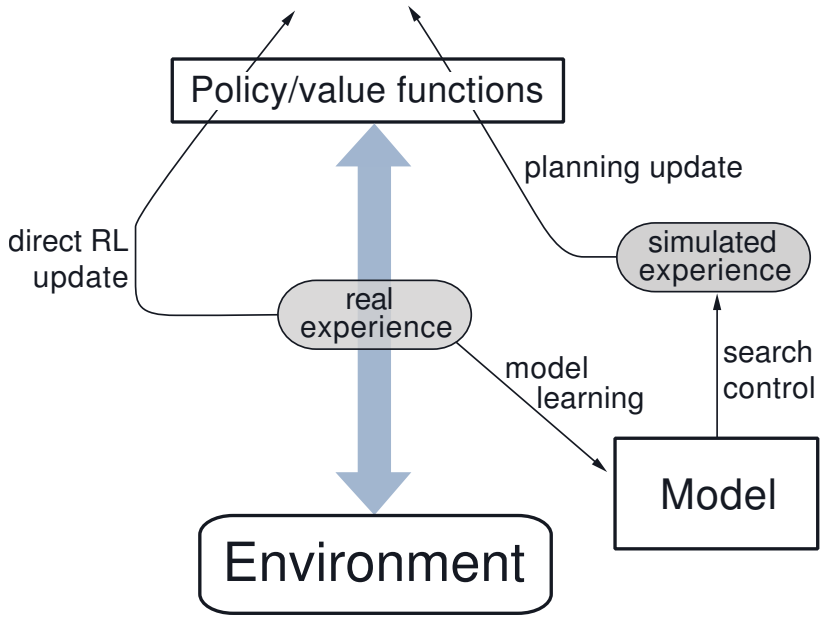



Figure 2.3: **The Dyna Framework.** Reprinted from Sutton & Barto, 2018.

## 2.5 Learning Environment Transition Dynamics Models

A fundamental component of any model-based RL algorithm is the environment transition dynamics model. How can we learn such a model for domains which are high-dimensional and exhibit complex dynamics? Inspired by Oh *et al*. [18], here we describe a simple neural network model capable of modelling to reasonable accuracy the dynamics of the domains we test on.

Our environment model is trained to predict as output either the next state $s_{t+1}$ or the previous state $s_{t-1}$ along with associated reward $r_{t+1}$ or $r_{t-1}$ (depending on if we are learning environment dynamics forward or backward in time) for inputs $s_t$ and $a$, the current state and a given action, respectively. The network architecture is as shown in 2.4. A 1-hot encoding of the action $a$ is concatenated to current state vector $s_t$. The output is a vector of the state in the next/previous time-step, $s_{t\pm1}$ as well as the reward $r_{t\pm1}$. The network consists of two fully connected hidden layers, each with 512 units. The units are initialised with Glorot [4] initialisation, and the network parameters $\boldsymbol{\theta}$ are optimised using the Adam optimiser [13].

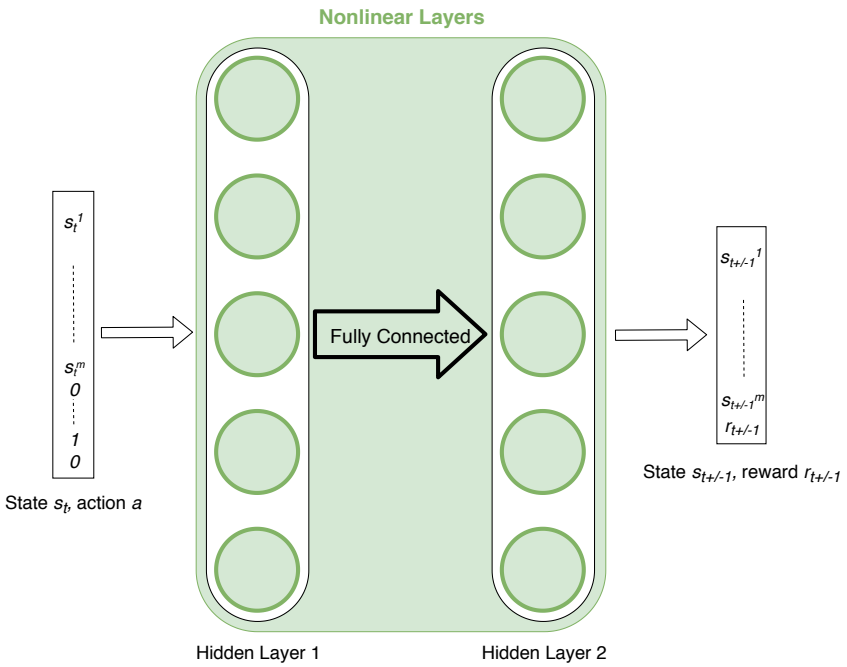



Figure 2.4: **Network Architecture to Learn Environment Models**. A state at a given time-step $t$, $s_t$, is input to the model along with a 1-hot encoding of action $a$. The model outputs state $s_{t+1}$ and reward $r_{t+1}$ if forward dynamics are being learned, or $s_{t-1}$ and reward $r_{t-1}$ if backwards dynamics are being learned.

In order to gather training data for the models, a *pre-trained* agent is run on the environment in question with an $\epsilon$-greedy policy ($\epsilon = 0.5$). Tuples consisting of the current state $s_t$, the action taken $a_t$, reward $r_{t+1}$ and next state $s_{t+1}$ are stored. A relatively high value of $\epsilon$ is used in order to gather samples of data from diverse parts of the environment as opposed to samples only at states visited by an optimal (or close to optimal) trained agent's policy. The training dataset is $100,000$ samples.

The loss function we optimise is the observation-wise (i.e., each element of the state vector) mean squared error (MSE) of the predicted output vector, $\hat{y}_{t\pm1}$, and ground-truth output vector, $y_{t\pm1}$ (note, the output vector consists of the state and reward as shown in Figure 2.4):

$$\mathcal{L}_K(\theta) = \frac{1}{2K} \sum_i ||\hat{y}_{t\pm1}^{(i)} - y_{t\pm1}||_2^2 \tag{2.9}$$

where $K$ is the mini-batch size.

The MSE loss function means this model is an *expectation* model: the output is a single, *expected* previous/next state and reward as opposed to a distribution over possible previous/next states. While this design is expedient for learning, it may cause the model to generate predictions which do not correspond to any *real* state or reward. Any value function updates using these fantasised predictions could potentially have adverse ramifications with respect to learning good policies. Indeed, we explore this phenomenon more thoroughly in the next chapter and claim that model error can have severely detrimental effects on the agent's learning.

# Chapter 3

# The *Hallucinated Value Hypothesis*

In this chapter, we present the main hypothesis of this thesis. We claim that Dyna algorithms with *imperfect* environment models may find learning difficult if planning temporal difference (TD) updates move values of real states towards values of simulated states. Before presenting the technical statement of our hypothesis, in the following section, we first develop intuition about why it is plausible. We then elaborate on the Dyna design space delineated in Chapter 2 and develop conjectures about algorithms which ought to learn successfully and which may struggle.

## 3.1 How Dyna Can Go Wrong

To develop intuition for how planning with imperfect models can cause failure, we introduce an illustrative new environment to test reinforcement learning (RL) algorithms: *Bordered Gridworld*.

### 3.1.1 *Bordered Gridworld*

*Bordered Gridworld* is an extension to the classical tabular RL Gridworld. It is designed to test the hypotheses we present in the remainder of this chapter. Figure 3.1 shows a graphical representation of *Bordered Gridworld*.

As in typical RL Gridworlds, the agent's task is to reach the goal state in as few steps as possible. To do this, the agent may move in the relative directions

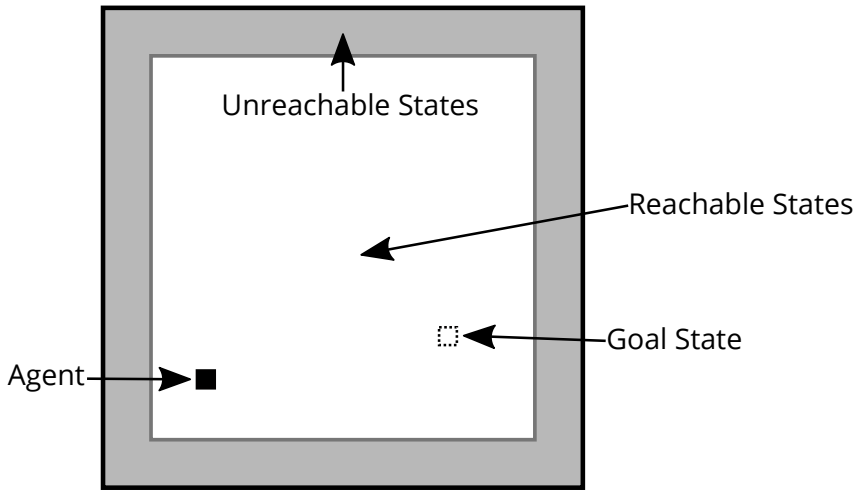
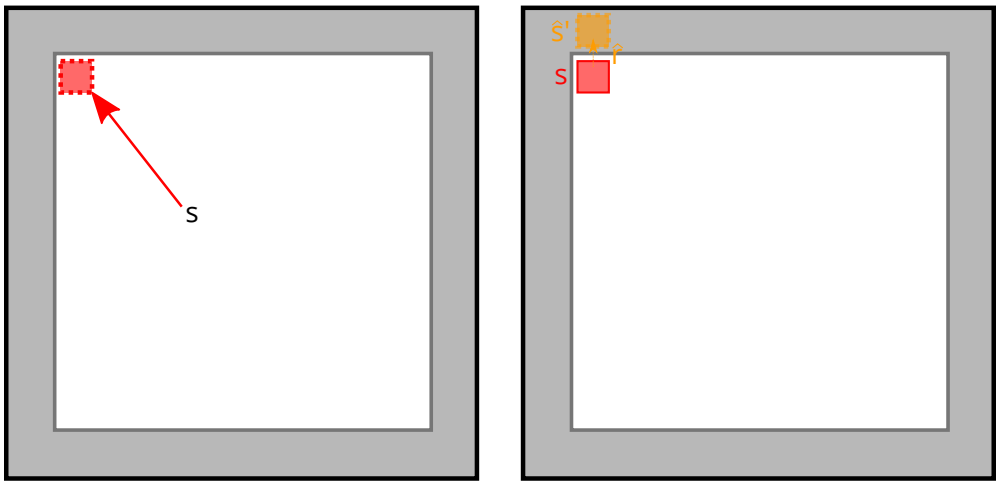



Figure 3.1: ***Bordered Gridworld***. The black square is the agent's position. The agent cannot transition into any states in the grey border. However, an imperfect environment dynamics models may produce state predictions to or within this border.

(`up, down, left,` or `right`) in the region marked 'Reachable States' in Figure 3.1. The feature distinguishing *Bordered Gridworld* from typical Gridworlds is a region of 'Unreachable States', defined by the set $\mathcal{S}_{r^c}$, surrounding the 'Reachable States', i.e., the set $\mathcal{S}_r$ (note these are disjoint). The agent cannot transition to these 'border' states and is never initialised to start in them either. For this following section consider, however, an imperfect environment model which produces simulated transitions from reachable to unreachable states and within unreachable states.

### 3.1.2 Dyna Planning on *Bordered Gridworld*

Consider the planning TD update (on a tabular value function) as shown in Figure 3.2. An agent performs a planning update with an *imperfect* model (which generates simulated transitions from reachable to border states) from real state $s$ to simulated next state $s'$ for action `up`. The state $s'$ is in the unreachable border due to imperfect predictions by the model. If the value of $Q(s, \texttt{up})$ is updated to the value $s'$ inside the border, the state-action value will be moved toward the essentially arbitrary value of $s'$. Concretely, the TD target for $Q(s, \texttt{up})$ is $r + \max_a Q(s', a)$ where $s'$ is the state inside the border




(a) Real state $s$

(b) Erroneous simulated transition from $s$ to $s'$

Figure 3.2: **An Erroneous Simulated Transition in *Bordered Grid-world***. From an observed state $s$, an imperfect model may generate a simulated transition of $s'$ for action `up`.

and $r$ is some prediction of the reward on the transition from $s$ to $s'$. The values of the states inside the border, $Q(s', \cdot)$, are arbitrary. If these have high value, this update may cause the agent to think that the best action is to 'run' into the walls. Consequently, the next time the agent is in state $s$, it will repeatedly take the actions `up` and get 'stuck'.

Moreover, the values of the $s'$ may only slowly change from their initialised values as they can only be changed by TD updates performed during further simulated transitions *from* them. Thus, the agent may be misled by the arbitrary initialised values of the border states for a long time. With this intuition in mind, in the next section we present the hypothesis at the centre of this thesis.

