The Effect of Planning Shape on Dyna-style Planning in High-dimensional State Spaces

by

Gordon Z. Holland

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Dyna is an architecture for reinforcement learning agents that interleaves planning, acting, and learning in an online setting. This architecture aims to make fuller use of limited experience to achieve better performance with fewer environmental interactions. Dyna has been well studied in problems with a tabular representation of states, and has also been extended to some settings with larger state spaces that require function approximation. In Dyna, the environment model is typically used to generate one-step rollouts from selected start states, but longer trajectories could also be generated. Given a fixed budget of computation, planning could take on a variety of shapes: many short rollouts, or fewer long rollouts. In this work, one-step Dyna was applied to several games from the Arcade Learning Environment (ALE) and the result was that the model-based updates offered surprisingly little benefit over performing more updates with the agent's existing experience, even when using a perfect model. However, when the model was used to generate longer trajectories of simulated experience, performance improved dramatically. The results show that to get the most from planning, the model must be used to generate unfamiliar experience, and that performing longer rollouts is an effective strategy to accomplish this. Similar observations were made with pre-trained learned models and a model that was learned online along with the value function.

Preface

Some of the work presented in this thesis — namely, the perfect model experiments described in Chapter 3, and the Rollout-Dyna-DQN experiments with the pre-trained learned model in Section 4.1.1 — was presented at the ICML 2018 workshop on Prediction and Generative Modeling in Reinforcement Learning. At the time of writing, a version of that work is under review for AAAI 2019, with the addition of the online learned model results in Section 4.2. This thesis is the definitive and expanded treatment of these topics. Additionally, the Sarsa experiments with the learned model in Section 4.1.2, and the Monte-Carlo experiments in Appendix B are new to this thesis.

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

– Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid.

Acknowledgements

I would like to thank:

- my supervisors, Professors Michael Bowling and Erik Talvite, for their support and guidance,
- Professor Richard Sutton, for introducing me to reinforcement learning and sparking my passion for the field,
- Marlos Machado, for his valuable advice and general ALE wisdom,
- fellow grad students from the RLAI group for their helpful discussions and suggestions,
- my partner, Kendall, for her endless love and support,
- the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta Innovates for the scholarships that helped in part to fund this thesis,
- WestGrid (westgrid.ca), CalculQuébec (calculquebec.ca), and Compute Canada (computecanada.ca) for providing some of the computational resources used for running the experiments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement learning (RL) is a computational approach to learning from interaction. In RL problems, an agent typically interacts with the environment by selecting actions over a series of discrete time steps to maximize a sum of scalar reward signals. Often, a *policy*, which is a way of behaving, can be learned to solve these problems without learning a predictive model of the environment; approaches that do this are called *model-free.* In contrast, *model-based* methods make use of an explicit environment model — a model that can predict future states and rewards, conditioned on actions — to perform planning. In this sense planning is any process that takes a model and uses it to produce or improve a policy. Although they are generally more complex than model-free methods, model-based methods have been shown to learn a good policy with fewer environmental interactions, making better use of limited experience (e.g., Sutton, 1990; Moore and Atkeson, 1993; Peng and Williams, 1993). This is especially important for domains where interacting with the environment and collecting real experience is expensive.

There are many ways that an environment model could be used for planning. One particularly appealing way is to use the model to generate simulated experience alongside the real experience observed from the environment. To plan, a start state is selected, either from the agent's prior experience or generated by some other process. Simulated experience is generated from that state and treated as though it were real experience in the environment. The real and simulated experience could then both be used in the same way: to update the agent's value function and/or policy. This is the paradigm underlying the Dyna architecture, and planning that proceeds in this way is referred to as *Dyna-style planning*. Another core idea of Dyna is that conceptually, planning, acting, and learning all occur simultaneously, and as fast as possible. However, for practical reasons, the computation available at each time step is shared between these three processes in proportions that can be set according to the resources available and the required response time of the system. Much of the work on Dyna has been based on tabular problems or relatively low-dimensional continuous problems. This thesis seeks to study it in a more complex, higher-dimensional setting: the Arcade Learning Environment.

High-dimensional domains, like those based on images, provide challenging environments for testing RL algorithms. The Arcade Learning Environment (ALE; Bellemare, Naddaf, Veness, and Bowling, 2013) is an important testbed in this category, where agents learn to play games from the Atari 2600 system with raw images as input. For agents to succeed in the ALE, they need to have general competency to learn to act in complex environments with varied dynamics across many games. The goal of general competence is a key component for creating more powerful artificial intelligence systems.

Many model-free approaches exist for playing games from the ALE (e.g., Mnih et al., 2015; Mnih et al., 2016; Hessel et al., 2018), but there are surprisingly few model-based approaches (e.g., Oh, Singh, and Lee, 2017). Ostensibly, this is because learning a model of an Atari game and planning with it is difficult, which makes the ALE a good testbed for model based approaches. Because of its high-dimensional observation space, the most successful approaches in the ALE have used non-linear function approximation. In this setting, unlike in the tabular or linear function approximation setting, start states for planning cannot be easily generated. However, they can be sampled from the agent's real experience, and planning could take on a variety of shapes to allow learning to occur over different distributions of states. From a start state selected from the agent's history, experience can be generated by rolling out a single step to the next state — which is the most common approach — or longer trajectories can be generated. This concept can be referred to as *planning shape*. This raises the question: with Dyna-style planning, given a fixed budget of computation, what is the most effective planning shape? Is it more effective to generate many short rollouts, or fewer long rollouts?

To investigate this question, Dyna's performance was evaluated on several games from the ALE using a variety of planning shapes. First, experiments using a perfect model were conducted to explore the idealized performance of the algorithm. Surprisingly, in these experiments, Dyna-style planning with many one-step rollouts provides almost no benefit over simply doing more updates with the real data already collected by the algorithm — a trivial form of a model. It is only when the model is used to produce multi-step rollouts, sequences of more than one decision, does the additional computation required for planning become beneficial.

The experiments were then repeated using a learned model by extending Oh, Guo, Lee, Lewis, and Singh's (2015a) action-conditional video prediction architecture to predict future rewards in addition to the future frames, and pre-training it on expert data. Even when the model is imperfect, planning with rollouts of length greater than one tend to provide the most benefit. The empirical results demonstrate that rollout length is a key factor in the effectiveness of Dyna-style planning.

Finally, this thesis explores learning the model online alongside the value function, which is Dyna in its most complete form. For some games, learning and planning in this manner improves the performance over simply doing more updates with the real data, which is the first time this has been demonstrated in the ALE.

Despite the introduction of increasingly effective approaches for learning predictive models in Atari Games (Bellemare, Veness, & Bowling, 2013; Bellemare, Veness, & Talvitie, 2014; Oh et al., 2015a), the application of model-based methods to the ALE is still an open problem, with many challenges to overcome. This thesis empirically studies one piece of the puzzle: the consideration of planning shape's impact on Dyna-style planning in problems with high-dimensional state spaces. For the first time (as far as the author is aware), a sample-complexity benefit was demonstrated from learning a dynamics model in some games. Additionally, the results give guidance for further progress. The perfect model results show that even if there were dramatic improvements in model-learning, where highly accurate models could be learned very quickly, there would still be little benefit when using one-step rollouts from previously visited states. Longer rollouts — as one means of generating a more diverse distribution of states to learn from — shows promise to allow planning to exploit more accurate models.

Chapter 2

Background

This chapter contains some relevant background and notation. By convention scalar random variables are denoted by capital letters (e.g. S_t, A_t), sets by calligraphic font (e.g. $\mathcal{S}$, $\mathcal{A}$), matrices by bold upper case letters (e.g. $\mathbf{W}$), vectors by bold lower case letters (e.g. $\mathbf{x}$, $\boldsymbol{\theta}$), and functions by non-bold lower case letters (e.g. $q(s, a)$).

2.1 Reinforcement Learning

Reinforcement learning (RL) problems are characterized by sequences of decisions typically modelled as a *Markov Decision Process* (MDP) (Sutton & Barto, 1998, 2018). In this framework, an agent interacts with an environment over a series of discrete time steps. Formally, an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r \rangle$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, p is the *state-transition probability*, and r is the *reward function*. At each time step, t, the agent receives information from the environment about its current state, $S_t \in \mathcal{S}$, and uses this information to select an action $A_t \in \mathcal{A}$. Then the agent transitions to a new state $S_{t+1} \in \mathcal{S}$ according to the state-transition probability, $p(s'|s, a) \doteq Pr\{S_{t+1} = s' | S_t = s, A_t = a\}$, and receives a reward $R_{t+1} \in \mathbb{R}$, according to the reward function $r(s, a, s') \doteq \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s']$.

Typically, the goal of a reinforcement learning agent is to maximize the expected discounted return: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$, where $\gamma \in [0, 1]$ is a discount factor. This is accomplished by learning a policy, $\pi(a|s)$, that is the probability of selecting action a, when in state s.

There are a class of solution methods called value-based methods that learn a value function that estimates the expected return when selecting an action a, while in state s and following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right]. \tag{2.1}$$

The optimal policy, π^*, is the policy that maximizes (2.1). Q-learning (Watkins, 1989; Watkins & Dayan, 1992) is a particular method that estimates the optimal policy by observing new rewards, and using old estimates of action-values to estimate new action-values according to the update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{2.2}$$

where $\alpha \in (0, 1]$ is a step size parameter. Q-learning is considered a *bootstrapping* method because its update is based on an old estimate of the action-value function.

Tabular solution methods can be used when the state and action spaces are small enough so that it is possible to maintain the estimates of the value functions in an array or table. When the state space is large and/or continuous, *approximate solution methods* need to be used, which combine RL algorithms with some kind of function approximation scheme.

With approximate solution methods, an approximate parameterized form of the action-value function is learned. In particular, the action-value function is parametrized by a weight vector $\boldsymbol{\theta} \in \mathbb{R}^n$, and the approximate action-value function can be written as $\hat{q}(s, a, \boldsymbol{\theta}) \approx q_\pi(s, a)$ (Sutton & Barto, 2018).

One important class of learning methods for function approximation is stochastic gradient descent (SGD). With SGD methods, the weight vector is a column vector with a fixed number of real valued components, and the approximate action-value function, $\hat{q}(s, a, \boldsymbol{\theta})$, is a differentiable function of $\boldsymbol{\theta}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Since $\boldsymbol{\theta}$ will be updated over a series of time steps, it is given a subscript denoting the weight vector at each step — e.g. $\boldsymbol{\theta}_t$.

SGD methods adjust the weight vector for each observed example — or batch of examples — in the direction that would most reduce the error. Assuming the objective function being minimized is the *mean squared value error*

(MSVE), the approximate update to the parameters is given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{1}{2}\alpha\nabla\left[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\right]^2 \tag{2.3}$$

$$= \boldsymbol{\theta}_t + \alpha\left[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\right]\nabla\hat{q}(S_t, A_t, \boldsymbol{\theta}_t), \tag{2.4}$$

where the update target U_t is some estimate of $q_\pi(S_t, A_t)$. For Q-learning, $U_t = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \boldsymbol{\theta}_t)$.

Bootstrapping estimates of $q_\pi(S_t, A_t)$, such as the target for Q-learning, depend on the current value of the weight vector $\boldsymbol{\theta}_t$. Thus, if they are used for the update target U_t, then the step from (2.3) to (2.4) is not possible as written because it relies on the target being independent of $\boldsymbol{\theta}_t$. As a result, bootstrapping methods are not instances of true gradient descent, and because they make use of only part of the gradient, they are called *semi-gradient methods*. Q-learning as a semi-gradient method can be written as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\left[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \boldsymbol{\theta}_t) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\right]\nabla_{\boldsymbol{\theta}_t}\hat{q}(S_t, A_t, \boldsymbol{\theta}_t). \tag{2.5}$$

Methods like Q-learning are examples of model-free RL since they do not learn or make use of an explicit model of the environment's dynamics, but there are also RL methods that make use of a model, which are discussed in the next section.

2.2 Model-based RL

Any approach that uses a model for planning can be considered a model-based RL method. Planning in this sense is the process of using a model of the environment to improve the value function and/or policy. An environment model is anything that can be used by an agent to predict the consequences of its actions. Given a starting state and action, a model often predicts the next state and reward, and can be used to simulate experience.