## 3.2 The Hallucinated Value Hypothesis

We claim:

> ***Planning temporal difference updates in Dyna style algorithms that update real state values towards simulated state values may impair learning of good control policies.***

| | 1-step Update | $n$-step Update |
|---|---|---|
| Successor Model | 1-step successor Dyna [6], [8], [11], [23] | $n$-step successor Dyna [3], [33] |
| Predecessor Model | 1-step predecessor Dyna [5], [17], [20], [21], [26] | $n$-step predecessor Dyna |

Table 3.1: Dyna Design Space

This hypothesis depends on the type of TD update performed during planning (cf. "...update real state values towards simulated state values" in our hypothesis). In the following section, we develop the 'design-space' of Dyna algorithms referred to in Section 2.4, and, based on our hypothesis, suggest which ones are likely to succeed or fail when using an imperfect environment model.

## 3.3 A Design-space of Dyna style Algorithms

As afore-mentioned in Section 2.4, we focus on two design aspects of Dyna: 1) whether the model is used to simulate the environment's dynamics forward or backward in time and 2) whether the TD updates used during planning are 1-step or multi-step. Table 3.1 graphically displays the dimensions along which we explore Dyna and names the algorithms we explore that occupy each quadrant of this design space. Furthermore, in the table we also cite examples in the literature of each type of algorithm.

Given this design-space, here we introduce concrete instantiations of algorithms in each quadrant of the design space. Moreover, we introduce $\beta$, a useful new parameter for Dyna style algorithms.

### 3.3.1 $\beta$-Prioritised Dyna

The generic algorithm is as follows: the agent explores the environment, updates the $Q$-function, and adds environment transitions $(s, a, r, s')$ to a prioritised planning queue subject to the TD error $\delta$ exceeding priority threshold $\rho$ (lines 4-8 of Algorithm 1). Then, for each real interaction with the environment, the agent performs a number of planning steps (lines 9-13 of Algorithm 1), and

**Algorithm 1** Generic Prioritised Dyna style Algorithm with $\beta$

**procedure** $\beta$-PRIORITISEDDYNA
    Initialise $Q$-function, planning queue $P$, priority threshold $\rho$, decay parameter $\beta$, learning rate $\alpha$; Load environment model $\mathcal{M}$
    **for** episode $= 1$ to $M$ **do**
        Observe state $s$ and select action $a = \pi(s)$
        Execute $a$ in environment and observe reward $r$ and next state $s'$
        Set $\delta = (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
        Update $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
        Add $(s, a, r, s', n = 0)$ to $P$ if $\delta \cdot \beta^n \geq \rho$
        **for** planning step $= 1$ to $N$ **do** ▷ This loop is deliberately vague here as it varies for each algorithm. Concrete pseudo-code is provided in the following subsections.
            Pop highest priority transition $\mathcal{T}$ which has $n = i$ in the transition tuple
            Simulate dynamics on $\mathcal{T}$ using $\mathcal{M}$ to yield $\mathcal{T}'$
            Calculate $\delta_{\mathcal{T}'}$ and update $Q$ with it
            Add $\mathcal{T}'$ with $n = i + 1$ back to $P$ subject to $\beta^{i+1} \cdot \delta_{\mathcal{T}'} \geq \rho$

it is here that our four instantiations differ. In each planning step, the highest priority transition is popped off the planning queue, its dynamics simulated using the environment model, and a TD-update performed on the simulated transition using one of the update rules corresponding to the Dyna design space quadrant of the particular algorithm. Finally, the simulated transition is added back to the planning queue according to its $\beta$-decayed priority.

$\beta$ helps control model iteration (where a model generates predictions off its own previous predictions). For instance, suppose a real environment transition $(s_t, a_t, r_t, s_{t+1})$ is popped off the priority queue and that planning generates some new transition $(s_{t+1}, a_{t+1}, \hat{r}_{t+1}, \hat{s}_{t+2})$ with associated TD-error $\delta_{t+1}$. If $\beta = 0$, the priority of this new transition is $\delta_{t+1} \cdot \beta = 0$, and if the priority threshold $\rho > 0$, this transition would not be added to the priority queue thereby halting iteration of the model's predictions. The priority queue would not contain any fully simulated transitions. At the other extreme, if $\beta = 1$, model iteration could theoretically continue unchecked possibly causing compounding model error as noted in prior work [27], [28], [30]. If $\beta$ is somewhere in between, it may possibly halt model iteration before compounding error became too egregious while still allowing the agent to benefit from longer planning trajectories as

recommended by Holland *et al.* [8].

Based on this generic algorithm, we now define the four algorithms we test. Moreover, with a view of our hypothesis, we speculate the ramifications of coupling these algorithms with imperfect models and make judgements about the success or failure of each.

### 3.3.2 1-step successor Dyna

**Algorithm 2** 1-step successor Dyna

**procedure** 1-STEP SUCCESSOR DYNA
  Initialise $Q$-function, planning queue $P$, priority threshold $\rho$, decay parameter $\beta$, learning rate $\alpha$; Load environment model $\mathcal{M}_s$
  **for** episode $= 1$ to $M$ **do**
    Observe state $s$ and select action $a = \pi(s)$
    Execute $a$ in environment and observe reward $r$ and next state $s'$
    Set $\delta = (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    Update $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
    Add $(s, a, r, s', n = 0)$ to $P$ if $\delta \cdot \beta^n \geq \rho$
    **for** planning step $= 1$ to $N$ **do**
      Pop highest priority transition $(s_t, a_t, r_t, s_{t+1}, n)$ from $P$ ▷ Note, some of the elements of the tuple could be previously simulated but we do not add ˆ for clarity. Furthermore, note that we subscript the transitions here for clarity as well so that it can be seen in which direction model simulations occur.
      **for** all actions $a \in \mathcal{A}$ **do**
        Simulate transition $(s_{t+1}, a, \hat{r}_{t+1}, \hat{s}_{t+2})$ using $\mathcal{M}_s$
        Set $\delta = (\hat{r}_{t+1} + \gamma \max_{a'} Q(\hat{s}_{t+2}, a') - Q(s_{t+1}, a))$
        Update $Q(s_{t+1}, a) \leftarrow Q(s_{t+1}, a) + \alpha \cdot \delta$
        Add $(s_{t+1}, a, \hat{r}_{t+1}, \hat{s}_{t+2}, n + 1)$ to $P$ if $\delta \cdot \beta^{n+1} \geq \rho$

Possibly the most common type of instantiation of the Dyna framework is 1-step successor Dyna [6], [8], [11] (we abbreviate to $FQ1$). Introduced by Sutton [23], as seen in Algorithm 2, given state $s_t$, the model is used to simulate environment dynamics forward in time to yield $\hat{r}_{t+1}, \hat{s}_{t+2}$ (note, that ˆ above elements in transition tuples indicate that these elements are generated by a model) for some action $a_t$ yielding a simulated transition $(s_t, a_t, \hat{r}_{t+1}, \hat{s}_{t+1})$ . Then, a 1-step $Q$-learning update is performed on this simulated transition :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\hat{r}_{t+1} + \gamma \max_{a_{t+1}} Q(\hat{s}_{t+1}, a_{t+1}) - Q(s_t, a_t)).$$

Furthermore, if the TD-error, $\delta$, of this transition is sufficiently high, it may be added back to the planning queue. Indeed, this simulated transition may even be popped from the planning queue in the future. In this case, the model then generates $(\hat{s}_{t+1}, a_{t+1}, \hat{r}_{t+2}, \hat{s}_{t+2})$, effectively continuing the 'rollout' from $s_t$ (note that when $\beta = 0$, this algorithm is equivalent to Dyna-Q [23]). Figure 3.3 shows a graphical representation of this process.

If $s_t$ is a *real* state, the planning TD update involves moving the estimate of a real state towards the sum of an immediate reward $\hat{r}_{t+1}$ and the value of a successor state $\hat{s}_{t+1}$. Thus, this planning update causes the value of real states to be updated towards the values of simulated states. According to our hypothesis, we expect to see a Dyna agent planning using this update to fail to learn a control policy.

Consider how this approach performs in *Bordered Gridworld*. It may be immediately clear that the same problematic transition discussed in Section 3.1.2 will be generated — the value of moving toward the border may be erroneously raised as a result of (potentially) arbitrary high values of the unreachable border states. However, unlike in Dyna-Q, this error may be corrected over time. Since the model is used to simulate multiple steps, it may not only simulate transitions from reachable states to border states, but also from border states back to reachable states as shown in Figure 3.4. As such, the values of impossible states *may be updated*. In this simple example, the values of border states may converge to their true value, however, this may take a great deal of time, likely harming rather than helping sample complexity.

### 3.3.3 $n$-step successor Dyna

An extension of 1-step successor Dyna is $n$-step successor Dyna [3], [33] (which we abbreviate to $FQN$) as shown in Algorithm 3. As suggested by the name, here we use multi-step trajectories in the TD update. Given some transition tuple $(\{s_{t+j}\}_{j=0}^{k-1}, \{a_{t+j}\}_{j=0}^{k-1}, \{r_{t+j}\}_{j=0}^{k-1}, s_{t+k})$ (note $\{\}_{j=0}^{k-1}$ indicates a trajectory or states, rewards, etc.) off the planning queue, the environment model generates, for some $a_t$, the successor state $\hat{s}_{t+k+1}$ of $s_{t+k}$ along with reward $\hat{r}_{t+k}$. A $n$-step $Q$-learning update is performed on the newly generated

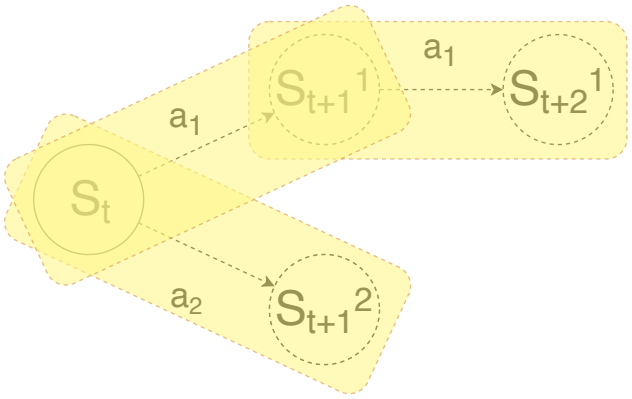



Figure 3.3: **Planning with 1-step successor Dyna**. The model is used to generate successor states $s^1_{t+1}$ and $s^2_{t+1}$ of $s_t$ (along with reward of these transitions) for actions $a_1$ and $a_2$, respectively. Then, a 1-step TD update is applied (as depicted by the yellow rectangles). Shown also is a simulated transition from $s^1_{t+1}$ to $s^1_{t+2}$. This update would happen occur if the priority $\delta_{t+1} \cdot \beta^1$ of the transition $(s_t, a_1, r_{t+1}\text{not shown}, s^1_{t+1})$ exceeds $\rho$. That is, the transition $(s_t, a_1, r_{t+1}\text{not shown}, s^1_{t+1})$ would have been added back to the planning queue and used in subsequent planning phases.

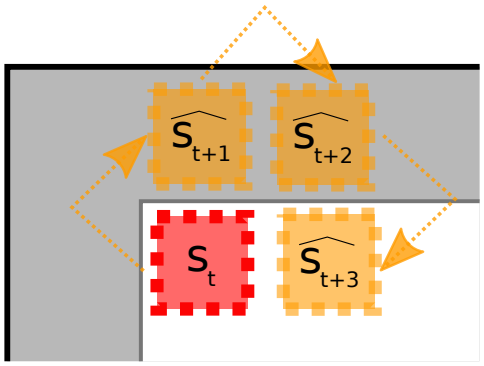



Figure 3.4: **Planning trajectories generated by 1-step successor Dyna on *Bordered Gridworld***

**Algorithm 3** $n$-step successor Dyna

**procedure** $n$-STEP SUCCESSOR DYNA
  Initialise $Q$-function, planning queue $P$, priority threshold $\rho$, decay parameter $\beta$, learning rate $\alpha$; Load environment model $\mathcal{M}_s$
  **for** episode $= 1$ to $M$ **do**
    Observe state $s$ and select action $a = \pi(s)$
    Execute $a$ in environment and observe reward $r$ and next state $s'$
    Set $\delta = (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    Update $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
    Add $(s, a, r, s', n = 0)$ to $P$ if $\delta \cdot \beta^n \geq \rho$
    **for** planning step $= 1$ to $N$ **do**
      Pop highest priority transition $(s_t, a_t, r_t, s_{t+1}, n)$ from $P$ ▷ Note, some of the elements of the tuple could be previously simulated but we do not add ˆ for clarity. Furthermore, note that we subscript the transitions here for clarity as well so that it can be seen in which direction model simulations occur. Finally, in $n$-step successor Dyna note that we could be dealing with trajectories of $s, a, r$ in the tuples popped off $P$ which are not shown for clarity.
      **for** all actions $a \in \mathcal{A}$ **do**
        Simulate transition $(s_{t+1}, a, \hat{r}_{t+1}, \hat{s}_{t+2})$ using $\mathcal{M}_s$
        Set $\delta = (r_t + \gamma \cdot \hat{r}_{t+1} + \gamma^2 \max_{a'} Q(\hat{s}_{t+2}, a') - Q(s_t, a_t))$
        Update $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \rho_{t:t+2}\delta$ ▷ $\rho_{t:t+2}$ is the importance sampling ratio of this trajectory
        Add $(\{s_t, s_{t+1}\}, \{a_t, a\}, \{r_t, \hat{r}_{t+1}\}, \hat{s}_{t+2}, n{+}1)$ to $P$ if $\delta \cdot \beta^{n+1} \geq \rho$

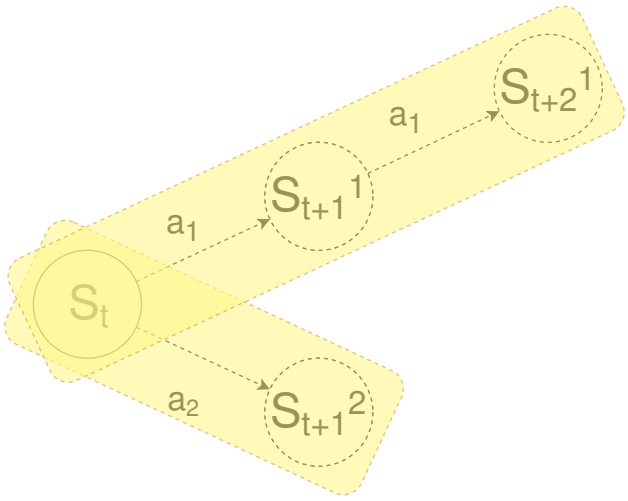



Figure 3.5: **Planning with $n$-step successor Dyna**. One trajectory generated during planning is from $s_t$ to $s^1_{t+1}$ to $s^1_{t+2}$ (along with reward). A 2-step TD update applied (as depicted by the yellow rectangles). Note, in practice we generate such trajectories for all actions from a particular state.

transition:

$$Q(s_t, a_t) \leftarrow \alpha \left( \sum_{i=0}^{n} \gamma^i \hat{r}_{t+i} + \gamma^{n+1} \max_{a_{t+n+1}} Q(\hat{s}_{t+n+1}, a_{t+n+1}) - Q(s_t, a_t) \right) + Q(s_t, a_t).$$

Importantly, this $n$-step TD update is weighted by importance sampling ratio (which is why we store the sequences $\{s_{t+j}\}$ , $\{a_{t+j}\}$). Finally, just as with $FQ1$, the newly generated transition may be added back to the planning queue if its TD-error $\delta_{t+n+1}$ is sufficiently high, and the planning 'rollout' continued from $\hat{s}_{t+n+1}$. A graphical representation of this TD update is shown in Figure 3.5.

This algorithm falls within the purview of our hypothesis: it updates the values of real states to simulated states. Thus, we expect it to struggle to learn a good control policy. In fact, it may be even worse than $FQ1$. To see why, again consider performance on *Bordered Gridworld*. Figure 3.6 shows two simulated transitions generated during planning. Firstly, as illustrated previously, the 1-step transition (top left of Figure 3.6) will be problematic. Secondly, more troublsome is the fact that $FQN$ will also perform TD updates from states deep in the 'interior' of the reachable states to the border (bottom left of Figure 3.6). Thus, not only will the values of states adjacent to the

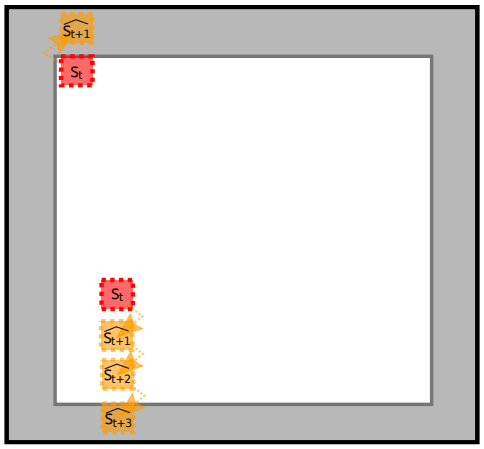



Figure 3.6: **Planning trajectories generated by $n$-step successor Dyna on *Bordered Gridworld*.**

border be directly updated to the imagined values of border states but so will the values of states deep in the reachable set. This may lead to even more detrimental outcomes for the control policy than $FQ1$.

Moreover, unlike $FQ1$, the design of $FQN$ rules out the possibility of the values of unreachable border states being updated. This is due to the way trajectories are stored in the planning queue. From a given real, observed state $s_t$, model simulates dynamics forward in time to yield $\hat{s}_{t+1}$. This results in an update to the value of the real state $s_t$. Moreover, supposing the 'rollout' is continued from $\hat{s}_{t+1}$, the model generates $\hat{s}_{t+2}$, but the update from this trajectory again only updates the value of $s_t$ — the value of $\hat{s}_{t+1}$ is not updated. Indeed, as can be seen from the trajectory shown in the bottom left of Figure-3.6, the multi-step trajectories result in updates only to real, observed states and not to border states.

So far we have explored algorithms covering half of our Dyna design space: algorithms that use models to simulate environment dynamics forward in time and perform TD updates on such simulated transitions. Clearly, these algorithms fall within the scope of our hypothesis and we have described plausible ways in which they may fail. In the next sections, we examine algorithms that use models to simulate dynamics *backward* in time. At first glance, one would think such algorithms would not update real state values to

potentially erroneous simulated state values and therefore not fail. However, as we show below, this is not necessarily the case.

### 3.3.4 1-step predecessor Dyna

**Algorithm 4** 1-step predecessor Dyna

**procedure** 1-STEP PREDECCESSOR DYNA
  Initialise $Q$-function, planning queue $P$, priority threshold $\rho$, decay parameter $\beta$, learning rate $\alpha$; Load environment model $\mathcal{M}_p$
  **for** episode $= 1$ to $M$ **do**
    Observe state $s$ and select action $a = \pi(s)$
    Execute $a$ in environment and observe reward $r$ and next state $s'$
    Set $\delta = (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    Update $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
    Add $(s, a, r, s', n = 0)$ to $P$ if $\delta \cdot \beta^n \geq \rho$
    **for** planning step $= 1$ to $N$ **do**
      Pop highest priority transition $(s_t, a_t, r_t, s_{t+1}, n)$ from $P$ ▷ Note, some of the elements of the tuple could be previously simulated but we do not add ˆ for clarity. Furthermore, note that we subscript the transitions here for clarity as well so that it can be seen in which direction model simulations occur.
      **for** all actions $a \in \mathcal{A}$ **do**
        Simulate transition $(\hat{s}_{t-1}, a, \hat{r}_{t-1}, s_t)$ using $\mathcal{M}_p$
        Set $\delta = (\hat{r}_{t-1} + \gamma \max_{a'} Q(s_t, a') - Q(\hat{s}_{t-1}, a))$
        Update $Q(\hat{s}_{t-1}, a) \leftarrow Q(\hat{s}_{t-1}, a) + \alpha \cdot \delta$
        Add $(\hat{s}_{t-1}, a, \hat{r}_{t-1}, s_t, n + 1)$ to $P$ if $\delta \cdot \beta^{n+1} \geq \rho$

1-step predecessor Dyna (abbreviated to $BQ1$) is the predecessor model analogue of $FQ1$. Algorithm 4 shows pseudocode of the algorithm. As shown in Figure 3.7, here, given some state $s_t$, environment dynamics are simulated *backward* in time. Thus, for action $a_{t-1}$ we obtain $\hat{s}_{t-1}, \hat{r}_{t-1}$. A 1-step $Q$-learning update is performed on this transition:

$$Q(\hat{s}_{t-1}, a_{t-1}) \leftarrow \alpha(\hat{r}_{t-1} + \gamma \max_{a_t} Q(s_t, a_t) - Q(\hat{s}_{t-1}, a_{t-1})) + Q(\hat{s}_{t-1}, a_{t-1}).$$

Furthermore, this transition may be added back to the planning queue (modulo $\rho$). If $\beta = 0$, this algorithm becomes Predecessor Dyna-Q [23] while $\beta = 1$ makes this equivalent to Prioritised Sweeping [17], [21].

Although at face value it would seem that this algorithm does not update real state values to simulated states values, in fact, it does so and therefore

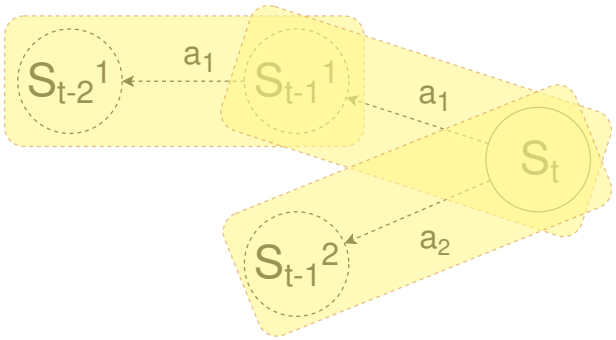
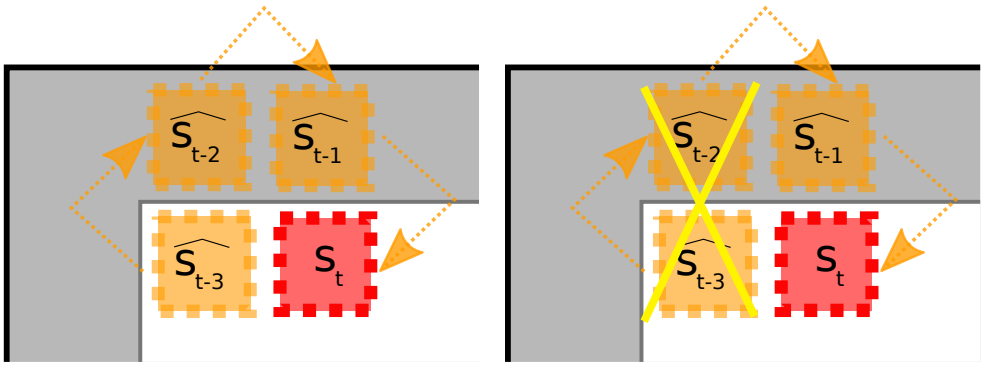



Figure 3.7: **Planning with 1-step predecessor Dyna**. From a given state $s_t$, environment dynamics are simulated backward in time to yield $s^1_{t-1}$ and $s^2_{t-1}$ (along with reward). A 1-step TD update applied *from* these simulated states. Moreover, as shown by the yellow rectangle over $s^1_{t-1}$ and $s^1_{t-2}$, the predecessor state 'rollout' can continue just as in $FQ1$.

according to our hypothesis it may cause learning failure. Figure 3.8 (a) shows a trajectory of transitions from the real state $s_t$ generated by the predecessor model during planning: $s_t, \hat{s}_{t-1}, \hat{s}_{t-1}, \hat{s}_{t-1}$. The planning transition from $s_{t-1}$ to $s_t$ updates the value of $Q(\hat{s}_{t-1}, \texttt{down})$, but as the agent can never reach $\hat{s}_{t-1}$, the policy there is irrelevant and similarly for $Q(\hat{s}_{t-2}, \texttt{right})$. However, the planning transition from $\hat{s}_{t-3}$ to $\hat{s}_{t-2}$ does impact the control policy as $\hat{s}_{t-3}$ is a reachable state. If the value of $\hat{s}_{t-2}$ is arbitrarily high, updating the value $\hat{s}_{t-3}$ towards $\hat{s}_{t-2}$ will cause the value of $Q(\hat{s}_{t-3}, \texttt{up})$ to increase. The next time the agent reaches $\hat{s}_{t-3}$, it may choose the `up` action and repeatedly 'run' into the wall.

That said, there is a key difference to $FQ1$. With a successor model, when the model is limited to simulating a single step trajectory (i.e., when $\beta = 0$), the result is catastrophic — the value of the border states perpetually misleads the agent. However, in Predecessor Dyna-Q, when a predecessor state model simulates a single step backward in time from a reachable state (setting $\beta = 0$), the target state of every update is a state the agent actually experienced. Thus, in this special case, the values of erroneous states *cannot* contaminate the values of reachable states. Predecessor Dyna-Q is, therefore, robust to model error, though it cannot make use of longer simulated trajectories.

Is there an algorithm that has the benefits of planning while avoiding




(a) $\beta \neq 0$ (b) $\beta = 0$

Figure 3.8: **Planning trajectories generated by 1-step predecessor Dyna on *Bordered Gridworld*.** (a) If $\beta \neq 0$, this algorithm may perform updates from real to simulated states. (b) If $\beta = 0$, it only generates 1-step trajectories to update simulated states to a real states.

updates from real to simulated states for any value of $\beta$? The answer is the affirmative. In the next section, we introduce $n$-step predecessor Dyna — an algorithm designed not to update real state values to simulated state values.

### 3.3.5 $n$-step predecessor Dyna

We have identified existing algorithm examples in three quadrants of the design space for Dyna planning depicted in Table 3.1. We are not aware of existing examples in the fourth quadrant – algorithms that use predecessor models and $n$-step TD updates – but we shall see that it has particularly attractive properties in the face of model error.