If the environment is stochastic, then there can be multiple possible next states and rewards. If the model of a stochastic environment produces the average of all possible outcomes, weighted by their respective probabilities, then it is an *expectation model*. A limitation of expectation models is that the expected next state and reward might not be realizable in the real environment. If the model produces a description of all the possibilities and their associated probabilities, then it is considered a *distribution model* (Sutton & Barto, 2018). If the model produces a single outcome according to the probabilities, then it is considered a *sample model*. Distribution models are more powerful than expectation and sample models, since they can be used to produce the expectation or samples, but expectation and sample models are often simpler and easier to describe or learn.

The simplest example of a sample model is experience replay (Lin, 1992). Experience replay works by storing transitions — state, action, next state, and reward — in a buffer that the agent can later use for planning. In fact, there is a deep connection between a more general class of models and experience replay; van Seijen and Sutton (2015) showed an exact equivalence between the sequence of value functions found by a model-based method with a multi-step linear model, and a model-free method with replay. However, experience replay as a model cannot generalize to new situations that have not been experienced by the agent and thus are not contained in the replay buffer (although the value function could provide some generalization). More powerful models might be able to generalize to novel states that have never been experienced by the agent. The results in this thesis suggest that a model that can generate novel experience can offer dramatic benefits over experience replay.

Learning a model to use for planning presents several challenges. The model and policy are closely connected, and if the model is trained using data generated under a different policy than is later used to query the model — called the *train-test mismatch* — poor performance can result. For example, imagine the model is trained using some state-action pair distribution, $\mathcal{X}$, that is the result of following some policy π_1. Then, the model is used for planning and creates a new policy π_2 that induces a new state-action pair distribution $\mathcal{Y}$. However, there may be state-action pairs in $\mathcal{Y}$ that are not in $\mathcal{X}$. Therefore, when the model is asked to make predictions about the states in $\mathcal{Y}$ that it has not been trained on, poor performance can result. Ross and

Bagnell (2012) showed that to achieve good model performance, continuously mixing states generated by the current policy into the model training data is required. Learning a model online, using the agent's current observations, is one way that this could be accomplished. Another related issue is the problems that arise when the model is used to roll out multiple steps. Since the model is learned, it is necessarily imperfect, and will make small errors in each prediction. When rolling out, the model's prediction at the current step is used to predict the next step. This can result in compounding errors which increase at each step in the rollout. Talvitie (2014, 2017) showed that to stabilize rollouts, the model needs to be trained on its own samples so that sensible predictions are made when its own samples are given as input. Oh et al. (2015a) addressed this issue by training the model on multi-step prediction error via backpropagation through time. This thesis employs their approach in Chapter 4.

There are many ways that a model can be used for planning. For example, Value Iteration Networks (Tamar, Wu, Thomas, Levine, & Abbeel, 2016) takes a dynamic programming approach and uses a neural network with an embedded planning module that approximates the value-iteration algorithm (Bellman, 1957) to improve the policy. Another example of using a model is online model-predictive control, which uses the model in a search process at every step in order to select a single action like in DAgger-MC (Talvitie, 2015). Another similar example is Imagination-Augmented Agents (Weber et al., 2017) where the model predictions are used as additional input to the policy, which is learned in a model-free way, to select actions. The model can also be used to generate simulated experience alongside real experience collected from the environment. The real and simulated experience can then be used in the same way to update the value function, which is the basis of the Dyna architecture described in the next section.

2.3 Dyna

Dyna is a general architecture for model-based RL, which is the focus of this work. The core idea is to integrate planning, learning, and acting. Conceptually, planning, learning, and acting in the environment happen simultaneously, and as fast as possible. While collecting real experience from the world, Dyna plans by using a model to also generate simulated experience. The real and simulated experience are treated in the same way to update the agent's value function and/or policy. Any planning that is done in this way is called *Dyna-style planning*. Dyna allows for flexible control over the amount of planning done by the agent: for every step taken in the environment, there can be many planning steps to generate additional simulated experience. This can be important in environments where acting is computationally or temporally expensive, and where simulating experience may be much faster than collecting it.

An important part of Dyna is that the real experience is not only used to learn the value function online, but it is also used to learn the model. In this work, the online model learning aspect of Dyna is omitted initially to better isolate the effects of planning shape. Nevertheless, Dyna with a online learned model is investigated at the end of Chapter 4.

One of the first instantiations of Dyna was Dyna-Q (Sutton, 1990). Dyna-Q combines one-step tabular Q-learning, with a model that, from a given start state, can predict the next state and reward. After taking a step in the environment the value function is updated with the conventional model-free Q-learning update, which is followed by a model learning update. Then, the following planning procedure is repeated n times: sample a start state, S, uniformly at random from the set of previously seen states; sample an action, A, uniformly from the the set of actions previously chosen in S; use the model to predict the next state, S' and reward, R; and update the value function using S, A, S', and R with the same Q-learning update.

Search control refers to the process of selecting start states and actions for planning, as well as how computation is distributed and executed during planning. The idea of selecting a planning shape can be thought of as a form of search control. In Dyna-Q, start states are selected uniformly at random from all the states experienced so far, but this uniform strategy might not be the most efficient. Instead, it may be possible to focus planning on particular states. This idea is the basis for *prioritized sweeping* (Moore & Atkeson, 1993; Peng & Williams, 1993), which maintains start states for planning in a priority queue according to some indicator of importance, like how much the value of the state changed when its value was last updated. During planning, it then selects the state with the highest priority from the queue, and works backwards by simulating the predecessor states and updating their values.

Dyna has been extended to use linear function-approximation, combined with a linear expectation model of the environment, so that it could be applied to problems that do not admit an easy tabular state representation (Sutton, Szepesvari, Geramifard, & Bowling, 2008). The function approximator maps states to feature vectors which are then used for planning and learning the value function. Instead of selecting start states from previously visited states, feature vectors can be generated according to a probability distribution. Linear Dyna was later extended to incorporate multi-step projections of sampled features, which was found to speed learning (Yao, Bhatnagar, Diao, Sutton, & Szepesvári, 2009).

There has been little previous work applying Dyna to high-dimensional state spaces like images, where non-linear function approximation may be important for success. Faulkner and Precup (2010) combined Dyna with deep belief networks as the function approximator and demonstrated its performance on toy image domains. Gu, Lillicrap, Sutskever, and Levine (2016) and Kalweit and Boedecker (2017) combined Dyna with neural networks to solve simulated robot control problems. During planning, they both used the model to roll out multiple steps, but did not study the impact of rollout length extensively. This thesis continues to explore Dyna-style planning and its application to complex problems with high-dimensional state spaces, as embodied in the ALE benchmark.

2.4 The Arcade Learning Environment

The Arcade Learning Environment (ALE) is a platform for evaluating general competency in artificial intelligence (AI), and is the problem domain used in this thesis. The ALE provides an interface for agents to play a suite of games for the Atari 2600 system. Atari 2600 games present an interesting and challenging set of problems for AI agents for several reasons: they are varied enough to present an assortment of different tasks, thus requiring general competence; they are interesting and challenging for humans; and they are free from experimenter bias, since they were not originally designed for RL experiments (Bellemare, Naddaf, et al., 2013; Machado et al., 2017).

Agents designed for the ALE generally perceive the game environment through the video stream of game frames without using game specific information. Each frame produced by the emulator is an 210×160 pixel image and occurs on the time scale τ. The reward at time τ is the difference between the game score at time τ and time $\tau - 1$. In the ALE, the agent does not usually select an action after receiving every frame. Instead, when the agent selects an action, the action is repeated for a fixed number of times called the *frame skip*. Let the time scale at which the agent selects actions be denoted by t. Thus, a single step experienced by the agent may consists of multiple time steps in the emulator. The reward received by the agent at time t is then the sum of all the intermediate rewards produced by the emulator for the skipped frames between time $t - 1$ and t.

In general, a single frame does not have the Markov property — for example, a single frame does not contain enough information to predict which directions objects on the screen are moving, or sometimes objects on the screen blink in and out of view — and thus is not a state in the MDP sense. However, there are some methods to help deal with the partial observability and provide an approximate state, which are described in Sections 2.5 and 2.6.

Originally, the ALE was entirely deterministic as the Atari 2600 system had no way to generate pseudo-random numbers. Various extensions have been employed to add forms of stochasticity to the ALE (random no-ops, Mnih et al., 2015; human starts, Nair et al., 2015; random frame skips, Brockman et al., 2016), but none have been universally accepted. Recently, Machado et al. (2017) proposed a new approach call *sticky actions*, which is considered the definitive solution to add stochasticity to the ALE and is the approach used in this thesis. The goal of adding stochasticity is to encourage the agent to learn robust policies, and discourage the agent from learning brittle polices by simply memorizing action sequences.

Sticky actions introduces a *stickiness* parameter ς to the environment that is the probability at each time step that the previous action sent to the emulator is repeated, instead of executing the current action. Sticky actions also interact well with frame skipping: for every action sent to the emulator, including actions to be executed on the skipped frames, there is a probability ς that the previous action will be executed. Machado et al. (2017) recommend setting $\varsigma = 0.25$.

The following sections describe two effective algorithms for agents in the ALE that are used in this thesis.

2.5 Deep Q-networks

Deep Q-networks (DQN) (Mnih et al., 2015) is a model-free deep RL method, based on Q-learning, that uses a deep convolutional neural network to approximate the value function. The network consists of two hidden convolutional layers, followed by a hidden fully connected layer, then finally an output layer, to estimate the action-values. The network parameters, $\boldsymbol{\theta}$, are updated, by computing a gradient with backpropagation, according to the semi-gradient Q-learning update rule from (2.5). For stability reasons, DQN maintains a copy of the parameters, $\boldsymbol{\theta}^-$, to compute the Q-learning update target, $R_{t+1} + \gamma \max_{a \in \mathcal{A}} \hat{q}(S_{t+1}, a, \boldsymbol{\theta}^-)$. The target parameters are copied from the estimator parameters at a longer interval than the estimator parameters are updated. This is referred to as the *target network update frequency*. For a given state, the network outputs the action-value estimate for each possible action simultaneously. This makes computing the maximum, which is required when computing the target, more efficient.

Generally, DQN uses an ϵ-greedy behaviour policy to select actions. With probability ϵ, the policy selects a random action, and with probability $(1-\epsilon)$, it selects the action with the highest value. During learning, ϵ is annealed from a high starting value, encouraging more exploration, to a low, but still non-zero, final value.

Unlike Q-learning, DQN does not update the value function after every step using a single transition; instead, DQN uses experience replay (Lin, 1992), and places each observed transition into an experience replay buffer. DQN uses the replay buffer to temporally decorrelate the samples for the sake of neural network training, and not necessarily to perform more updates than it has interactions with the environment. Then, for a single training step, the algorithm selects a mini-batch of transitions from the experience buffer uniformly at random to update the parameters. Training steps are performed after every f observed transitions, which is referred to as the *training frequency*.

When DQN is used with the ALE, the the emulator frames are converted to grayscale and scaled to be 84×84 pixels. Then, the pixelwise maximum of the current and previous emulator frames (i.e. frames $\tau - 1$ and τ) is taken to produce the preprocessed frame at time τ: $\mathbf{x}_\tau$. When the frame skip is greater than one, the agent receives frames and makes decisions at a different timescale than the emulator produces frames. For example, with a frame skip of 5, the agent selects an action every 5th emulator frame, and the selected action is sent to the emulator for each intermediate frame between decisions. The agent's approximate state that is used to select an action at time t is a stack of the current preprocessed frame, $\mathbf{x}_t$, and the frames from the previous three agent steps:

$$\mathbf{x}_{t-3:t} \doteq (\mathbf{x}_{t-3}, \mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^{84\times 84\times 4}. \tag{2.6}$$

This is illustrated in Figure 2.1. This frame stack is an approximation of the state and helps account for the partial observability inherent in a single frame.

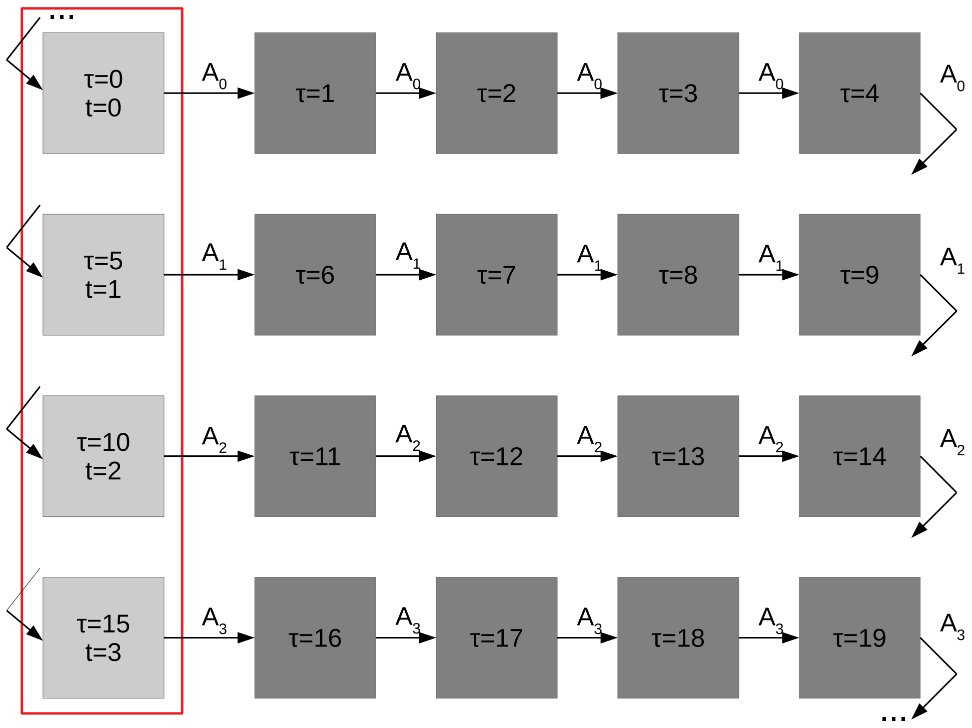


Figure 2.1: How the input to DQN is constructed. This figure shows a sequence of emulator frames (gray squares) and the agent decision points (light gray squares) for a frame skip of 5. At each time t the agent's approximate state is the frames at times $t-3$, $t-2$, $t-1$, and t (red box), which the agent uses to select the action A_t. The action is then repeated 5 times in the emulator, and the intermediate frames are ignored (dark gray squares), before the next action is selected. For simplicity the pixelwise maximum operation, and the emulator's sticky actions are not shown.

2.6 Sarsa with Linear Function Approximation

Sarsa is a RL solution method for control that updates the value function using state and reward samples observed from the environment similar to Q-learning. Unlike Q-learning, Sarsa does not compute the maximum over the possible actions in the update target, and instead uses the action that was actually taken at time $t+1$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]. \qquad (2.7)$$

Therefore Sarsa does not necessarily learn the value function for the optimal policy, but instead learns the value function for the policy being used to select the actions. However, if the policy is greedy, or nearly greedy, with respect to the value function, then Sarsa can learn a policy that is close to the optimal policy. To be able to use Sarsa with the ALE, it needs to be extended to use function approximation.

Linear function approximation is an important special case of function approximation where the approximate action-value function, $\hat{q}(\cdot, \cdot, \boldsymbol{\theta})$, is a linear function of the weight vector $\boldsymbol{\theta}$, and where each state-action pair corresponds to a real valued vector of features $\boldsymbol{\phi}(s, a)$ with the same number of components as $\boldsymbol{\theta}$. The approximate action-value function is then given by the inner product between $\boldsymbol{\theta}$ and $\boldsymbol{\phi}(s, a)$:

$$\hat{q}(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \boldsymbol{\phi}(s, a). \qquad (2.8)$$

With linear function approximation, $\nabla \hat{q}(s, a, \boldsymbol{\theta}) = \boldsymbol{\phi}(s, a)$, and thus the update rule for Sarsa with linear function approximation can be written as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{\theta}_t) - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\right] \boldsymbol{\phi}(S_t, A_t). \qquad (2.9)$$

2.6.1 Blob-PROST Features

To use Sarsa with linear function approximation in the ALE, you need a way to extract features from the games screens to produce $\boldsymbol{\phi}(\cdot, \cdot)$. This work makes use of Blob Pairwise Relative Offsets in Space and Time (Blob-PROST) features (Liang, Machado, Talvitie, & Bowling, 2016), which builds upon many of the ideas presented in earlier work on feature extraction for the ALE (e.g. Bellemare, Naddaf, et al., 2013).

Atari 2600 screens are 160×210 pixels in size with a 128 colour palette. Blob-PROST works by first processing the screen into *blobs*, which are contiguous regions of pixels that are all the same colour. The position of a blob is then the centroid of the smallest bounding box. The screen is then divided into tiles 4×7 pixels in size, and for every tile (c, r) and colour k, where $c \in \{1, ..., 40\}$, $r \in \{1, ..., 30\}$, and $k \in \{1, ..., 128\}$, the feature $\phi^{\text{blob}}_{c,r,k}$ is 1 if the tile (c, r) contains the blob of colour k.

Next, features are generated to capture the relative offsets between the blobs on the screen. If there is a pair of blobs on the screen with colours $k_1, k_2 \in \{1, ..., 128\}$ separated by an offset (i, j), where $-29 \leq i \leq 29$ and $-39 \leq j \leq 39$, then the feature $\phi^{\text{space}}_{k_1,k_2,(i,j)}$ is 1.

Finally, features are generated to capture the relative offsets between the blobs on the current screen and a screen from the past. These features help deal with some of the partial observability inherent in a single frame from the ALE. If there is a pair of blobs with colours $k_1, k_2 \in \{1, ..., 128\}$ separated by an offset (i, j), where the blobs of colours k_1 and k_2 are from the current screen and the screen 5 frames in the past respectively, then the feature $\phi^{\text{time}}_{k_1,k_2,(i,j)}$ is 1.

This procedure results in a total of 114,702,400 possible Blob-PROST features. However, the blobs are generally sparse, and thus many of the features are never generated.

2.7 Learning a Model in the ALE

To do planning in Dyna, a model of the environment's dynamics is required. However, learning a model that can predict future screens in Atari 2600 games is a hard problem. The screens are high dimensional and the games often have complex dynamics. A particular line of work (Bellemare, Veness, & Bowling, 2013; Bellemare et al., 2014), divides the game screen into patches and applies a Bayesian framework to predict the next screen. An attractive alternative approach using a neural network for predicting frames in Atari was proposed by Oh et al. (2015a) and is the learned model that is used in this work. The model, which is discussed in the next section and employed in Chapter 4, is able to make multi-step predictions conditioned on actions.

2.7.1 Action-conditional Video Prediction

Oh et al. (2015a) introduced a deep-neural network architecture that was shown to make visually accurate predictions for hundreds of steps on video input from Atari games conditioned on actions. The architecture was presented in two variants: a feedforward encoding, and a recurrent encoding. For the purposes of this work only the feedforward version is examined.

The feedforward version of the architecture takes as input a stack of four frames (concatenated through channels). The architecture can be used to predict video frames in general, but specifically when used in combination with DQN, the input at time t would be the same as the input to the DQN network: four preprocessed frames, $\mathbf{x}_{t-3:t} \in \mathbb{R}^{84\times 84\times 4}$. The input is then encoded into a feature vector using a series of convolutional layers, followed by a single fully connected layer. The encoded feature vector at time t, $\mathbf{h}_t^{\text{enc}} \in \mathbb{R}^n$, given the stack of history frames, $\mathbf{x}_{t-3:t}$, is:

$$\mathbf{h}_t^{\text{enc}} = \texttt{encode}(\mathbf{x}_{t-3:t}). \tag{2.10}$$

The effect of the action is applied via a multiplicative interaction between the feature vector and the action as suggested by Memisevic (2013):

$$h_{t,i}^{\text{dec}} = \sum_{j,l} W_{ijl} h_{t,j}^{\text{enc}} a_{t,l} + b_i, \tag{2.11}$$

where $\mathbf{h}_t^{\text{dec}} \in \mathbb{R}^n$ is the action-transformed feature vector, $\mathbf{a}_t \in \mathbb{R}^z$ is a one-hot encoding of the action, $\mathbf{W} \in \mathbb{R}^{n\times n\times z}$ is a 3-way weight tensor, and $\mathbf{b} \in \mathbb{R}^n$ is a bias. Computing $\mathbf{h}_t^{\text{dec}}$ can be prohibitively expensive if $\mathbf{W}$ is large. However,

$\mathbf{W}$ can be approximated by factorizing into three matrices:

$$W_{ijl} = \sum_f W^{\text{enc}}_{jf} W^{\text{dec}}_{if} W^{\text{act}}_{lf}, \tag{2.12}$$

where f is the factor number, $\mathbf{W}^{\text{enc}} \in \mathbb{R}^{f \times n}$, $\mathbf{W}^{\text{dec}} \in \mathbb{R}^{n \times f}$, and $\mathbf{W}^{\text{act}} \in \mathbb{R}^{f \times z}$. The number of factors is a hyper-parameter, and Oh et al. (2015a) used 2048. Then, substituting the factorized weight matrix in (2.11), $\mathbf{h}^{\text{dec}}_t$ becomes:

$$\begin{aligned} h^{\text{dec}}_{t,i} &= \sum_{j,l} \Big(\sum_f W^{\text{enc}}_{jf} W^{\text{dec}}_{if} W^{\text{act}}_{lf} \Big) h^{\text{enc}}_{t,j} a_{t,l} + b_i, \\ &= \sum_f W^{\text{dec}}_{if} \Big(\sum_j W^{\text{enc}}_{jf} h^{\text{enc}}_{t,j} \Big) \Big(\sum_l W^{\text{act}}_{lf} a_{t,l} \Big) + b_i. \end{aligned} \tag{2.13}$$

Finally, the transformation can be re-written more compactly as:

$$\mathbf{h}^{\text{dec}}_t = \mathbf{W}^{\text{dec}} \left(\mathbf{W}^{\text{enc}} \mathbf{h}^{\text{enc}}_t \odot \mathbf{W}^{\text{act}} \mathbf{a}_t \right) + \mathbf{b}. \tag{2.14}$$

After the action transformation, the resulting vector is decoded using a single fully connected layer followed by a series of deconvolutions before finally outputting the single next predicted frame:

$$\hat{\mathbf{x}}_{t+1} = \texttt{decode}(\mathbf{h}^{\text{dec}}_t). \tag{2.15}$$

The model can be used to make k-step predictions by concatenating the predicted frame with the most recent three history frames, and running the model forward another step. For example, if the model has just predicted the frame at time $t+1$, $\hat{\mathbf{x}}_{t+1}$, and we wish to predict the frame at time $t+2$, the input to the network will be $(\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_t, \hat{\mathbf{x}}_{t+1})$. The output of the network will be $\hat{\mathbf{x}}_{t+2}$. To predict the frame at time $t+3$, the input to the network will be $(\mathbf{x}_{t-1}, \mathbf{x}_t, \hat{\mathbf{x}}_{t+1}\hat{\mathbf{x}}_{t+2})$, etc... A diagram of the model is shown in Figure 2.2.

To train the model, Oh et al. (2015a) created a training data set by running a trained DQN agent and recorded the actions and frames. Then, batches of image histories, actions, and image targets are drawn and used to train the model to minimize the average squared error between the predicted and target frames (denoted $\hat{\mathbf{x}}$ and $\mathbf{x}$ respectively) over the k-steps:

$$\mathcal{L}_k(\boldsymbol{\theta}) = \frac{1}{2k} \sum_{\kappa=1}^{k} \left\| \hat{\mathbf{x}}_\kappa - \mathbf{x}_\kappa \right\|^2. \tag{2.16}$$

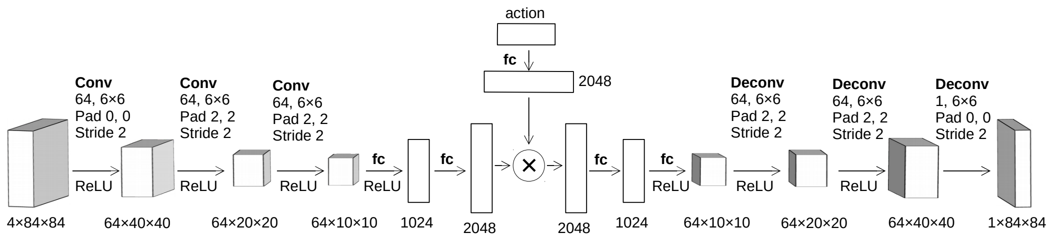

action
fc
2048
Conv 64, 6×6 Pad 0, 0 Stride 2
Conv 64, 6×6 Pad 2, 2 Stride 2
Conv 64, 6×6 Pad 2, 2 Stride 2
fc
fc
fc
fc
Deconv 64, 6×6 Pad 2, 2 Stride 2
Deconv 64, 6×6 Pad 2, 2 Stride 2
Deconv 1, 6×6 Pad 0, 0 Stride 2
ReLU
4×84×84
64×40×40
64×20×20
64×10×10
1024
2048
2048
1024
64×10×10
64×20×20
64×40×40
1×84×84

Figure 2.2: The feedforward grayscale version of Oh et al.'s (2015a) action-conditional video prediction model.