In $n$-step predecessor Dyna, as shown in Algorith 5, we apply multi-step updates always with the selected state $s_t$ as the *bootstrap target*. That is, for a particular transition tuple $(\{\hat{s}_{t-j}\}_{j=k}^{1}, \{a_{t-j}\}_{j=k}^{1}, \{\hat{r}_{t-j}\}_{j=k}^{1}, s_t)$, we simulate dynamics backwards from $\hat{s}_{t-k}$ to get state $\hat{s}_{t-(k+1)}$ and reward $\hat{s}_{t-(k+1)}$ for action $a_{t-(k+1)}$. Thus, we generate the multi-step trajectory $(\{s_{t-j}\}_{j=k+1}^{1}, \{a_{t-j}\}_{j=k+1}^{1}, \{r_{t-j}\}_{j=k+1}^{1}, s_t)$ and perform the $n$-step TD update:

**Algorithm 5** $n$-step predecessor Dyna

**procedure** $n$-STEP PREDECESSOR DYNA
    Initialise $Q$-function, planning queue $P$, priority threshold $\rho$, decay parameter $\beta$, learning rate $\alpha$; Load environment model $\mathcal{M}_p$
    **for** episode $= 1$ to $M$ **do**
        Observe state $s$ and select action $a = \pi(s)$
        Execute $a$ in environment and observe reward $r$ and next state $s'$
        Set $\delta = (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
        Update $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
        Add $(s, a, r, s', n = 0)$ to $P$ if $\delta \cdot \beta^n \geq \rho$
        **for** planning step $= 1$ to $N$ **do**
            Pop highest priority transition $(s_t, a_t, r_t, s_{t+1}, n)$ from $P$ ▷ Note, some of the elements of the tuple could be previously simulated but we do not add ˆ for clarity. Furthermore, note that we subscript the transitions here for clarity as well so that it can be seen in which direction model simulations occur. Finally, in $n$-step predecessor Dyna note that we could be dealing with trajectories of $s, a, r$ in the tuples popped off $P$ which are not shown for clarity.
            **for** all actions $a \in \mathcal{A}$ **do**
                Simulate transition $(\hat{s}_{t-1}, a, \hat{r}_{t-1}, s_t)$ using $\mathcal{M}_p$
                Set $\delta = (\hat{r}_{t-1} + \gamma \cdot r_t + \gamma^2 \max_{a'} Q(s_{t+1}, a') - Q(\hat{s}_{t-1}, a))$
                Update $Q(\hat{s}_{t-1}, a_t) \leftarrow Q(\hat{s}_{t-1}, a_t) + \alpha \cdot \rho_{t-1:t+1}\delta$ ▷ $\rho_{t-1:t+1}$ is the importance sampling ratio of this trajectory
                Add $(\{\hat{s}_{t-1}, s_t\}, \{a, a_t\}, \{\hat{r}_{t-1}, r_t\}, s_{t+2}, n+1)$ to $P$ if $\delta \cdot \beta^{n+1} \geq \rho$

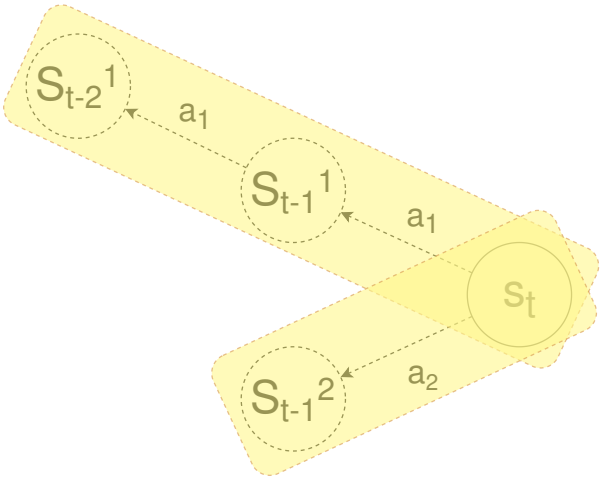



Figure 3.9: **Planning with $n$-step Predecessor Dyna**. Multi-step trajectories are generated from a particular state. In this figure, we generate one 2-step predecessor state trajectory and a 1-step predecessor state trajectory. The yellow rectangles represent the $n$-step TD updates that are performed. Notice, if $\beta = 0$, *BQN* is equivalent to *BQ1*.

$$Q(\hat{s}_{t-(k+1)}, a_{t-(k+1)}) \leftarrow \alpha \left( \sum_{l=k+1}^{1} \gamma^{k+1-l} \hat{r}_{t-j} + \gamma^{k-l} \max_{a_t} Q(s_t, a_t) - Q(\hat{s}_{t-k-1}, a_{t-k-1}) \right) + Q(\hat{s}_{t-k-1}, a_{t-k-1})$$

Within the design of $\beta$-Prioritits ed Dyna (Algorithm 1), this update always results in simulated state values being updated toward real state values. This is because of the way transitions are initially added to the planning queue. When the agent is exploring the environment, if the TD-error of a transition is sufficiently high, it will be added to the planning queue. Thereafter, during planning, $n$-step predecessor Dyna generates trajectories of predecessor transitions and updates the values of states at the end of these predecessor transitions to the value of the real experienced state $s_t$. Thus, the values of real states are never updated to the values of simulated states (the target is *always* $s_t$) and hallucinated state values cannot contaminate reachable state values. As seen in Figure 3.10 if these states correspond to real, reachable states then the planning update is useful. On the other hand, if these states are erroneous simulated states, any value function update to these states does

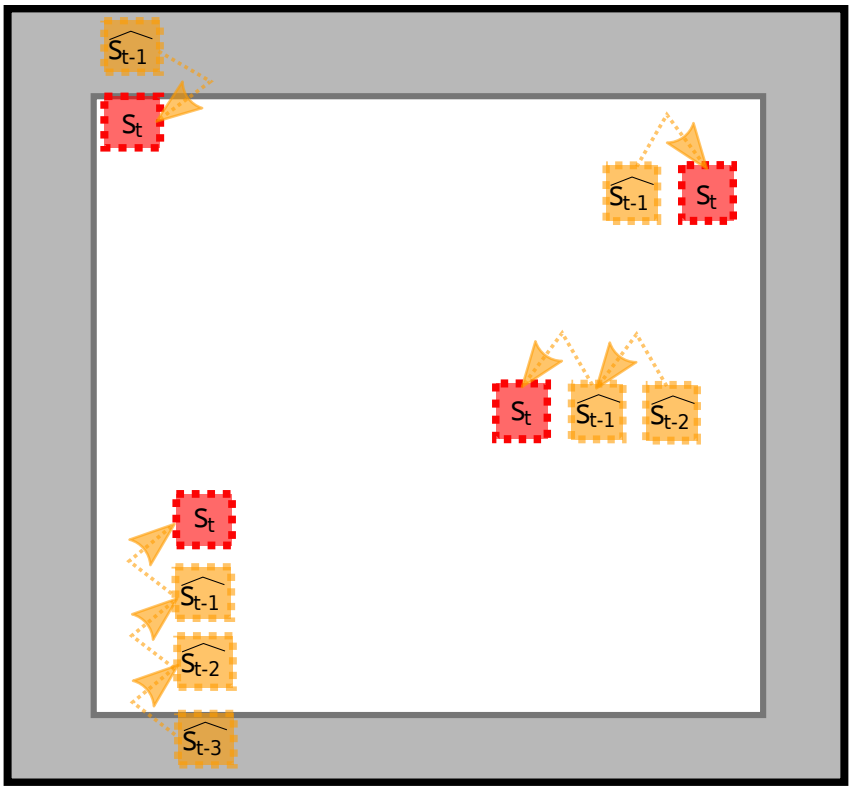



Figure 3.10: **Planning trajectories generated by $n$-step predecessor Dyna on *Bordered Gridworld***

not harm the agent's policy as the agent can never reach these states to make a bad action choice. An added bonus of this algorithm is that it allows for planning to generate longer trajectories of simulated transitions than 1-step predecessor Dyna with $\beta = 0$. As per the work of Holland *et al.* [8], this ought to have a beneficial impact on learning.

# Chapter 4

# The *Hallucinated Value Hypothesis* in *Bordered Gridworld*

We report experimental results of running the algorithms defined in the previous chapter on *Bordered Gridworld*. First, we present results of 1-step successor Dyna & $n$-step successor Dyna. Then, we evaluate the predecessor Dyna algorithms 1-step predecessor Dyna & $n$-step predecessor Dyna. The results of these experiments seem to affirm our hypothesis: algorithms which update real state values to simulated state values fail in the face of erroneous model predictions.

## 4.1 Experiment Methodology

The experiments in this chapter are conducted on *Bordered Gridworld*. To test the *Hallucinated Value Hypothesis*, we make specific experiment choices as detailed below.

**Value Function**

We use a tabular value function to represent $Q$-values for both reachable and unreachable states. That is, we define the value function table to contain capacity to be able to represent $Q(s, \cdot)$ for $s \in \mathcal{S}_r$ and $s \in \mathcal{S}_{r^c}$.

In Chapter 3 we explained that temporal difference (TD) updates from real to simulated states may mislead the agent's control policy. Our intuition

for this is that simulated states may have values that incorrectly influence the agent's policy away from real value in the environment. Thus, in these experiments, to 'maximally' mislead the agent away from real environment value, we optimistically initialise the value function to 1 for all $s \in \mathcal{S}$. In particular, this includes initialising the values of $s \in \mathcal{S}_{r^c}$ to 1 as well. Thus, it is likely that real environment value may be over-shadowed by spurious value of border states.

**Environment Models**

Unlike the standard Dyna framework where an environment model is learned online (i.e., as the agent is interacting with the world), in these experiments, to study our hypotheses, we use *predefined* environment dynamics models with known fixed errors.

The *successor* state model $\mathcal{M}_s$ simulates environment dynamics forward in time: $\mathcal{M}_s : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S} \times \mathcal{R}$. That is, $\mathcal{M}_s(s, a) \mapsto r, s'$, where state $s'$ and reward $r$ succeed state $s$ given action $a$. As *Bordered Gridworld* is deterministic, the model perfectly captures the environment's dynamics — all reward and state predictions are perfectly accurate. However, to fabricate the effect of an imperfect environment model which produces imagined states that cannot be realised, $\mathcal{M}_s$ produces predictions to states in the border; the model makes the agent believe that transitions between states $s \in \mathcal{S}_r$ and $s \in \mathcal{S}_{r^c}$ are possible (as well as transitions from border states to border states).

The *predecessor* state environment model, $\mathcal{M}_p$, is analogous to the successor state environment model, $\mathcal{M}_s$. $\mathcal{M}_p$ is a mapping from a state $s' \in \mathcal{S}$ to a state $s \in \mathcal{S}$ and a reward $r \in \mathbb{R}$ given an action $a \in \mathcal{A}$ — $\mathcal{M}_p(s', a) \mapsto s, r$. Intuitively, given a state $s'$ and an action $a$, $\mathcal{M}_p$ returns a prediction of the state $s$ which would have been the state that led to $s'$ if action $a$ had been taken from $s$.

The type of error fixed into $\mathcal{M}_s$ and $\mathcal{M}_p$ may be plausible in learned environment models. Consider a model which aggressively generalises. It will see that in most states in *Bordered Gridworld* ($|S_r|$ is larger $|S_r^c|$) that when the agent takes the action `up`, for example, the position of the black square

representing the agent moves up a little on the screen. It may generalise this behaviour across all states in $\mathcal{S}$. This will lead to model error where the agent generates transitions from states near the top border to states inside the top border.

**Experiment Settings**

All experimental results are averages over 10 runs. The dark lines in the plots represent means while the shaded region represents (which may be smaller than the width of the mean performance line and therefore not visible) standard errors.

We use the following parameters for this study:

$$\alpha \in \{0.023, 0.03, 0.05, 0.09, 0.11, 0.13, 0.15, 0.18, 0.19\}$$

$\beta = 0.9$ unless otherwise stated

$\epsilon = 0.1$

$\gamma = 0.9$

$\rho = 0.1$

All graphs in the learning curves that follow are for the best choice of $\alpha$ for the particular algorithms shown.