To increase stability during training a curriculum approach is used: the model is first trained to make 1-step, 3-step, then 5-step predictions.

The loss function being minimized (squared error) makes this model an expectation model. Unlike a sample model, which predicts a particular outcome, an expectation model learns to predict what will happen on average. Thus, when the environment is stochastic, predictions will be some combination of the possible outcomes that minimizes the loss on the training data. This is potentially problematic when making multi-step predictions since to roll out, the model predictions are treated as samples and used to predict the frame, but the model might predict combinations of frames that are not real frames that could ever be generated by the emulator. Nevertheless, making multi-step predictions in this way appears to work well in practice. This could be due in part to the relatively low stochasticity imparted by sticky actions in the ALE, and to the curiculum training approach; the model learns to make predictions based on its own potentially wrong predictions, which helps the model remain stable during rollouts.

Chapter 3

Planning with a Perfect Model

This chapter explores the idealized performance of two algorithms that make use of Dyna-style planning — Dyna-DQN and Dyna-Sarsa — by assuming a perfect sample model is available. This isolates the effects of planning from the accuracy of the model and provides an estimate of the maximum benefit that the model can provide.

3.1 The Ineffectiveness of One-step Rollouts

Planning in the Dyna architecture is accomplished by using a model to make predictions of future states based on a start state and action. When the state is high dimensional, like an image, it is not clear how to generate reasonable start states. One solution is to sample the start state from the previously observed states, as in Dyna-Q. The advantage of selecting states in this way is that the distribution or structure of the state space does not need to be known, but planning may not well cover the state space. To explore this strategy DQN and Sarsa with linear function approximation are extended to incorporate Dyna-style planning, and their performance is evaluated in several Atari games.

3.1.1 Dyna-DQN

It is straightforward to extend DQN to use the Dyna architecture. After every step taken in the environment, simulated experiences are generated starting from a state sampled from a planning buffer containing the agent's recent real experience. Keeping a separate planning buffer from the experience replay buffer ensures that start states are always from the agent's actual experience. Using the agent's current policy to select actions, transitions are simulated from the start states using a model and placed into the experience replay buffer alongside the transitions observed from the real environment. Training continues to happen after every f observed transitions — real or simulated. As a result, mini-batches sampled at training time will contain a mix of real and simulated experience.

Experiments

Experiments were run on six games from the ALE. Sticky actions were used to inject stocasticity into the emulator (repeat_action_probability = 0.25), as suggested by Machado et al. (2017). Each game usually has 18 possible actions, but some actions are redundant in certain games. Therefore, the minimal action set like Mnih et al. (2015) was used.

The games chosen for the experiments are from the original training set outlined by Bellemare, Naddaf, et al. (2013), supplemented with two additional games, Q-Bert and Ms. Pac-Man, that Oh et al. (2015a) used to evaluate their model learning approach, which is employed in Chapter 4. Results in Freeway have been omitted since the implementations of DQN and Dyna-DQN used in this work almost always score zero points at the number of training frames used in the experiments.

The implementation of DQN used the same hyper-parameters as Mnih et al. (2015) with a couple of small changes used by Machado et al. (2017) (see Table 3.1). At each step, the DQN algorithm has an ϵ probability of selecting a random action instead of the best action. ϵ was annealed from 1.0 to 0.01 over the first 10% of frames (real and simulated) during learning. The frame skip is the number of times a selected action is repeated before a new action is selected. A frame skip of 5 was used. Additionally, to reduce memory use, an experience replay buffer size of 500k transitions instead of the original 1M was used.

In these experiments, Dyna-DQN made use of an environment model that

Table 3.1: DQN Hyperparameters used in the experiments.

Hyperparameter	Value	Description
Step-size (α)	0.00025	Step size used by the RMSProp optimizer.
Gradient momentum	0.95	Gradient momentum used by RMSProp.
Squared gradient momentum	0.95	Squared gradient momentum used by RMSProp (in the denominator).
Min squared gradient	0.01	Constant added to the denominator of the RMSProp update.
Discount factor (γ)	0.99	Discount factor used in the Q-learning update rule.
Initial exploration rate (ϵ)	1.0	Initial probability that a random action will be taken at each time step.
Final exploration rate (ϵ)	0.01	Final probability that a random action will be taken at each time step.
Final exploration frame	10%	Percentage of total training frames over which to linearly anneal ϵ.
Minibatch size	32	The number of states over which the value function update is computed.
Replay memory size	500,000	The number of states to keep in the replay buffer.
Replay start size	50,000	The number of steps using the random policy used to populate the replay buffer at the start of training.
Agent history length	4	The number of recent frames observed by the agent that are input into the network.
Training frequency	4	The number of agent steps between updates to the value function.
Target network update frequency	40,000	The frequency (in terms of agent steps) at which the estimator network parameters are copied to the target network parameters.

was a perfect copy of the emulator. Start states for planning were selected from the planning buffer containing the 10,000 most recent real states observed by the agent, which for all games was multiple episodes of experience. In preliminary experiments it was found that larger planning buffers tended to perform better than smaller planning buffer, but little to no benefit was observed by increasing the buffer size beyond 10,000 states. For each real step,

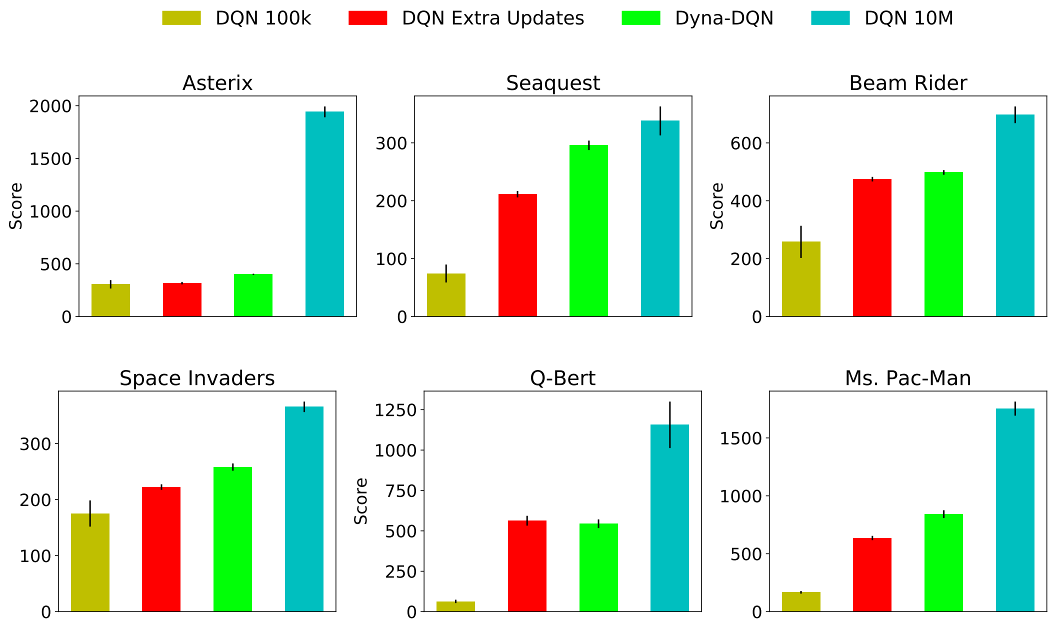


Figure 3.1: The results of running Dyna-DQN on six games from the ALE compared to several DQN baselines. Dyna-DQN provides almost no benefit over simply doing more updates with the same amount of data from the environment.

Dyna-DQN did 100 iterations of planning. Dyna-DQN was trained for 100k real frames, or equivalently 10M combined model and real frames. The training frequency was every 4 steps of real and model experience. After training, the mean score in 100 evaluation episodes using a fixed $\epsilon = 0.01$ was recorded. This training and evaluation procedure was repeated for 30 independent runs. The mean scores and standard errors for the six games are shown in Figure 3.1 (bright green bars are Dyna-DQN). To better understand the benefit of model-based updates, Dyna-DQN was also compared to the following model-free DQN baselines.

DQN 100k: DQN trained only for 100k real frames (yellow bars). This allows us to compare DQN and Dyna-DQN with an equivalent amount of real experience. This benchmark serves as a sanity check to show that using the perfect model to gather additional data does improve sample complexity. As expected, Dyna-DQN outperformed DQN 100k; it uses the model to generate more experience and does many more updates. However, this benchmark does not indicate whether the performance increase is due to the additional data generated by Dyna-DQN, or the extra updates to the value function completed during planning.

DQN Extra Updates: DQN trained for 100k real frames, but with the same number of updates to the value function as Dyna-DQN (red bars). For each time DQN would normally perform a single training step, DQN Extra Updates performs 100 training steps. This way DQN Extra Updates is like Dyna-DQN, but it uses only experience gathered from the environment, while Dyna-DQN also generates experience from the model. DQN Extra Updates allows us to evaluate the advantage of using the model to generate new experience compared to simply doing more updates with the real experience. Surprisingly, in every game except SEAQUEST, Dyna-DQN provided little benefit over DQN Extra Updates, even with a perfect model. This indicates that most of the benefit of planning was from simply updating the value function more often, which does not require a model.

DQN 10M: DQN trained for 10M frames (cyan bars). This allows us to compare DQN and Dyna-DQN with an equivalent amount of total experience. One might hope the experience generated by a perfect model would allow Dyna-DQN to perform comparably to this baseline, but in most games the performance of Dyna-DQN did not approach that of DQN 10M. This shows that there is significant room to improve the performance. Dyna-DQN and DQN 10M both gather additional data from the true system and perform the same number of updates; the only difference is the distribution over the start states of the additional transitions.

These experiments used only one set of hyperparameters for both DQN and Dyna-DQN, and therefore the strength of the conclusions may be limited. However, many of the parameters should not interact with planning and were kept consistent across all the experiments. The discount factor and the frame skip remained the same in each case, and can be considered part of the problem rather than the agent. The optimizer parameters, such as the step-size, should not interact with planning since 1-step update targets are always used, and the network inputs and rewards will be on the same scale whether it is a real or simulated transition. The exploration rate was kept the same in each condition, and was annealed in the same way. The minibatch size remained constant to be consistent, and to ensure that the value function updates were computed over the same number of samples. The agent history length specifies the approximate state that is used to make decisions, and thus was the same in each case. The training frequency affects how many updates are made to the value function given a fixed number of training frames, and thus should remain the same in each case. Nevertheless, the DQN 100k baseline did 100 times fewer updates than Dyna-DQN and DQN 10M since it saw fewer training frames. The DQN Extra Updates baseline attempts to normalize this and provide an additional point for comparison by doing 100 times more updates than DQN 100k, but using the same amount of real experience. Changing the target network update frequency might affect stability and is closely related to the training frequency. Since the training frequency remained the same in each case, the target network update frequency was kept the same as well.

The only hyperparameter that might interact with planning is the size of the experience replay buffer. With Dyna-DQN, for every real step there was 100 planning steps. As a result, there was 100 times more simulated experience in the buffer than real experience. Therefore, the amount of real experience contained in Dyna-DQN's replay buffer is much less than that contained in DQN's replay buffer. This difference could have some effect on learning, and having an larger experience replay buffer that contains a larger amount of real experience might affect performance. Due to computational restrictions, the size of the buffer could not be increased, and thus this hypothesis was not tested. However, a similar set of experiments was repeated using Sarsa with Blob-PROST features (as described in the next section), which does not use an experience replay buffer, and a similar trend was observed. This could indicate that if the amount of real experience contained in the buffer does affect performance, the effect is likely small.

Table 3.2: Sarsa with Blob-Prost Features Hyperparameters

Hyperparameter	Value	Description
Step-size (α)	0.5	Step size used in the Sarsa update rule.
Discount factor (γ)	0.99	Discount factor in the Sarsa update rule.
Exploration rate (ϵ)	0.01	Probability of selecting a random action at each time step.
Feature set	Blob-PROST	The feature set used with Sarsa. See Section 2.6.1 for details.

3.1.2 Dyna-Sarsa

Sarsa can also be extended to use the Dyna architecture. As with Dyna-DQN, after every step taken in the environment, simulated experiences are generated starting from a state sampled from a planning buffer containing the agent's recent real experience. However, instead of placing the transitions in an experience replay buffer, the value function is updated with the transition immediately using the normal Sarsa update rule.

Experiments

The experimental setup is the same as for Dyna-DQN in Section 3.1.1, but Sarsa with linear function approximation and Blob-PROST features (Liang et al., 2016) was used (see Table 3.2). The same hyper-parameters as Liang et al. (2016) were used, except $\lambda = 0$, instead of $\lambda = 0.9$, to better isolate the effects of planning shape from any interactions with eligibility traces.

The start states for planning were selected uniformly at random from the 10k most recent states experienced by the agent. Like with Dyna-DQN, for these experiments the agent was assumed to have access to a perfect model. The results are shown in Figure 3.2. The reported scores are the mean for each algorithm in 100 evaluation episodes after learning for 10M combined real and model frames, and are an average of 30 independent runs. The model-free baselines that Dyna-Sarsa was compared to are similar to the ones used for Dyna-DQN.

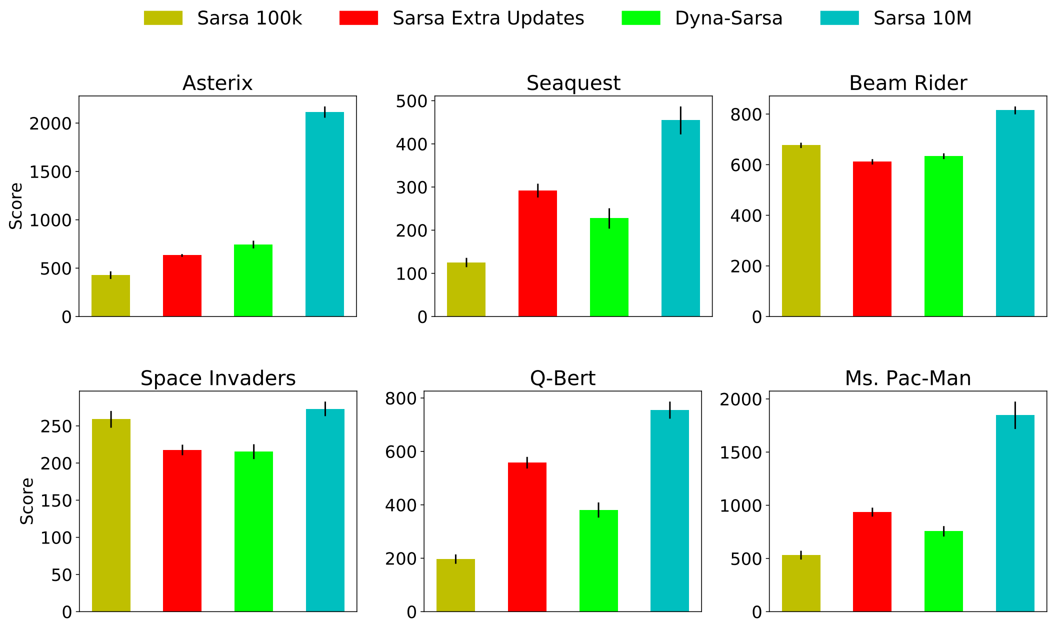


Figure 3.2: The results of running Dyna-Sarsa on six games from the ALE compared to several Sarsa baselines. Similar to Dyna-DQN, Dyna-Sarsa provides no benefit over simply doing more updates with the same amount of data from the environment.

Sarsa 100k: Sarsa agent trained for 100k real frames (yellow bars). Unlike Dyna-DQN, Dyna-Sarsa did not strongly outperform this baseline even though it does many more updates.

Sarsa Extra Updates: Sarsa trained for 100k real frames, except after every real step it does 100 extra updates using experiences sampled from the agent's recent history (red bars). Similarly to Dyna-DQN, in every game Dyna-Sarsa provided no benefit over Sarsa Extra Updates. In some games like Q-Bert, Dyna-Sarsa even performed significantly worse.

Sarsa 10M: Sarsa agent trained for 10M real frames (cyan bars). As was the case with Dyna-DQN, the performance of Dyna-Sarsa did not approach that of Sarsa 10M.

Like with the Dyna-DQN experiments, these experiments use only one set of hyperparameters. However, like the previous experiments, the hyperparameters should not interact with planning and are kept consistent across all the experiments. The discount factor and the frame skip remained the same in each case, and are parameters that are chosen as part of the problem. The exploration rate was kept constant (nearly completely greedy) in every case to remain consistent. The same feature set and step size were also used in each case. These choices are unlikely to interact with planning since in these experiments the model is perfect, thus the features generated by Blob-PROST, and the scale and distribution of the rewards, will be similar with both the real and simulated experience.

Overall, the extra computation required by both Dyna-DQN and Dyna-Sarsa to utilize the model does not appear to be worth the effort. A possible explanation for these results is that planning in this way — taking a single step from a previously visited state — does not provide data that is much different than what is already contained in the experience replay buffer. If true, a strategy is needed to make the data generated by the model different from what was already experienced.

3.2 Planning with Longer Rollouts

One possible strategy to generate more diverse experience is to roll out more than a single step from the start state during planning, as was done by Gu et al. (2016) and Kalweit and Boedecker (2017). Since the current policy will be used for the rollout, the model may generate a different trajectory than what was originally observed. Longer rollouts would also allow the agent to see the longer-term consequences of exploratory actions or alternative stochastic outcomes in the environment.

It is straightforward to modify each planning iteration of Dyna-DQN and Dyna-Sarsa so that instead of rolling out a single step, the model is used to rollout k steps from the start state, producing a sequence of k states and rewards. Let these algorithms be called Rollout-Dyna-DQN and Rollout-Dyna-Sarsa. With Rollout-Dyna-DQN the transitions observed during the rollouts are placed in the experience replay buffer, and training continues to occur and at the training frequency, f, of combined steps in both the real environment and the model. With Rollout-Dyna-Sarsa the value function is updated using

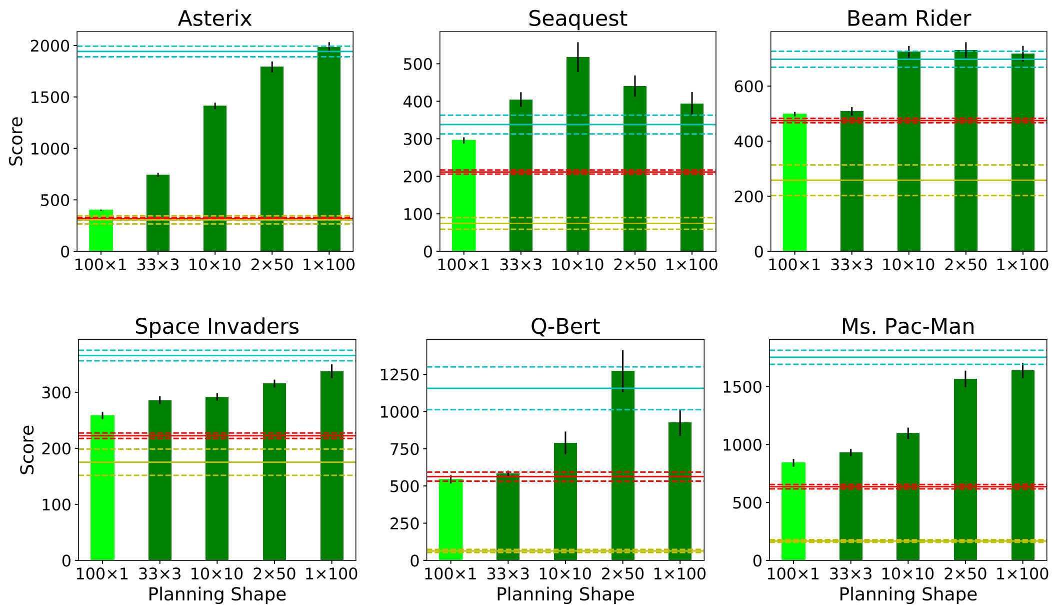


Figure 3.3: The results of running Rollout-Dyna-DQN on six games from the ALE compared to the DQN baselines. The performance of Rollout-Dyna-DQN tends to increase as the rollout length increases across all the games. The horizontal lines show the same baseline scores as Figure 3.1.

the normal Sarsa update rule after every step of the rollouts. When $k = 1$ we recover exactly Dyna-DQN and Dyna-Sarsa described in Sections 3.1.1 and 3.1.2.

Given a budget of planning time in terms of a fixed number of model prediction steps, planning could take on a variety of shapes. Let the planning shape be described by the notation $n \times k$, where n is the number of planning iterations. For example: 100 rollouts of 1 step (100×1); 10 rollouts of 10 steps (10×10); or 1 rollout of 100 steps (1×100), each require the same amount of computation from the model, but the way that the resulting predictions are distributed in the state space are very different. In the next section we investigate the effects of planning shape on the performance of Rollout-Dyna-DQN and Rollout-Dyna-Sarsa compared to the baselines.

3.2.1 Experiments

The experimental setup is the same as in Sections 3.1.1 and 3.1.2, but now the planning shape for Dyna-DQN and Dyna-Sarsa is varied. 100×1, 33×3, 10×10, 2×50, and 1×100 planning shapes were evaluated.

The results for Rollout-Dyna-DQN in the six games are shown in dark green in Figure 3.3. The 100×1 planning shape is the same as Dyna-DQN and remains bright green. Note that the ratio of real transitions to simulated transitions remains the same in each case. The DQN baselines are the same as in Section 3.1.1 and are shown as horizontal lines: yellow is DQN 100k, red is DQN Extra Updates, and cyan is DQN 10M.

In each game there was a longer rollout length that resulted in a dramatic improvement over 100×1 planning, significantly outperforming DQN Extra Updates. Further, in every game, there was a planning shape that approached the performance of DQN 10M.

Like in the previous experiments, one set of hyperparemeters was used, but now the planning shape is varied. However, the only hyperparameter that planning shape might interact with is the experience replay buffer size. As was the case before, the amount of real experience in the replay buffer of Rollout-Dyna-DQN is much less than DQN, and this could affect performance in some way. Although it is not clear how planning shape specifically would interact with the buffer size, since the ratio of real experience to simulated experience is the same in each case. Despite this caveat, the positive impact of longer rollouts is also not specific to DQN.

Similar results to Rollout-Dyna-DQN were obtained with Rollout-Dyna-Sarsa, shown in Figure 3.4. The Sarsa baselines are the same as in Section 3.1.2: yellow is Sarsa 100k, red is Sarsa Extra Updates, and cyan is Sarsa 10M. Similar to Rollout-Dyna-DQN, in every game there was a planning shape with a rollout length greater than one that outperformed both 100×1 planning and Sarsa Extra Updates. These results demonstrate that this phenomenon is not specific to only DQN.