## 4.2 Dyna style Algorithms with Successor Models

### 4.2.1 1-step Successor Dyna

Learning curves on *Bordered Gridworld* for $Q$-learning and 1-step successor Dyna (abbreviated to $FQ1$) are shown in Figure 4.1. The plot shows cumulative reward as a function of cumulative experience in the environment. As reward is only gained when the agent reaches the goal state, cumulative reward is, in this case, equivalent to the number of episodes the agent has completed. The black dashed line on the plot represents the performance of a hypothetical agent that employs the optimal policy from the start; the orange line shows

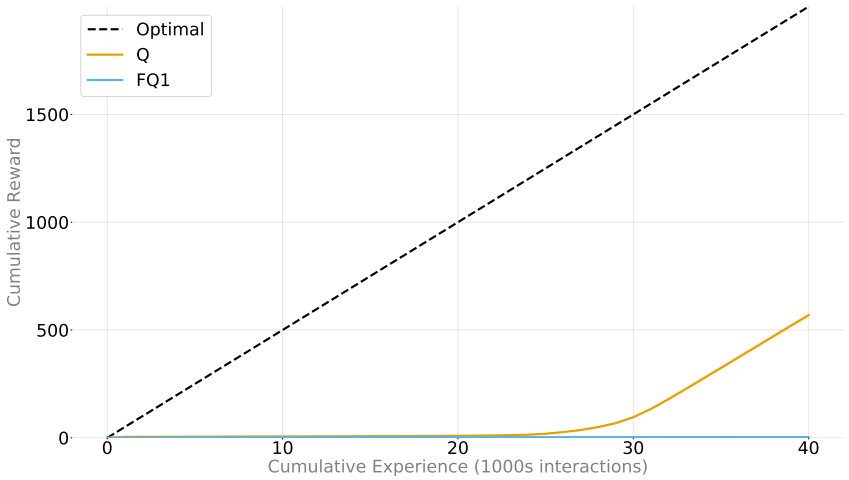
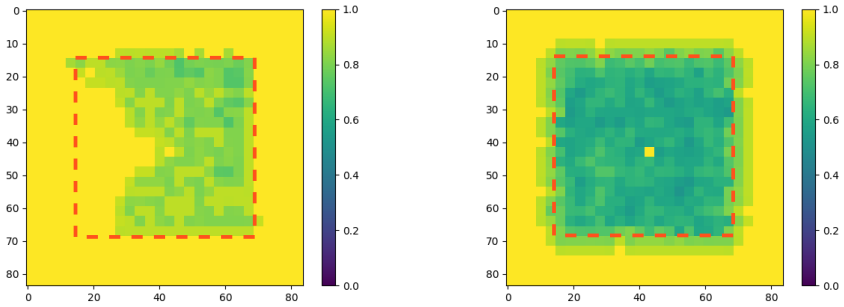



Figure 4.1: **Learning Curves of *Q*-learning and *FQ1* on *Bordered Gridworld*.**

the performance of the best model-free $Q$-learner, and the turquoise line shows the performance of the best *FQ1* agent.

The model-free $Q$-learning agent's learning curve is typical of an agent which learns how to solve this task. It is flat initially as the agent explores the environment and does not reach the goal frequently. As learning progresses and the agent learns improved policies the gradient of the learning curve increases until it almost matches the gradient of the optimal policy (note, it does not match the gradient of the optimal policy as $\epsilon \neq 0$). The learning curve of the *FQ1* agent, however, reflects a failure to learn *Bordered Gridworld*. After accumulating some reward in the initial episodes (when optimistic initialisation still holds), the agent's learning curve flattens out indicating that it is not completing any more episodes — it is stuck either running into a wall or exhibiting other similar behaviour preventing it from reaching the terminal state before time-out. Thus, this experiment seems to support our hypothesis that planning updates that change the values of real states towards simulated states cause failure.

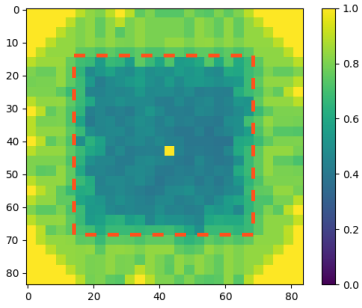
Why does the planning update of *FQ1* cause it to fail in learning this task? Figure 4.2 shows plots of $\max_a Q(s, a)\ \forall s \in \mathcal{S}$. In the initial stages of learning, Figure 4.2 (a), we see that the value for the border states $s \in \mathcal{S}_{r^c}$ and many of the reachable states $s \in \mathcal{S}_r$ are high due to optimistic initialisation. As learning



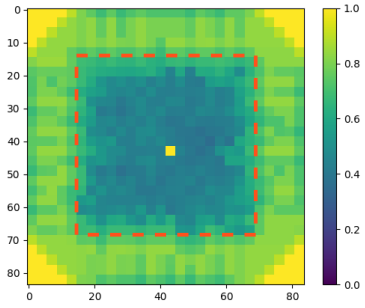
(a) 1000 steps



(b) 5000 steps



(c) 20000 steps



(d) 50000 steps

Figure 4.2: **Heatmaps of** $\max_a Q(s,a)\ \forall s \in \mathcal{S}$ **for** $FQ1$ **on *Bordered Gridworld*. The dashed red line shows the border.**

progresses, the values of the internal states $s \in \mathcal{S}_r$ are driven down but the values of the unreachable border states remain high (Figure 4.2 (b)). The key phenomenon is shown in Figure 4.2 (c) where the values of states *close* to the border are high, and indeed this is even more evident in Figure 4.2 (d). The values of the states close to the border are increased due to planning updates in *FQ1* which updates them to the optimistic values of states in the border. Indeed, this phenomenon causes the agent 'run into the walls' and thus fail to learn this task.

The results shown are for *FQ1* with $\beta = 0.90$. Is there any difference for different values of $\beta$? In fact, the performance of *FQ1* remains the same no matter the value of $\beta$. Even if $\beta = 0$ reachable states adjacent to the border are added to the planning queue and simulating forward from these states will result in $Q$-values of these states being misled by the values of border states.

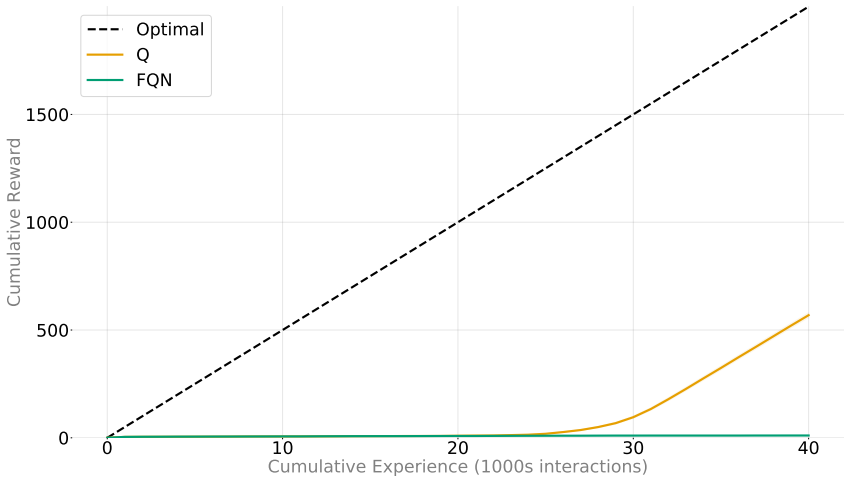


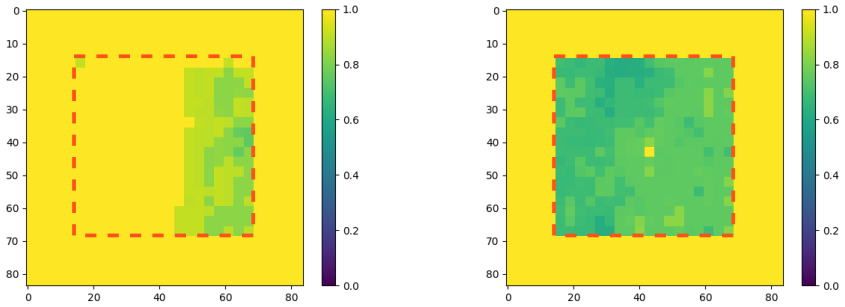

Figure 4.3: Learning Curves of ***Q*-learning and *FQN* on *Bordered Gridworld*.**

We have established that planning with 1-step successor state TD update causes failure. What about $n$-step successor state TD updates (which, again, update real to simulated states)? Can this TD update circumvent the issues of *FQ1*? We evaluate this algorithm in the next section.

### 4.2.2 $n$-step Successor Dyna

As can be seen in Figure 4.3, the $n$-step successor Dyna agent (abbreviated to *FQN*), represented by the green learning curve, fails to learn *Bordered Gridworld*. Similarly to *FQ1*, the agent gathers some initial reward but fails to collect any more. This is indicative of a policy that is causing the agent to either run into walls or circle around the terminal state. On the other hand, the model-free agent, $Q$ (represented by the orange curve), learns the optimal policy. Thus, the same conclusions as in *FQ1* hold: the agent's failure to learn this task is seemingly a consequence of the planning update.

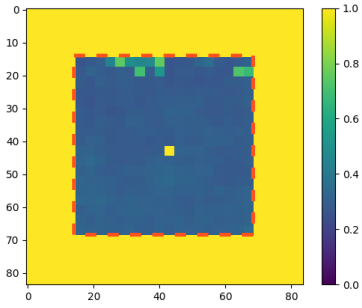
The *FQN* agent suffers from the same problem as the *FQ1* agent. The TD updates effectuated in planning cause the agent to learn a policy inimical to learning the task. Namely, planning updates that use the optimistically initialised values of border states in the TD target precipitate a policy where the agent repeatedly 'runs' into the wall. Evidence of this phenomenon is apparent in Figure 4.4 showing plots of $\max_a Q(s,a)\ \forall s \in \mathcal{S}$. As with the *FQ1*



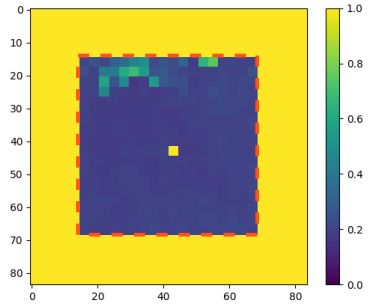
(a) 1000 steps



(b) 5000 steps



(c) 20000 steps



(d) 50000 steps

Figure 4.4: **Heatmaps of** $\max_a Q(s,a)\ \forall s \in \mathcal{S}$ **for** *FQN* **on *Bordered Gridworld*. The dashed red line shows the border.**

agent, the *FQN* agent's value function is corrupted by planning updates; the values of states near the border are incorrectly raised to values of border states thus adversely affecting the control policy.

In sum, this section shows evidence supporting the *Hallucinated Value Hypothesis*. Both algorithms which update the values of real states towards the values of simulated states fail to learn in *Bordered Gridworld*. Once the action values of real states are updated towards the values of unreachable optimistically initialised states, the agent simply keeps 'running' into a wall and fails to gather any further reward.

In the next section, we investigate 1-step predecessor Dyna and $n$-step predecessor Dyna. The planning TD updates are the converse of successor state TD planning updates: they move the values of simulated predecessor states towards the values of observed real states. Ostensibly, since this update

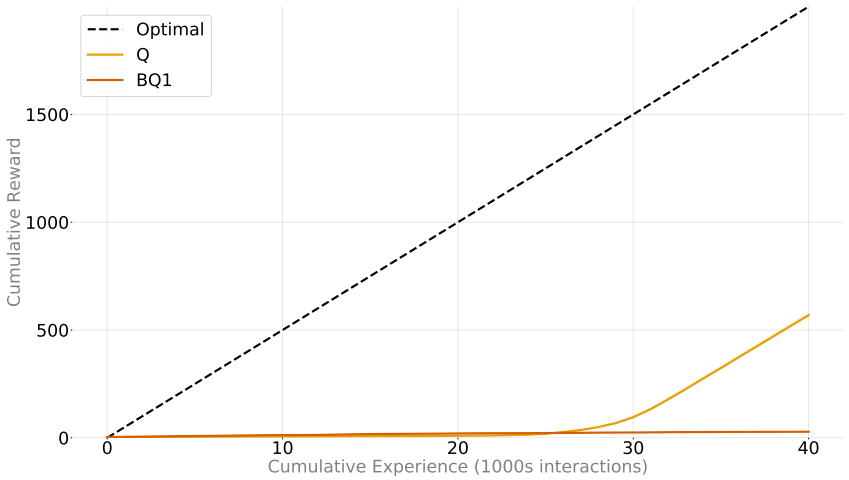



Figure 4.5: **Learning curves of *Q*-learning and *BQ1* with $\beta > 0$ on *Bordered Gridworld*.**

does not conform to our hypothesis, we would expect model-based agents that use this TD update to not suffer from the same problem.

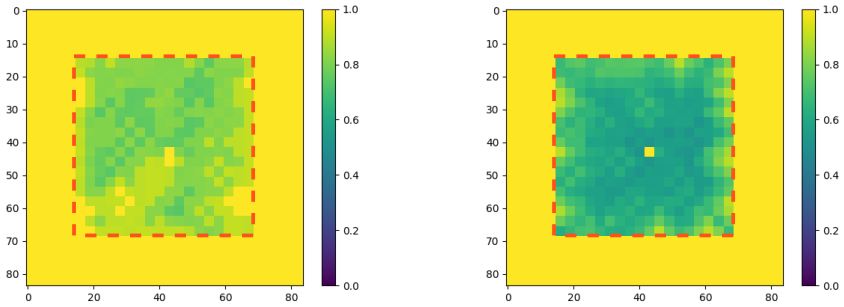
## 4.3 Dyna style Algorithms with Predecessor Models

### 4.3.1 1-step Predecessor Dyna

In the preceding sections, we demonstrated that successor state planning fails when the the planning process updates the values of real states towards simulated states. Here, we first show that, under specific settings of $\beta$, this issue plagues 1-step predecessor Dyna too.