Again, recall that the only difference between two planning shapes is the distribution of experience generated by the model. Thus, the results suggest that with the Dyna architecture it is critical for the model to generate sufficiently novel experience, and using multi-step rollouts appears to be an effective strategy. Doing longer rollouts during planning makes using the model

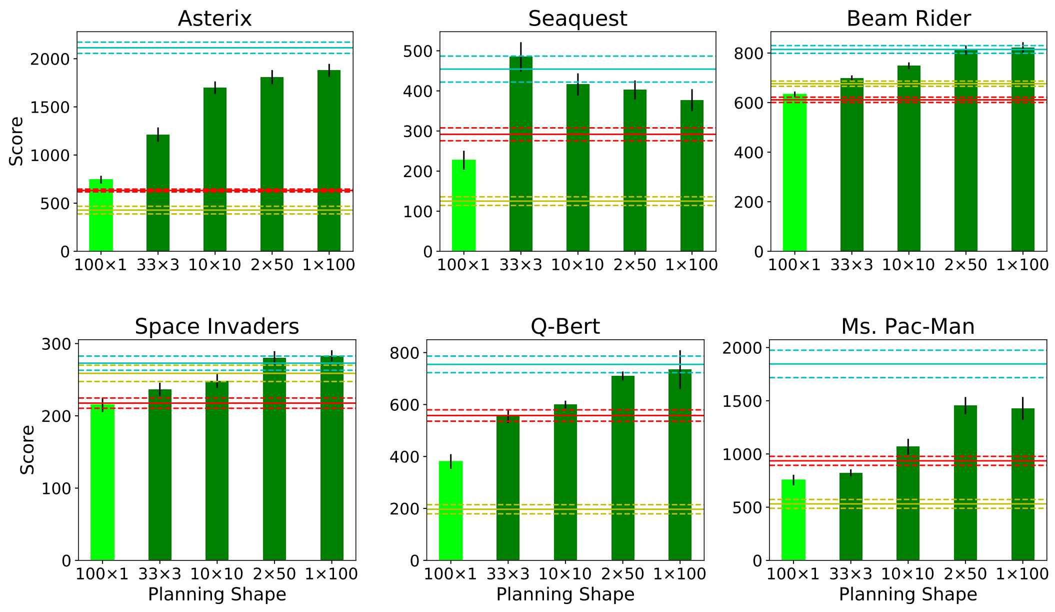


Figure 3.4: The results of running Rollout-Dyna-Sarsa on six games from the ALE compared to the baselines. There is a planning shape with a rollout length greater than one that outperforms both 100×1 and Sarsa Extra Updates (red line) across all the games.

worth the effort whereas the 100×1 planning is no better than doing extra updates with only real experience. Also, recall that in these experiments the agent had access to a perfect model. With a learned model, performance will likely be worse due to model errors, so rollouts may be the only way to obtain a benefit over simply performing extra updates using the agent's real experience.

Chapter 4

Planning with an Imperfect Model

In the previous chapter we have drawn conclusions from an ideal setting, but if the agent has an imperfect model do the same conclusions hold? To investigate this question the perfect copy of the emulator was replaced with a learned model. First, using a learned model pre-trained on data from expert play is explored, then results for a model learned online alongside the value function are presented.

Previously, we have seen that increasing the length of the rollouts during planning tends to provide an increased benefit to performance. However, since the model is now imperfect, there is a limit on how far it can roll out before small errors compound and make the predictions unreliable (e.g. Talvitie, 2014). Therefore, it is reasonable to hypothesize that there will be competing effects: the algorithm benefits from long rollouts, but the model's performance degrades as rollout length increases. This may result in the best performance at shorter rollout lengths.

4.1 The Imperfect Model

In its original formulation, Oh et al.'s (2015a) action-conditional video prediction model described in Section 2.7.1 predicts only the next state, but an environment model for reinforcement learning needs to predict both the next state and the next reward. Therefore, the model was extended to make reward

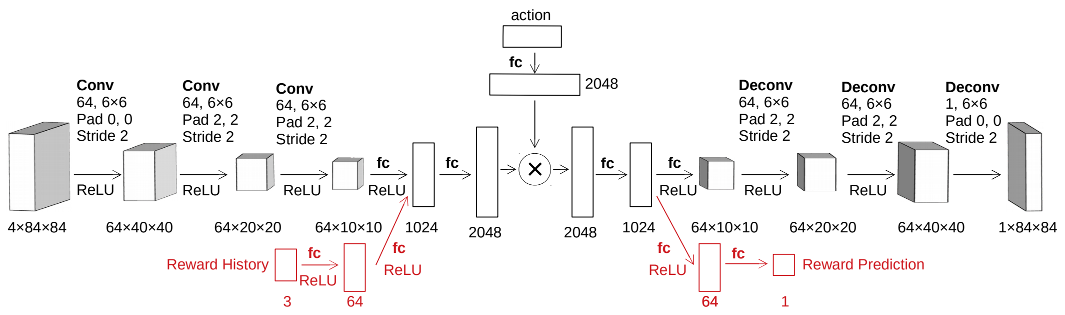


Figure 4.1: The feedforward grayscale version of Oh et al.'s (2015a) action-conditional video prediction model (black) and its extension to predict the next reward (red).

predictions by adding a separate fully connected layer after the action transformation, followed by an output layer that predicts a single scalar reward (shown in red in Figure 4.1). Thus, in addition to the k-step image reconstruction loss, the model is trained to minimize the k-step squared difference between the predicted, and target rewards ($\hat{r}$ and r respectively):

$$\mathcal{L}_k(\boldsymbol{\theta}) = \frac{1}{2k} \sum_{\kappa=1}^{k} \left(\|\hat{\mathbf{x}}_\kappa - \mathbf{x}_\kappa\|^2 + \|\hat{r}_\kappa - r_\kappa\|^2 \right). \tag{4.1}$$

This approach is similar to what was used by Leibfried, Kushman, and Hofmann (2017) to jointly predict frames and rewards. As input, the reward history is provided for the three transitions associated with the input frames. After the reward history input layer, there is a fully connected layer, before joining with the output of the encoder at the beginning of the action transformation. Since DQN clips the rewards to the interval $[-1, 1]$, the input and target rewards for the model are also clipped to the same interval. This new architecture was used to train three different models for each game. Details of the training procedures can be found in Appendix A.

4.1.1 Rollout-Dyna-DQN Experiments

For this section, the experimental setup is the same as in Chapter 3, but the perfect model has been replaced with an imperfect model trained using the

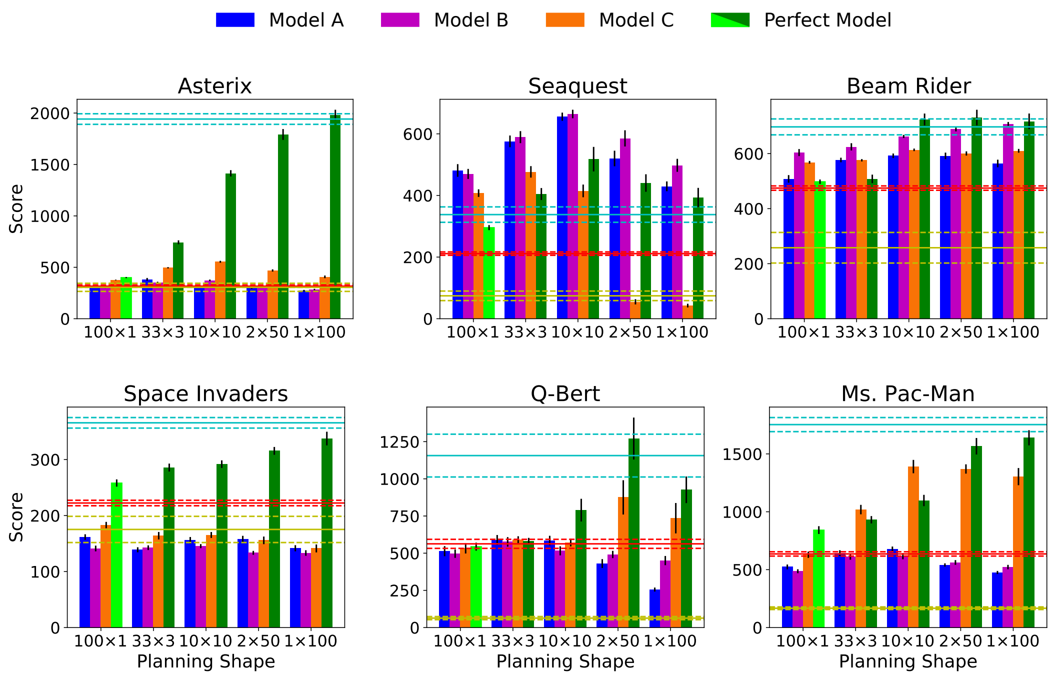


Figure 4.2: The results of running Rollout-Dyna-DQN with the perfect and imperfect models on six games from the ALE. Like the perfect model, using an imperfect model with a rollout length greater than one provides the most benefit. The horizontal lines show the same baseline scores as Figure 3.1.

approach described above.[1] Three different models were pre-trained and evaluated with Rollout-Dyna-DQN to ensure that any trends that were observed were not specific to a particular model. With each of the three models held fixed, the experiment from Section 3.2.1 was repeated, measuring Rollout-Dyna-DQN's performance with various planning shapes. Note that because each model is pre-trained on a single dataset, our results cannot be used to draw reliable conclusions about the comparative effectiveness of the different training regimes. The aim in this experiment is only to study the impact of model error on Rollout-Dyna-DQN. As such, the models are referred to merely as Models A, B, and C. The results of applying Rollout-Dyna-DQN with the three imperfect models are shown in Figure 4.2. The perfect model results and baselines are the same as in Figure 3.3.

As with the perfect model, rollouts longer than one step provided the most

[1]The performance of the learned models is explored in a different context using 1-ply Monte-Carlo planning in Appendix B.

benefit. For every game, except for Space Invaders, there was a model and planning shape that performed better than DQN Extra Updates, which demonstrates that even when the model has flaws, planning with rollouts can provide some benefit. The reason the performance was poor in Space Invaders is that the model had trouble predicting bullets, which is fundamental to scoring points in the game. Oh et al. (2015a) attribute this flaw to the low error signal produced by small objects, which can make it difficult to learn about small details in the image.

The results also support the hypothesis that there is a trade-off between the benefits of long rollouts for planning and the model's error increasing with rollout length. For example, in Asterix using Model C, the performance peaked at 10×10 planning and dropped off as rollouts became shorter or longer. Similarly, in most other game and model combinations the best-performance was achieved at medium rollout length.

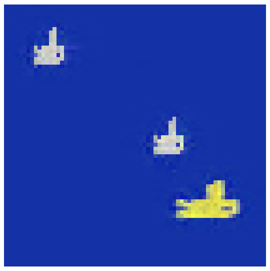
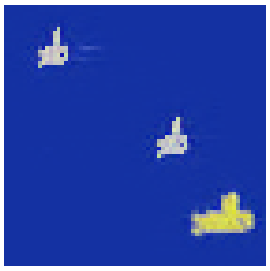
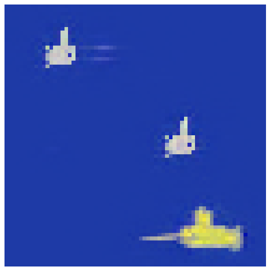
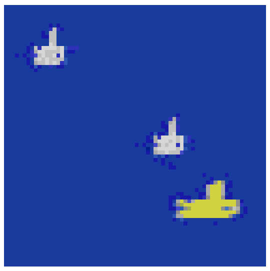
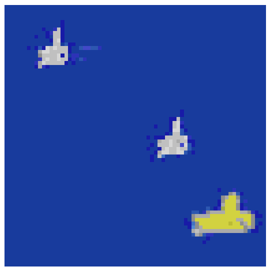
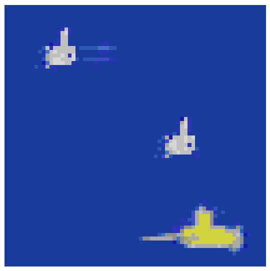
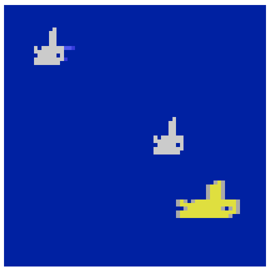
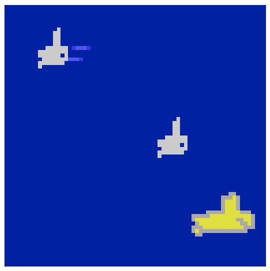
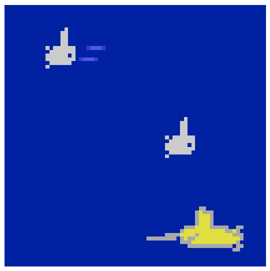
4.1.2 Rollout-Dyna-Sarsa Experiments

The predictions of the imperfect model worked well with Rollout-Dyna-DQN, but in preliminary experiments it was found that the model does not work well with Sarsa and Blob-PROST features.

The root of the problem is likely that Blob-PROST operates on images encoded in the NTSC colour palette, which has 128 indexed colours, but the output of the colour version of Oh et al.'s (2015a) model is a three channel RGB image. Therefore, each pixel from the model prediction needs to be projected to the nearest NTSC palette colour before it can be used with Blob-PROST. Since the model predictions will not always be able to reproduce the target image exactly, and there will be some amount of noise in the predicted image, there may be artifacts in the resulting projected image. Because of the way Blob-PROST detects blobs — by finding contiguous regions of pixels that are all the same colour — model predictions with artifacts contain many features that would never occur in the real emulator screens; even a single pixel that is a different colour from its surroundings will be treated as a new blob.

An example of this effect in Seaquest during a model rollout can can be

Table 4.1: The problem that results when projecting the RGB predictions from the model to the NTSC colourspace. Small artifacts appear around the submarines that are not present in the ground truth images. This results in many more features in the predictions than in the ground truth.

	Step 1	Step 2	Step 3
Model RGB prediction			
NTSC projection			
Features	4674	7187	9623
Ground truth			
Features	1271	1243	1080

seen in Table 4.1. The RGB model prediction images have a small amount of noise around the submarines (there is noise in other parts of the image as well that is not shown in the cropped images.) In the projected images, this noise turns into isolated pixels that are a different colour from the background. These isolated pixels, that are not present in the ground truth images results in many more features being extracted by Blob-PROST.

To test the impact of this effect, some preliminary experiments were conducted using the same setup as in Chapter 3, but the perfect model has been

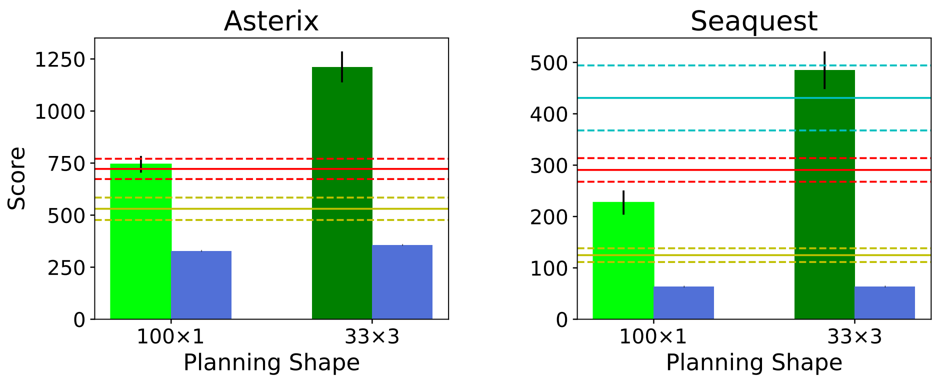


Figure 4.3: The results of running Rollout-Dyna-Sarsa with the perfect (green bars) and imperfect models (blue bars). The horizontal lines show the same baseline scores as Figure 3.2.

replaced with a learned model. The only other difference is that the reported scores are only an average of five runs, thus their error bars have been omitted.

Figure 4.3 shows the performance of Rollout-Dyna-Sarsa with the learned model in Asterix and Seaquest. The scores are poor and do not even reach the performace of Sarsa 100k (yellow line). The extra features appear to inhibit the ability of the Rollout-Dyna-Sarsa agent to learn effectively.

In contrast, the value function for DQN seemed to be much more robust to small errors in the image produced by the model. It may be that learning the feature representation using both the real and predicted images helps account for the model error. The network is likely able to ignore errors in the predicted images that are not important for predicting the value of a state.

4.2 Learning and Planning Online

In all the experiments so far, a perfect model or a pre-trained learned model has been used; the obvious next step is to study Rollout-Dyna-DQN in the case where the model is learned alongside the value function. Learning the model and value function together adheres to the original conception of Dyna.

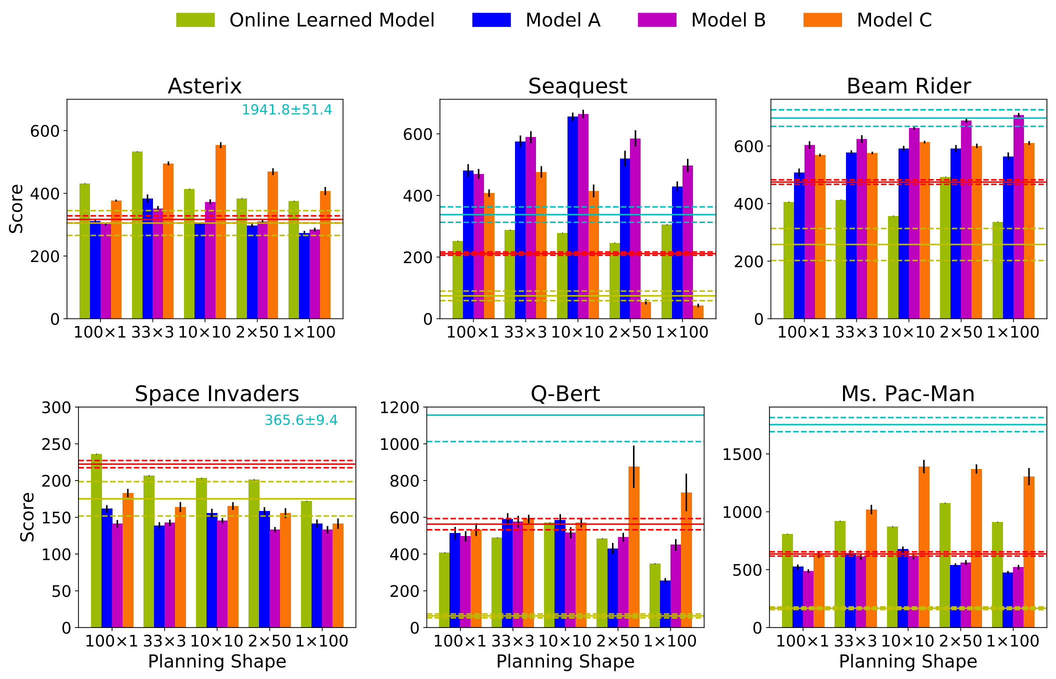


Figure 4.4: The results of running Rollout-Dyna-DQN with an online learned model, compared to Rollout-Dyna-DQN with the pre-trained learned models, on six games from the ALE. In Asterix, Seaquest, and Ms. Pac-Man, there was at least one planning shape that outperformed the DQN Extra Updates baseline (red horizontal line). Standard error bars have been omitted for Rollout-Dyna-DQN with an online learned model since these scores are only an average of 10 runs.

4.2.1 Experiments

The experimental setup was the same as in the previous experiments with Rollout-Dyna-DQN, except the model was learned online alongside the value function. Details of the online model training procedure are available in Appendix A.

The results for the six games from the ALE compared to the pre-trained learned models and the model-free baselines are shown in Figure 4.4. Due to computational limitations, only 10 independent runs for Rollout-Dyna-DQN with an online learned model were completed, thus the standard error bars have been omitted and strong conclusions about trends in performance cannot be made. In Asterix, Seaquest, and Ms. Pac-Man, Rollout-Dyna-DQN consistently outperformed the DQN Extra Updates baseline. As was the case with the perfect and pre-trained models, the best performance was achieved using a planning shape with a rollout length greater than one. In Beam Rider, Space Invaders, and Q-Bert, there was no planning shape with the online learned model that outperformed the DQN Extra Updates baseline. However, in Space Invaders and Q-Bert the pre-trained learned models generally did not outperform that baseline either. Overall, these results indicate that in some cases there may be an advantage to learning and planning with a dynamics model online, over simply doing more updates with the real experience. To the author's knowledge, this is the first time that this has been demonstrated in the ALE.

Chapter 5

Conclusion

Despite the introduction of increasingly effective approaches for learning predictive models in Atari Games (Bellemare, Veness, & Bowling, 2013; Bellemare et al., 2014; Oh et al., 2015a), this is the first time that a learned dynamics model has been successfully used for planning in this challenging domain. The results show that, combined with deep RL methods, Dyna is a promising approach for model-based RL in high-dimensional state spaces and that planning shape is a critical consideration in extracting the most benefit from the model. In every game from the ALE that was tested, the best performance was achieved using a rollout length greater than one. Even when the model was learned online and was necessarily imperfect, in some games there was a planning shape with a rollout length greater than one that outperformed DQN Extra Updates. Longer planning rollouts appears to be an effective strategy for generating novel experience, which seems to be necessary to use the model to its full potential.

The findings in this thesis suggest multiple next steps. Some of the model flaws observed by Oh et al. (2015a) were indeed harmful for planning — perhaps improvements in architecture or the introduction of new loss functions could benefit planning performance. In the experiments start states for planning were selected from a buffer containing the agent's recent real history; it would be interesting to study Rollout-Dyna-DQN where start states are generated, and may not have been visited by the agent. This would necessarily involve learning a generative model of the states, which might be accomplished with a solution like a variational autoencoder (Kingma & Welling, 2013) or a generative adversarial network (Goodfellow et al., 2014). Finally, though longer rollouts were found to be an effective way to use the model to generate experience, there are other promising approaches. For instance Pan, Zaheer, White, Patterson, and White (2018) and Goyal et al. (2018) use inverse dynamics models to effectively propagate value updates backwards in a manner similar to prioritized sweeping (Moore & Atkeson, 1993; Peng & Williams, 1993). It may be possible to combine these insights, exploiting a forward model's ability to reveal novel states and a backward model's ability to efficiently improve the value function.

Overall, there are several important conclusions from this work to consider when approaching a new problem with Dyna. The first is that the value function being learned should be robust to errors produced by the model; we saw that Blob-PROST features are sensitive to model error, while the neural network used by DQN appears to be more robust. Next, the model should be able to make multi-step predictions since the best performance might be achieved with rollouts longer than a singe step. Finally, planning shape is an important consideration to be able to extract the most benefit from the model.

References

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: an evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellemare, M. G., Veness, J., & Bowling, M. (2013). Bayesian learning of recursively factored environments. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (Vol. 28, *3*, pp. 1211–1219). Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR.

Bellemare, M. G., Veness, J., & Talvitie, E. (2014). Skip context tree switching. In E. P. Xing & T. Jebara (Eds.), *Proceedings of the 31st international conference on machine learning* (Vol. 32, *2*, pp. 1458–1466). Proceedings of Machine Learning Research. Bejing, China: PMLR.

Bellman, R. E. (1957). *Dynamic programming*. Princeton: Princeton University Press.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *CoRR*, *abs/1606.01540*. arXiv: 1606.01540. Retrieved from http://arxiv.org/abs/1606.01540

Chiappa, S., Racaniere, S., Wierstra, D., & Mohamed, S. (2017). Recurrent environment simulators. Conference paper at the International Conference on Learning Representations 2017. Retrieved from http://arxiv.org/abs/1704.02254

Faulkner, R. & Precup, D. (2010). Dyna planning using a feature based generative model. Presented at the NIPS 2010 Deep Learning and Unsupervised Feature Learning Workshop. Retrieved from https://deeplearningworkshopnips2010.files.wordpress.com/2010/11/nips2010_submission_cr1.pdf.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., . . . Bengio, Y. (2014). Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 2672–2680). Curran Associates, Inc.

Goyal, A., Brakel, P., Fedus, W., Lillicrap, T. P., Levine, S., Larochelle, H., & Bengio, Y. (2018). Recall traces: backtracking models for efficient reinforcement learning. *CoRR*, *abs/1804.00379*. arXiv: 1804.00379. Retrieved from http://arxiv.org/abs/1804.00379

Gu, S., Lillicrap, T., Sutskever, I., & Levine, S. (2016). Continuous deep Q-learning with model-based acceleration. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning* (Vol. 48, pp. 2829–2838). Proceedings of Machine Learning Research. New York, NY, USA: PMLR.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2018). Rainbow: combining improvements in deep reinforcement learning. In *AAAI conference on artificial intelligence.*

Kalweit, G. & Boedecker, J. (2017). Uncertainty-driven imagination for continuous deep reinforcement learning. In S. Levine, V. Vanhoucke, & K. Goldberg (Eds.), *Proceedings of the 1st annual conference on robot learning* (Vol. 78, pp. 195–206). Proceedings of Machine Learning Research. PMLR.