As can be seen from the curves in Figure 4.5, the *BQ1* agent accumulates some reward and, in fact, outpaces $Q$-learning briefly (the red line is above the orange line in the plot). However, after approximately 25,000 cumulative steps the learning curve of *BQ1* plateaus whereas the learning curve of $Q$-learning rapidly improves. Indeed, the learning behaviour reflected in this plot is reminiscent of both *FQ1* and *FQN* — the agent accumulates some reward initially, but later the policy is sufficiently corrupted such that it is unable to solve *Bordered Gridworld*.

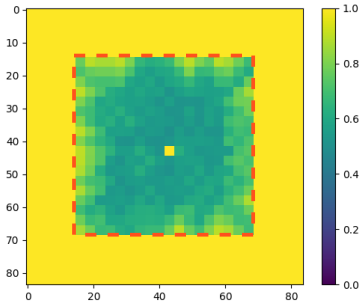
The plots $\max_a Q(s, a)\ \forall s \in \mathcal{S}$ shown in 4.6 reveal why $BQ1$ fails. The first



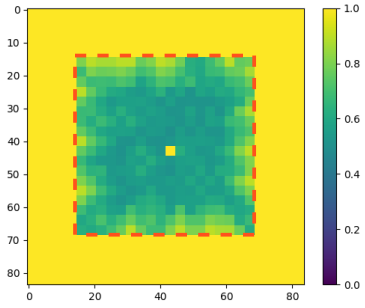
(a) 1000 steps



(b) 5000 steps



(c) 20,000 steps



(d) 50,000 steps

Figure 4.6: **Heatmaps of** $\max_a Q(s,a)\ \forall s \in \mathcal{S}$ **for** $BQ1$ **with** $\beta > 0$ **on *Bordered Gridworld*. The dashed red line shows the border.**

observation we wish to highlight is that unlike the direct forward planning agents *FQ1* and *FQN*, the value function of $BQ1$ agent, as shown in 4.6 (a), (b) & (c), is not immediately corrupted planning. The value function *seems* relatively accurate even up-to 20,000 steps. However, after some number of steps, $Q$-values close to the border gradually become high. It is these high $Q$-values that result in the agent failing to learn the task. We believe this phenomenon occurs because *BQ1* with $\beta > 0$ eventually results in updating real states values to simulated states values.

In Chapter 3 we suggested that $BQ1$ with $\beta = 0$ would not suffer from model error as it does not update real state values to simulated state values. Figure 4.7 shows the learning curve for *BQ1* with $\beta = 0$. Here, we see the first evidence of planning with erroneous models *not* causing failure. The red learning curve of *BQ1* with $\beta = 0$ rises earlier than the orange learning curve

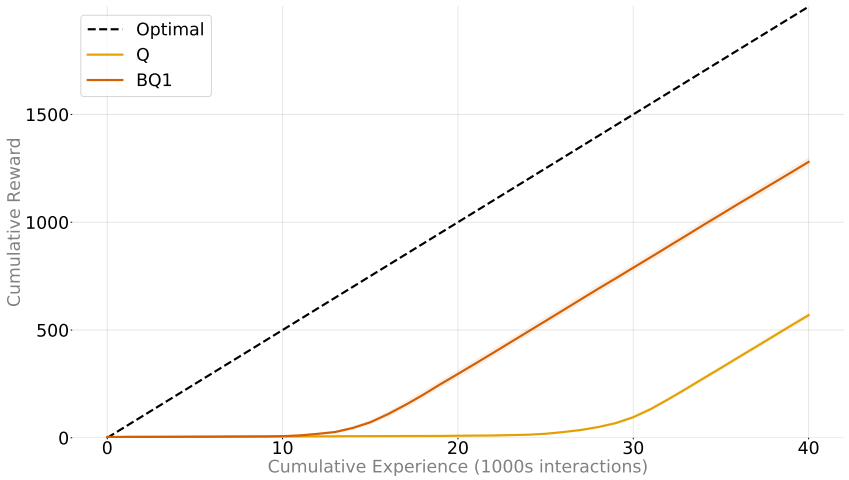



Figure 4.7: **Learning Curves of *Q*-learning and *BQ1* with $\beta = 0$ on *Bordered Gridworld*.**

of $Q$-learning, indicating that the agent learns the optimal policy sooner.

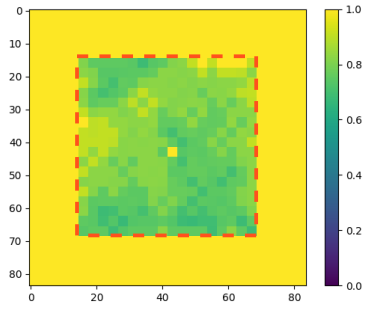
Indeed, as seen in Figure 4.8 showing plots of $\max_a Q(s, a)\ \forall s \in \mathcal{S}$ for *BQ1* with $\beta = 0$, the value function is uncorrupted. Values of states near the border are not influenced by the values of border states, and the values of reachable states only reflects real reward available in the environment.

While setting *BQ1* with $\beta = 0$ addresses the problems caused by updates from real to simulated states, it removes the possibility of multi-step planning. In the next section, we show we can entirely avoid the problem of real states' values being updated towards imagined states' values by utilising $n$-step predecessor state TD updates. This update allows us to derive the benefit of planning while seeming to avoid the risk of failure due to an imperfect model.

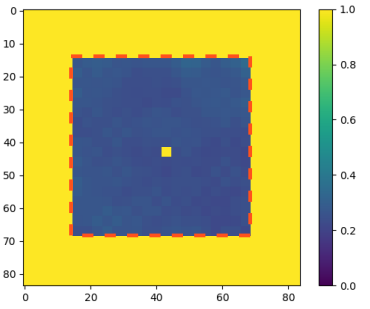
### 4.3.2 $n$-step Predecessor Dyna

As explained earlier, $n$-step predecessor Dyna (abbreviated to *BQN*) is guaranteed to *only* update the values of simulated states to the values of real states. Thus, as per the *Hallucinated Value Hypothesis*, we expect this algorithm not to struggle when updating with erroneous simulated transitions.

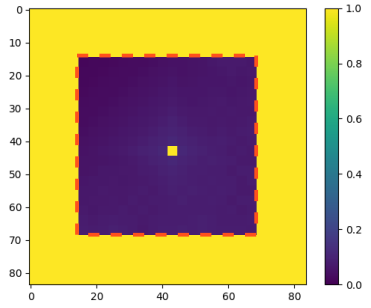
Figure 4.9 shows learning curves of *BQN* on *Bordered Gridworld*. Unlike all the other algorithms evaluated (except *BQ1* with $\beta = 0$), *BQN* does not fail to learn to solve *Bordered Gridworld* when $\beta > 0$. As shown by the dark



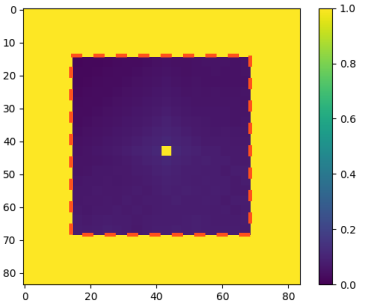

(a) 1000 steps




(b) 5000 steps




(c) 20000 steps




(d) 50000 steps

Figure 4.8: **Heatmaps of** $\max_a Q(s,a)\ \forall s \in \mathcal{S}$ **for** $BQ1$ **with** $\beta = 0$ **on *Bordered Gridworld*. The dashed red line shows the border.**

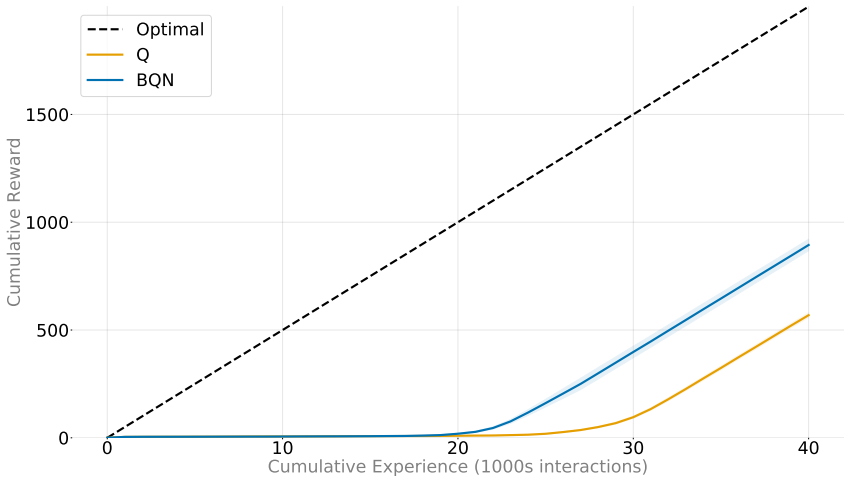
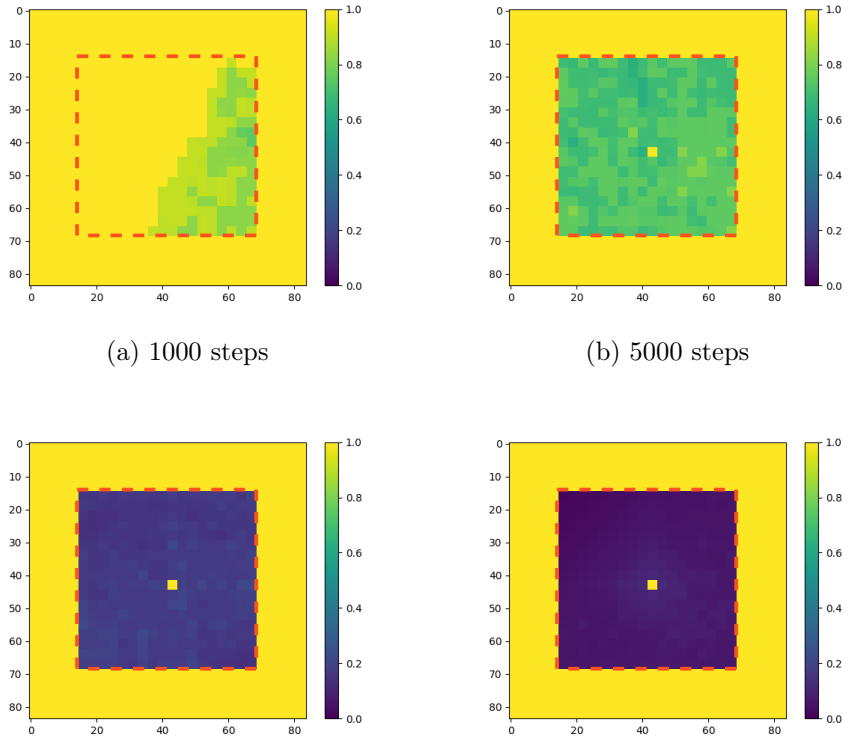



Figure 4.9: **Learning Curves of *Q*-learning and *BQN* on *Bordered Gridworld*.**

blue curve, for the first $20,000$ interactions with the environment, it does not accumulate much reward. However, after this point, it quickly learns the optimal policy as shown by the fact that the gradient of its learning curve closely matches the gradient of the learning curve of the optimal policy.

Furthermore, as demonstrated in Figure 4.10, $Q$-values of reachable states are not corrupted by values of states within the border. To wit, unlike $FQ1$, $FQN$ and $BQ1$ with $\beta > 0$, as the value of reachable states are *not* updated towards the values of states within the border, the value function never contains 'misleading' values. Accordingly, $BQN$ planning updates are robust in the presence of an erroneous environment model.

From the results presented in Section 4.2 and the results of *BQ1* with $\beta > 0$, we have shown that planning with an imperfect model fails when planning causes the values of real, reachable states to be updated towards the values of imagined, unreachable states. While one could argue that this effect is only problematic due to optimistic initialisation, we claim that in function approximator setting, it is impossible to enforce a regime of unreachable states not having pernicious $Q$-values. In the next section, we explore the *Hallucinated Value Hypothesis* in this setting.



(a) 1000 steps

(b) 5000 steps

(c) 20000 steps

(d) 50000 steps

Figure 4.10: **Heatmaps of** $\max_a Q(s,a)\ \forall s \in \mathcal{S}$ **for** *BQN* **on *Bordered Gridworld*. The dashed red line shows the border.**

# Chapter 5

# ... and in *The Wild*

Chapter 4 showed experimental results supporting the *Hallucinated Value Hypothesis* on *Bordered Gridworld*. Those experiments were, however, carefully designed to explore the hypothesis: the environment model was designed to generate flawed simulated transitions, and the value function explicitly contained capacity to represent arbitrary values of border states.