Kingma, D. P. & Ba, J. (2015). Adam: A method for stochastic optimization. Conference paper at the International Conference on Learning Representations 2015. Retrieved from http://arxiv.org/abs/1412.6980

Kingma, D. P. & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114.*

Leibfried, F., Kushman, N., & Hofmann, K. (2017). A deep learning approach for joint video frame and reward prediction in atari games. Presented at the ICML 2017 Workshop on Principled Approaches to Deep Learning. Retrieved from http://arxiv.org/abs/1611.07078

Liang, Y., Machado, M. C., Talvitie, E., & Bowling, M. (2016). State of the art control of Atari games using shallow reinforcement learning. In J. Thangarajah, K. Tuyls, S. Marsella, & C. Jonker (Eds.), *Proceedings of the 15th international conference on autonomous agents and multiagent systems (AAMAS 2016).*

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3-4), 293–321.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., & Bowling, M. (2017). Revisiting the arcade learning environment: evaluation protocols and open problems for general agents. *CoRR*, *abs/1709.06009.*

Memisevic, R. (2013). Learning to relate images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1829–1846.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Moore, A. W. & Atkeson, C. G. (1993). Prioritized sweeping: reinforcement learning with less data and less time. *Machine Learning*, *13*(1), 103–130.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., ... Silver, D. (2015). Massively parallel methods for deep reinforcement learning. Presented at the ICML 2015 Deep Learning Workshop. Retrieved from http://arxiv.org/abs/1507.04296

Oh, J., Guo, X., Lee, H., Lewis, R. L., & Singh, S. (2015a). Action-Conditional Video Prediction using Deep Networks in Atari Games. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett, & R. Garnett (Eds.), *Advances in neural information processing systems 28* (pp. 2845–2853). Curran Associates, Inc.

Oh, J., Guo, X., Lee, H., Lewis, R. L., & Singh, S. (2015b). Action-conditional video prediction using deep networks in atari games. https://github.com/junhyukoh/nips2015-action-conditional-video-prediction. GitHub.

Oh, J., Singh, S., & Lee, H. (2017). Value prediction network. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems 30* (pp. 6118–6128). Curran Associates, Inc.

Pan, Y., Zaheer, M., White, A., Patterson, A., & White, M. (2018). Organizing experience: a deeper look at replay mechanisms for sample-based planning in continuous state domains. To appear at the 27th International Joint Conference on Artificial Intelligence. Stockholm, Sweden.

Peng, J. & Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, *1*(4), 437–454.

Ross, S. & Bagnell, J. A. (2012). Agnostic system identification for model-based reinforcement learning. In J. Langford & J. Pineau (Eds.), *Proceedings of the 29th international conference on machine learning* (pp. 1703–1710). Edinbrugh, Scotland: Omnipress.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning: an introduction.* Cambridge, Massachusetts: MIT Press.

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: an introduction* (2nd). Manuscript in preparation.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In B. Porter & R. Mooney (Eds.), *Machine learning proceedings 1990* (pp. 216–224). San Francisco (CA): Morgan Kaufmann.

Sutton, R. S., Szepesvari, C., Geramifard, A., & Bowling, M. (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the twenty-fourth conference annual conference on uncertainty in artificial intelligence* (pp. 528–536). Corvallis, Oregon: AUAI Press.

Talvitie, E. (2014). Model regularization for stable sample rollouts. In *Proceedings of the 30th conference on uncertainty in artificial intelligence* (pp. 780–789).

Talvitie, E. (2015). Agnostic system identification for Monte Carlo planning. In *AAAI conference on artificial intelligence* (pp. 2986–2992).

Talvitie, E. (2017). Self-correcting models for model-based reinforcement learning. In *AAAI conference on artificial intelligence* (pp. 2597–2603).

Tamar, A., Wu, Y., Thomas, G., Levine, S., & Abbeel, P. (2016). Value iteration networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in neural information processing systems 29* (pp. 2154–2162). Curran Associates, Inc.

Tieleman, T. & Hinton, G. (2012). Lecture 6.5 – RMSProp: Divde the gradient by a running average of its recent magnitude. In *Neural networks for machine learning.* Coursera.

van Seijen, H. & Sutton, R. S. (2015). A deeper look at planning as learning from replay. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (Vol. 37, pp. 2314–2322). Proceedings of Machine Learning Research. Lille, France: PMLR.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation, Cambridge University).

Watkins, C. J. C. H. & Dayan, P. (1992). Q-learning. *Machine learning, 8*(3-4), 279–292.

Weber, T., Racanière, S., Reichert, D., Buesing, L., Guez, A., Jimenez Rezende, D., ... Wierstra, D. (2017). Imagination-augmented agents for deep reinforcement learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems 30* (pp. 5690–5701). Curran Associates, Inc.

Yao, H., Bhatnagar, S., Diao, D., Sutton, R. S., & Szepesvári, C. (2009). Multi-step Dyna planning for policy evaluation and control. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, & A. Culotta (Eds.), *Advances in neural information processing systems 22* (pp. 2187–2195). Curran Associates, Inc.

Appendix A

Details of Learned Model Training

This appendix describes how the learned dynamics models from Chapter 4 were trained. To train the models, a procedure similar to what was used by Oh et al. (2015a) was employed. In addition to the extension of the architecture to enable reward prediction, there were also two other changes from the original description. Instead of RMSProp (Tieleman & Hinton, 2012), the Adam optimizer was used (Kingma & Ba, 2015), which Oh et al. (2015b) found converged more quickly. And for preprocessing the images, instead of computing and subtracting a pixelwise mean, the mean value per channel was computed and subtracted (grayscale has one channel), following Chiappa, Racaniere, Wierstra, and Mohamed (2017).

Model A. For each game, a single DQN agent was trained for 10M emulator frames. The trained agent was then run for a series of episodes without learning, and 500k transitions (frames, actions, next frames, and rewards) were recorded to create the training set. The model was then trained, using the training set, for 1M updates with a 1-step prediction loss (batch size 32, learning rate 1×10^{-4}), followed by 1M updates with a 3-step prediction loss (batch size 8, learning rate 1×10^{-5}), for a total of 2M updates.

Model B. The procedure and training data was exactly the same as for Model A, except that it was trained for an additional 1M updates using a 5-step prediction loss (batch size 8, learning rate 1×10^{-5}), for a total of 3M updates.

Model C. For this model, several independent DQN agents at different times during their learning were used to collect the training data. For each game, five independent DQN agents were trained for 10M frames. Then, 25k transitions were recorded from evaluation episodes using a snapshot of each agent at 2.5M, 5M, 7.5M, and 10M frames during their learning. The resulting 500k transitions were then combined to create the training set. The model was then trained for 1M updates with a 1-step prediction loss (batch size 32, learning rate 1×10^{-4}), followed by 500k updates with a 3-step prediction loss (batch size 8, learning rate 1×10^{-5}), then finally 500k updates using a 5-step prediction loss (batch size 8, learning rate 1×10^{-5}), for a total of 2M updates.

Online Learned Model. To train the model online, batches of data are sampled from the agent's real experience in the experience replay buffer. The model is first trained on 1-step predictions using a learning rate of 1×10^{-4} for 125k updates (500k agent steps, with training occurring every 4 steps), before switching to 3-step predictions with a learning rate of 1×10^{-5}. The batch size is 32 for both phases of training.

Appendix B

1-Ply Monte-Carlo Experiments

To explore the effectiveness of the learned models in a different context from Dyna-DQN, several experiments using 1-ply Monte-Carlo planning were conducted using the six games from the ALE. Monte-Carlo planning isolates using the model from learning a policy, which may provide insight into how well the models have learned the environment's dynamics.

In 1-ply Monte-Carlo planning, at each time step t, an estimate of the return for every possible action is obtained by using the model to simulate rollouts from the next states that result from taking each action. Specifically, from the current state S_t, the result of taking each action $a_i \in \mathcal{A}$ is simulated using the model to obtain a set of next states and rewards $\mathcal{S}_{\text{next}} = \{S^i_{t+1} \mid i \in 1, 2, ..., |\mathcal{A}|\}$, and $\mathcal{R}_{\text{next}} = \{R^i_{t+1} \mid i \in 1, 2, ..., |\mathcal{A}|\}$. Then from the each $S^i_{t+1} \in \mathcal{S}_{\text{next}}$, k random rollouts of length n are performed. Let the notation $k \times n$ be referred to as the rollout shape. Next, the mean n-step return,

$$\tilde{G}^i_{t:t+1+n} = R^i_{t+1} + \frac{1}{k} \sum_j^k \sum_l^n \gamma^l R^{i,j}_{t+1+l},$$

for all the rollouts is computed. Finally, the action a_i that had the highest estimated return is then executed in the real environment.

Several experiments were conducted using 1-ply Monte-Carlo planning using both the learned models A, B, and C, and the perfect model. 5×10, 10×5, 10×10, and 5×20, rollout shapes were evaluated. Note that the pairs 5×10 and 10×5, and 10×10 and 5×20 share the same number of simulation steps. In each case $\gamma = 1$. The results of these experiments are shown in Figure B.1.

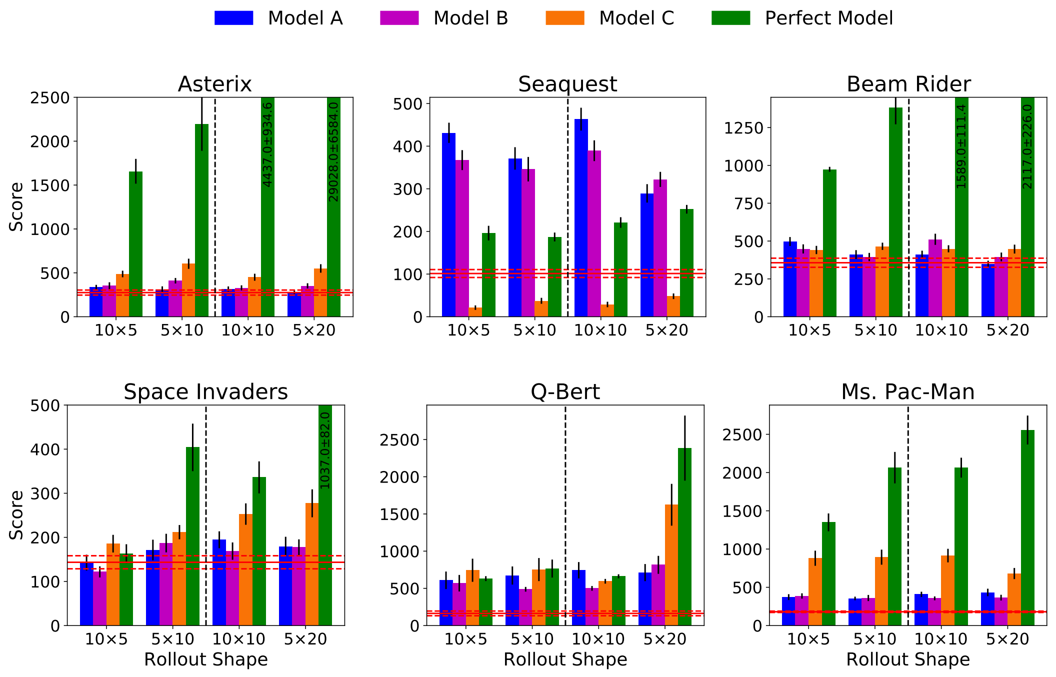


Figure B.1: The results of running 1-ply Monte-Carlo planning on six games from the ALE using both imperfect and perfect models. For each game and rollout shape there was a model that outperformed the random policy baseline (in red).

As a baseline, the result of the random policy — selecting actions uniform at random — is shown as a red horizontal line. The scores reported are the average of 30 independent episodes.

With the perfect model in each game, as the rollout depth is increased, the score tends to increase as well. In addition, there was at least one imperfect model that outperformed the random policy. This demonstrates that the models are able to learn useful information about the environment and that they are functional for planning.