This chapter explores whether evidence for our hypothesis can be observed 'in *The Wild*' — in settings not explicitly constructed to highlight the phenomenon. We evaluate the hypothesis on three reinforcement learning (RL) benchmarks: *Cartpole*, *Catcher*, and *PuddleWorld*. As we show below, even in these domains updating real state values towards simulated state values can cause failure.

## 5.1 Methodology

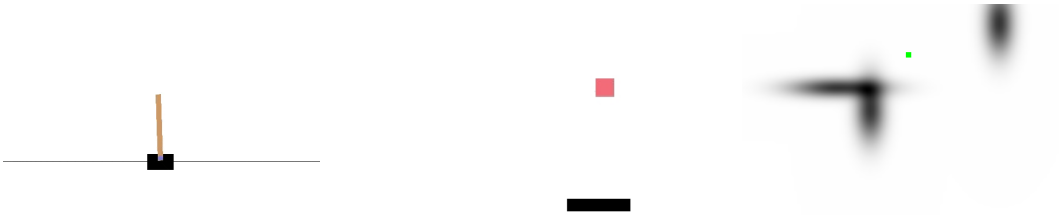
The experiments on *Bordered Gridworld* were contrived for two reasons: first, the environment model was purposefully inaccurate, and second, the value function explicitly contained capacity to represent fictitious states. In experiments in this chapter, we trained a neural network to predict environment dynamics and used a linear function to approximate the $Q$-value function. Details of both these design choices are described in the following sections. First though, we describe the three environments.

### 5.1.1 Environments

The goal in ***Cartpole*** is to balance a pole hinged to a movable cart as shown in Figure 5.1 (a). The agent observes $(x_c, \dot{x_c}, \theta, \dot{\theta})$, the current position of the cart on the $x$-axis, the cart's velocity, the angle of the pole with relative to the vertical axis, and the rate of change of the angle, respectively. Note, all elements of this tuple are *continuous*, thus making the state space a continuous space. Observing these values, the agent may choose to move the cart to the left or to the right. For every time step that the pole remains within 15° of the vertical axis, the agent receives a reward of +1. The episode terminates when the pole's angle exceeds 15° of the vertical axis. We use the implementation provided in OpenAI Gym [2].

In ***Catcher*** an agent tries to catch as many 'fruits' as possible (Figure 5.1 (b)). The agent observes $(x_p, \dot{x_p}, x_f, y_f)$, the position of the paddle (which 'catches' the fruits), the paddle velocity, the fruit position on the $x$-axis, and the fruit position on the $y$-axis, respectively. The agent may choose to move the paddle left, do nothing, or to move the paddle right. The agent is initialised with 3 lives in each episode; every missed fruit reduces 1 life and returns a reward of $-1$. Losing all 3 lives results in episode termination. On the other hand, catching fruit results in a reward of +1. We use the implementation provided by the PyGame Learning Environment [29].

***PuddleWorld*** is a continuous state space version of classic RL Gridworlds. The agent is situated in a state defined by the tuple $(x, y)$ giving its position in the world. It may take actions to move up, down, left, or right. The agent's position post-action is affected by stochastic noise drawn from a uniform distribution over $[-0.025, 0.025]$. At each time-step, the agent receives a reward of $-1$. Moreover, the state space contains 'puddles' (shown as black regions in Figure 5.1 (c)) with increasing negative reward – the further the agent ventures into a puddle, the greater the negative reward. We use the implementation of *Puddleworld* provided in [9].



(a) *Cartpole* (b) *Catcher* (c) *PuddleWorld*

Figure 5.1: **Benchmark RL Domains**: *Cartpole*, *Catcher*, and *Puddleworld*.

### 5.1.2 Environment Models

We sought to learn environment models using the method described in Section 2.4.1. This method was inspired by Oh *et al*. [18] in that it produces expectations of the reward and of the next or previous state. By following this method, we hope to achieve model accuracy similar to that of methods currently prevailing in the literature and to capture typical modelling errors generated by such methods.

### 5.1.3 Value Function

In our *Bordered Gridworld* experiments, we used a tabular value function which represented the values of impossible, fictitious states in addition to real, reachable states of the environment. To remove this contrivance from these experiments, we use a linear value function with a dense state representation obtained from a pre-trained DQN agent.

The state representation was obtained by training a DQN agent on each of the domains tested. Once this agent had converged, we froze the network weights. To obtain a state representation $s_t$ for feature vector $o_t$ emitted by the environment, we passed $o_t$ through this pre-trained network. The penultimate layer of the network (i.e., prior to the layer containing state-action values) was 200 units wide, and the activation of these 200 units formed the state representation $s_t$ which we fed to our linear function approximator. These representations allowed us to more realistically mimic real-world non-tabular

value functions.

With this mechanism for generating representations, the actual linear function approximator agent was trained completely online using standard $Q$-learning with linear function approximation as described in Section 2.4.1. The weights of the value function were all initialised to 0.1.

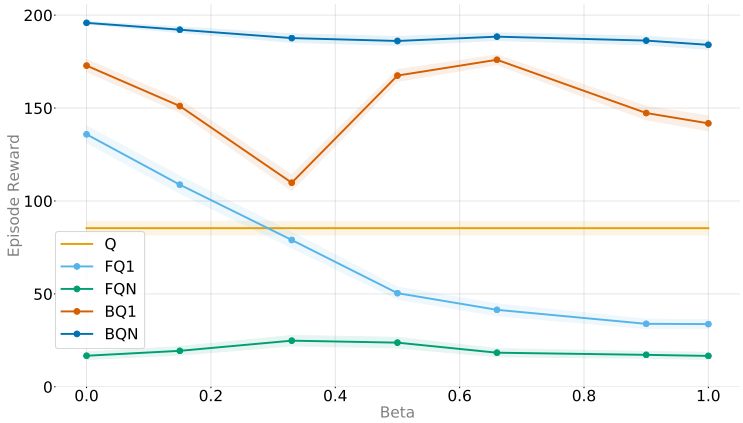
## 5.2 Results

### 5.2.1 Robustness to Model Error

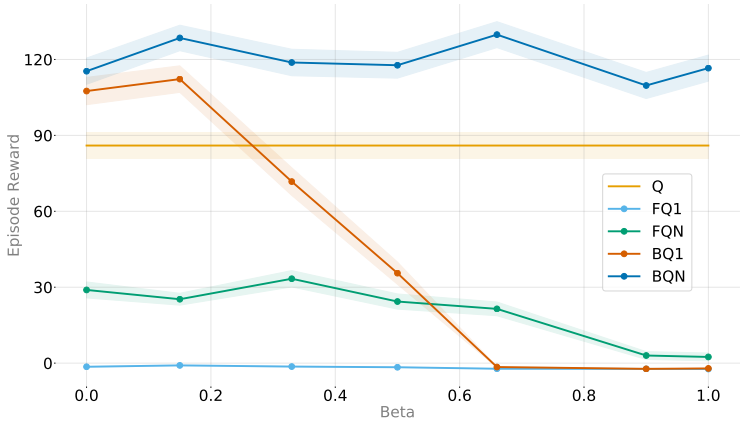
Figure 5.2 shows plots of performance – average episode reward over the final 10 episodes of a 100 episode run all averaged over 30 runs – versus a range of values of $\beta$ on the three benchmarks. The performance shown is for the best value of $\alpha$ for each setting of $\beta$ following an exhaustive sweep. The value of $\beta$ acts as a proxy for model error – greater values imply greater compounding error due to model 'iteration' (see [12], [27], [28] for discussion) while $\beta = 0$ prohibits model 'iteration' thereby eliminating compounding error (though not model error as the even the first prediction made by the model may contain error). An algorithm robust to model error, therefore, should not exhibit significant performance degradation as $\beta$ is increased.

We hypothesised that planning updates in Dyna which move real state values toward simulated state values may cause failure to learn control policies. In terms of specific algorithms, we expect 1-step successor Dyna (abbreviated to $FQ1$) and $n$-step successor Dyna (abbreviated to $FQN$) to find it difficult to learn good control policies as they directly update real state values to simulated state values. We also expect 1-step predecessor Dyna (abbreviated to $BQ1$) with large values of $\beta$ to struggle as well as it increasingly updates real state values to simulated state values as $\beta$ is increased (see Section 3.3.4 for discussion). On the other hand, $n$-step predecessor Dyna (abbreviated to $BQN$) ought *not* to exhibit significant performance degradation no matter the setting of $\beta$ as it is designed not to update real state values to simulated state values in planning.

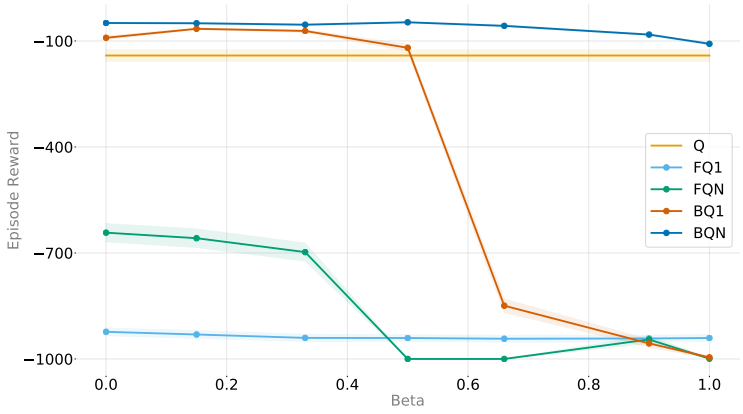
The benchmark results largely support our hypothesis and speculations




(a) *Cartpole*




(b) *Catcher*




(c) *PuddleWorld*

Figure 5.2: **Performance on *Cartpole*, *Catcher*, and *Puddleworld*.** Average episode performance against a range of values of $\beta$. *BQN* proves to be robust to model error induced by high values of $\beta$ while other algorithms seem to fail as $\beta$ is increased.

about specific algorithms. In *Catcher* and *Puddleworld* $FQ1$ and $FQN$ failed to reach the performance of $Q$-learning for any value of $\beta$ indicating that successor Dyna updates toward erroneous simulated states impeded agent's ability learn effective control policies. Similarly, $FQN$ failed in *Cartpole* as well. However, $FQ1$ performed surprisingly well for $\beta \in \{0, 0.15, 0.33\}$ before eventually deteriorating.

On *Catcher* and *Puddleworld* $BQ1$ performed as expected: it performs well for lower values of $\beta$ but degrades as $\beta$ is increased until it fails to learn effective control policies. On *Cartpole*, though, it performs surprisingly well across all values of $\beta$.

In all domains, $BQN$ was robust to $\beta$ and therefore model error. Even for $\beta = 1$, a setting with maximal model 'iteration' (and therefore highest likelihood of generating fictitious states) and error, performance barely degraded. On *Cartpole* – the game with maximum performance degradation – $BQN$ performance with $\beta = 1$ was only 6% lower than performance with $\beta = 0$. Furthermore, unlike in *Bordered Gridworld*, $BQN$ was the best performing algorithm overall and outperformed $BQ1$ with $\beta = 0$.

It is surprising that the results predicted by the *Hallucinated Value Hypothesis* are so apparent in these experiments. In tabular domains it is easy to see that fictitious states may have arbitrary values. For instance, in *Bordered Gridworld*, as the value function explicitly represents values of border states, the border states may be initialised to arbitrary values, and one can see that propagation of this value may mislead the agent. Here, however, values are generated by a linear function over a dense state representation. As the weights of the linear learner are used to generate values for *all* states $s \in \mathcal{S}$, fictitious states are unlikely to have their own set of reserved weights inducing arbitrary value. This suggests that the values of fictitious states are not entirely arbitrary — they may be partially informed by values of real states. Despite this the agent is still misled by updates towards them.

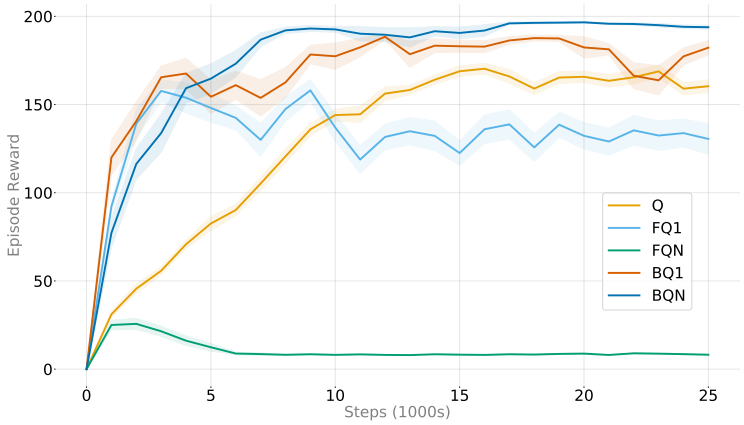
Conversely, now consider the updates performed by algorithms that do not update real state values to simulated state values. Even in this case, as weights are likely common between real states and fictitious states, it is possible that

the updates performed by $BQ1$ with $\beta = 0$ and $BQN$ can mislead the value function. For example, the predecessor models may generate fictitious states $\hat{s}$ which very likely has an overlap in activation of the feature vector with some real state $s$. A value function update to the weights associated with $\hat{s}$ will also lead to a change in the weights that represent $s$. This may change $Q(s, \cdot)$, and could potentially mislead the agent. However, we do not see such behaviour occurring on any domain tested and this is rather surprising.

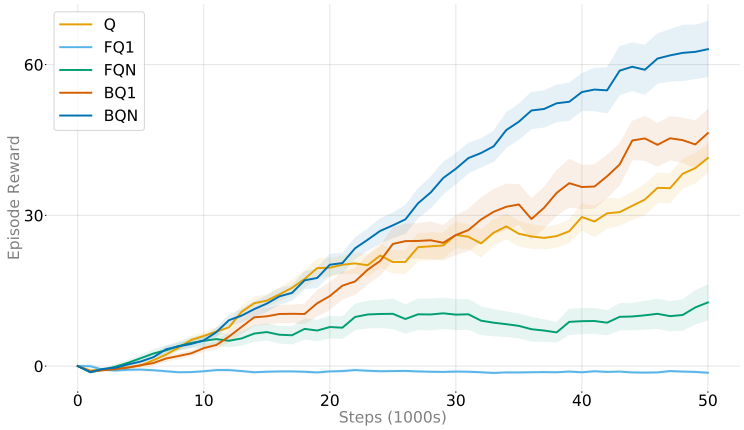
### 5.2.2 Learning Speed

While $BQN$ does indeed outperform $BQ1$ with $\beta = 0$, one may argue that the performance gain does not outweigh the additional computational cost of performing $n$-step TD updates. Why should one use $BQN$ then? Figure 5.3 plots performance as a function of cumulative interactions with the environment. Specifically, the plots show performance averaged over the 10 most recently completed episodes every 1000 cumulative environment interactions. For each algorithm – $Q$-learning, $FQ1$, $FQN$, $BQ1$, and $BQN$ – we select the best settings of $\alpha$ and $\beta$. These plots are averages over 30 runs.

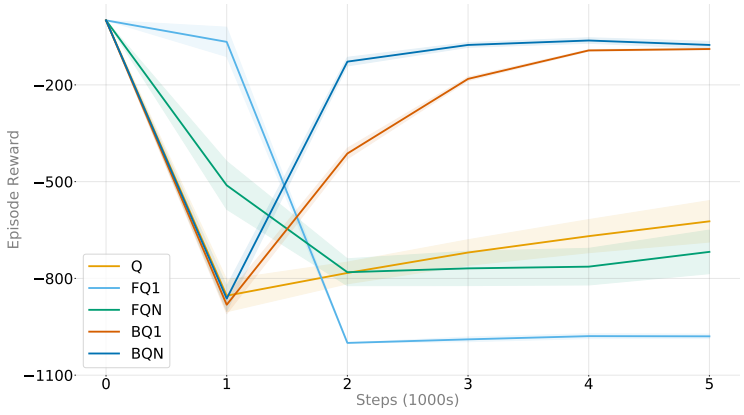
In all domains, $BQN$ was the fastest learning algorithm. In *Puddleworld*, $BQN$ reaches close to optimal performance in about 2000 interactions with the environment while the next best, $BQ1$, requires double that at 4000 steps. Similarly, in *Cartpole* $BQN$ reaches optimal performance in about 7500 interactions while $BQ1$ performance has not stabilised by 25000 interactions. In *Catcher*, $BQ1$ needs 50000 steps to achieve the performance that $BQN$ reaches in approximately 32000 steps. Succinctly, the greatest sample efficiency benefits are reaped by $BQN$. As the algorithm is not catastrophically affected by high values of $\beta$, it reaps the benefits of diverse model experience accrued by such settings of $\beta$ (see [8] for discussion on the benefits of diversity brought by long rollouts).




(a) *Cartpole*




(b) *Catcher*




(c) *PuddleWorld*

Figure 5.3: **Learning curves on *Cartpole*, *Catcher*, and *Catcher***. Curves are shown for the best setting of $\alpha$ and $\beta$ of each algorithm.

# Chapter 6

# Conclusions & Future Work

This thesis set out to address the questions,

*What are some consequences of using an imperfect model in Dyna?*

In this final chapter, we summarise our findings towards answering this question. We also describe new directions for future research that arise from these findings.

## 6.1 Contributions

Our two primary contributions are the following.

***Hallucinated Value Hypothesis***. We speculated that planning temporal difference (TD) updates in Dyna algorithms that updated real state values towards simulated state values may harm an agent's control policy. We suggested that this may occur because an imperfect model might generate fictitious states that do not correspond to real environment states, and that the values for these states may be arbitrary. The propagation of this arbitrary value through the value function could potentially mislead the control policy to leave agent's *chasing hallucinated value*. Given this reasoning, we proposed the *Hallucinated Value Hypothesis*. We showed the hypothesis held not only in the designed domain *Bordered Gridworld*, but also in reinforcement learning (RL) benchmarks *Cartpole*, *Catcher*, and *Puddleworld*.

**$n$-step Predecessor Dyna**. We proposed $n$-step predecessor Dyna, an algorithm that is designed not to perform updates from real states to simulated states in planning. We showed it was robust to model error in all experiments, and that it had attractive properties in terms of learning speed.

Additionally, there is one secondary contribution of this thesis.

**$\beta$-Prioritised Dyna**. We introduced an extension to standard Dyna style algorithms which incorporated $\beta$, a simple mechanism to control imperfect model 'iteration' thereby preventing updates with inaccurate simulated states.

Concisely, for the question of what problems might be caused by using an imperfect model in Dyna, we identified and explored the *Hallucinated Value Hypothesis*. As for what may be done to solve the identified issues, we proposed $n$-step Predecessor Dyna.

## 6.2 Directions for Future Work

We have identified the following directions as being fruitful for future work.

**Improved methods for halting model 'iteration'**. We proposed $\beta$-Prioritised Dyna as a simple method to ameliorate the problem of model 'iteration' causing compounding error problems. Clearly, this is a crude solution and many better approaches can be readily suggested. For instance, one may query the model for a confidence estimate of its own predictions and use this to decide whether the 'rollout' ought to continue. In parts of the state space where the model is confident, it may be advantageous to have longer 'rollouts' thus obtaining the benefits suggested by Holland *et al*. [8] rather than simply cutting short all 'rollouts'.

**Learning localised environment models**. In Section 3.1.1, we described one way in which a global environment model may generate erroneous simulated transitions in *Bordered Gridworld*. To reiterate, a model may

generalise the behaviour on states in the 'interior' of the reachable states (where actions results in regular, consistent movement of the black square) to the borders and thus produce erroneous simulated transitions into the border. If many local models were to be learned, however, it may be possible to learn separate local models for the interior of the reachable states and for the borders. The local models at the border may learn that it is not possible to transition into border states as they do not need to generalise across the entire state space.

## 6.3 Dénouement

Ultimately, this thesis is a small step toward building effective model-based reinforcement learning algorithms for large, complex-dynamics real-world problems where perfect environment models cannot be obtained. It complements and builds upon work done by many others. Holland *et al.* [8] showed that longer rollouts resulted in the greatest benefit to model-based updates. Kaiser *et al.* [10] built on the work of Oh *et al.* [18] to develop an environment model for Arcade Learning Environment [1] games with impressive results. On the other hand, van Hasselt *et al.* [7] suggested that, in some circumstances, it may be better to use an experience replay buffer as a type of model rather than explicitly developing a parametric model of environment dynamics. The afore-mentioned works are but a small sample of the various directions that many researchers have pursued to make Dyna in large domains a reality. This work complements many of these and may be used in conjunction with them.

# References


[1] 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*, 2013. 3, 12, 59

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, *OpenAI gym*, 2016. eprint: `arXiv:1606.01540`. 50

[3] J. Buckman, D. Hafner, G. Tucker, E. Brevdo and H. Lee, 'Sample-efficient Reinforcement Learning with Stochastic Ensemble Value Expansion', in *Advances in Neural Information Processing Systems*, 2018. 22, 25

[4] X. Glorot and Y. Bengio, 'Understanding the Difficulty of Training Deep Feedforward Neural Networks', in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010. 12, 16

[5] A. Goyal, P. Brakel, W. Fedus, S. Singhal, T. Lillicrap, S. Levine, H. Larochelle and Y. Bengio, 'Recall Traces: Backtracking Models for Efficient Reinforcement Learning', *arXiv preprint arXiv:1804.00379*, 2018. 22

[6] S. Gu, T. Lillicrap, I. Sutskever and S. Levine, 'Continuous Deep $Q$-learning with Model-based Acceleration', in *International Conference on Machine Learning*, 2016. 22, 24

[7] H. van Hasselt, M. Hessel and J. Aslanides, 'When to Use Parametric Models in Reinforcement Learning?', *arXiv preprint arXiv:1906.05243*, 2019. 59

[8] G. Z. Holland, E. Talvitie and M. Bowling, 'The Effect of Planning Shape on Dyna style Planning in High-dimensional State Spaces', *arXiv preprint arXiv:1806.01825*, 2018. 22, 24, 35, 55, 58, 59

[9] E. Imani, *Puddleworld*, `https://github.com/EhsanEI/gym-puddle`, 2018. 50

[10] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine *et al.*, 'Model-based Reinforcement Learning for Atari', *arXiv preprint arXiv:1903.00374*, 2019. 59

[11] G. Kalweit and J. Boedecker, 'Uncertainty-driven Imagination for Continuous Deep Reinforcement Learning', in *Proceedings of the First Conference on Robot Learning*, 2017. 22, 24

[12] N. R. Ke, A. Singh, A. Touati, A. Goyal, Y. Bengio, D. Parikh and D. Batra, 'Learning Dynamics Model in Reinforcement Learning by Incorporating the Long Term Future', in *Proceedings of the Eight International Conference on Learning Representations*, 2019. 52

[13] D. P. Kingma and J. Ba, 'Adam: A Method for Stochastic Optimization', *arXiv preprint arXiv:1412.6980*, 2014. 12, 16

[14] S. Legg and M. Hutter, 'Universal Intelligence: A Definition of Machine Intelligence', *Minds and machines*, 2007. 1

[15] L.-J. Lin, 'Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching', *Machine learning*, 1992. 13

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, 'Human-level Control Through Deep Reinforcement Learning', *Nature*, 2015. 3, 11, 13

[17] A. W. Moore and C. G. Atkeson, 'Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time', *Machine Learning*, 1993. 22, 30

[18] J. Oh, X. Guo, H. Lee, R. L. Lewis and S. Singh, 'Action-conditional video prediction using deep networks in atari games', in *Advances in neural information processing systems*, 2015, pp. 2863–2871. 16, 51, 59

[19] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng and W. Zaremba, 'Learning dexterous in-hand manipulation', *CoRR*, 2018. [Online]. Available: `http://arxiv.org/abs/1808.00177`. 3

[20] Y. Pan, M. Zaheer, A. White, A. Patterson and M. White, 'Organizing Experience: A Deeper Look at Replay Mechanisms for Sample-based Planning in Continuous State Domains', in *Proceedings of the Twenty Seventh Internation Joint Conference on Artificial Intelligence*, 2018. 22

[21] J. Peng and R. J. Williams, 'Efficient Learning and Planning Within the Dyna Framework', *Adaptive Behavior*, 1993. 22, 30

[22] R. S. Sutton, 'Learning to Predict by the Methods of Temporal Differences', *Machine learning*, 1988. 9

[23] ——, 'Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming', in *Machine Learning Proceedings 1990*, 1990. 2, 22, 24, 25, 30

[24] ——, 'Dyna, an Integrated Architecture for Learning, Planning, and Reacting', *ACM SIGART Bulletin*, 1991. 14

[25] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. 2018. 9

[26] R. S. Sutton, C. Szepesvári, A. Geramifard and M. Bowling, 'Dyna-style Planning with Linear Function Approximation and Prioritized Sweeping', in *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, 2008. 22

[27] E. Talvitie, 'Model Regularization for Stable Sample Rollouts', in *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, 2014. 23, 52

[28] ——, 'Self-correcting Models for Model-based Reinforcement Learning', in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 23, 52

[29] N. Tasfi, *Pygame Learning Environment*, `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016. 50

[30] A. Venkatraman, M. Hebert and J. A. Bagnell, 'Improving Multi-step Prediction of Learned Time Series Models', in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. 23

[31] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis and D. Silver, *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*, `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019. 3

[32] C. J. Watkins and P. Dayan, '$Q$-learning', *Machine Learning*, 1992. 10

[33] H. Yao, S. Bhatnagar, D. Diao, R. S. Sutton and C. Szepesvári, 'Multi-step Dyna Planning for Policy Evaluation and Control', in *Advances in Neural Information Processing Systems*, 2009. 22, 25