Should Models Be Accurate?

by

Esraa M M Saleh

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Learning only by direct interaction with the world can be expensive in many real world applications. In such settings, Model-based Reinforcement Learning (MBRL) methods are a promising avenue towards data-efficiency. By planning with a model, a sequential decision making agent can decrease its reliance on direct interaction with the world. However, when the world is large, complex or seemingly changing, a learned model will be inevitably imperfect. Past work demonstrates that the effects of model imperfection can be difficult to avoid. In this thesis, we question the traditional objective of models that aims for accuracy in simulating the world. A model really only needs to be *useful*. Inspired by advances in meta-learning, we design a novel model learning loss. We show that a useful but inaccurate model can be learned with this loss so that it matches or surpasses accurate models in performance.

Preface

Work in this thesis builds upon my work in [SMK+22]. I presented this work as an Oral / Contributed Talk at The Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM) in 2022. I plan on submitting an expanded version of my thesis results for publication.

To Mockingjays

For showing me the art of courage and perseverance.

SAM: It's like in the great stories, Mr. Frodo. The ones that really mattered. Full of darkness and danger they were. And sometimes you didn't want to know the end. Because how could the end be happy? How could the world go back to the way it was when so much bad had happened? But in the end, it's only a passing thing, this shadow. Even darkness must pass. A new day will come. And when the sun shines it will shine out the clearer. Those were the stories that stayed with you. That meant something. Even if you were too small to understand why. But I think, Mr. Frodo, I do understand. I know now. Folk in those stories had lots of chances of turning back only they didn't. Because they were holding on to something.

FRODO: What are we holding on to, Sam?

SAM: That there's some good in this world, Mr. Frodo. And it's worth fighting for.

– J. R. R. Tolkien in The Lord of the Rings : The Two Towers

Acknowledgements

My life is characterized by a series of unique experiences with formal and informal mentors, who have played key roles in enriching my development in science. While I have focused on the earliest and most recent academic mentors here, there are many more in between. It has truly taken a village.

I would like to thank Mike (Michael Bowling), my supervisor, for his unwavering support and belief in me. People might notice the research itself, but to me, Mike's mentorship was always about building the researcher within. His commitment to mentorship in research and careers is embodied in his oft-repeated motto: "Once my student, always my student". Mike taught me invaluable lessons on the subtle art of developing a research taste, formulating problems and empirical design. We had extensive career discussions, where I always felt welcome to share my challenges and my triumphs. There were times in our meetings when I expressed how some of my long-held dreams seemed utterly out of reach, but Mike guided me patiently to get to the point where I could imagine them and the steps to reaching them. I look forward to building a future where I can be to others what he is to me.

When I was in high school in Ottawa, I had an amazing Math teacher, Mrs. Lida Chiarelli. She was a constant source of inspiration and support to me in a time of great turbulence in my life. She loved her students as they were, with all their imperfections. She appreciated our strengths and helped us iterate on our weaknesses. She not only encouraged questions outside the curriculum but would actually respond to our curiosity with new materials for us to follow up on. That style of curious thinking formed my foundation as a Computer Scientist today.

While I was working on projects within and around this thesis, I had the pleasure of collaborating with Anna Koop and John Martin. I'm grateful for the discussions I had with Anna, especially the ones early on in my Masters when I was writing this project's initial proposal and its first prediction algorithm. I really appreciate John's efforts, especially on our RLDM 2022 paper's second section as well as his thoughtful feedback on follow-up experimental designs. While I'm excited for the future, leaving the U of A is not something I have taken lightly. This was the first time where I felt like I really clicked with a high concentration of people. What I love about students here is that we really do lift each other up. RLAI, Amii and AI lab friends, you will always be special to me. I want to extend special thanks to Alex Kearney and Matthew Schlegel who have been a great source of feedback and advice on writing, speaking and careers.

Last but not least, I would like to sincerely thank my parents, family and friends who were there for me throughout this adventurous journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data efficiency is a characteristic of autonomous agents that we aspire to build. To learn, agents need to gather information by interacting with the world. Yet, these interactions are often expensive; they cost time and money. Giving agents the ability to build an internal model of how the world works is a promising avenue towards data efficiency, as shown in various settings in robotics and autonomous vehicles [WEH+23, IFKC16]. By having a model of the world, agents can decrease their reliance on interactions with the world. Instead, they can learn more internally by interacting with their model. However, building a world model is a non-trivial task in reality. If agents were to operate autonomously in our vast world, they cannot possibly observe the cause and effect of every event.

Intuitively, we recognize that that our own mental model of our environment is necessarily imperfect given the vastness of the universe and its intricate complexities. Yet, we still think with imperfect mental models because they can be *useful*. An example of this is in how Physicists sometimes operate on a model of objects in the world where the irregular is treated as spherical to simplify computations. As illustrated in Figure 1.1, this leads to funny but useful abstractions like modelling a cow as a spherical object to make computations on gravity or motion more feasible. If our agents *aim towards perfect models*, their models will be inevitably imperfect and the effects of imperfect models are difficult to mitigate. So the main question driving this thesis is: if we ignored model accuracy, how can we design an agent with an ability to learn

a *useful* world model when the world is large, complex or seemingly changing?

Figure 1.1: Spherical cow by Ingrid Kallick. "Assume a spherical cow" is not a far fetched phrase to hear in Physics. Spherical cows in Physics are examples of useful but imperfect models.

We adopt the Reinforcement Learning (RL) problem formalism to structure agent-environment interactions. We assume that these interactions happen over discrete time-steps. At every step, an agent observes the environment, constructs a perception of its state in the environment, takes an action in the environment and receives a reward from the environment. With trial and error, the agent learns a policy in order to maximize its expected *return*, a discounted sum of rewards. A *policy* specifies how likely it is for the agent to take an action from a specific state. In the context of RL, a *model* is typically thought of as a function that predicts transitions in the world. *Model-based Reinforcement Learning* (MBRL) refers to a set of RL algorithms that use a model to improve a policy or a value function. Using a model, an agent can interact or *plan* with its model to reduce the need for more expensive interactions with the environment. *Model-free Reinforcement Learning* refers to a set of RL algorithms that learn a policy directly from interactions without constructing a model. Given the potential for greater sample efficiency through model-supported policy or value function learning, we focus on MBRL.

Despite the demonstrated potential of models in planning, it remains a challenge to observe that potential in large, complex or seemingly changing environments [MBT+18]. We will often intentionally say "seemingly changing" just to highlight that, in our view, the world is what it is and it does not really change if one takes an omniscient view of it. When the world is too big or too complex for the agent to fully observe or when some important parts of the world are impossible to observe, the world can look like it is changing

from the agent's point of view. In such situations, the traditional model-learning objective of simulating an environment with perfect accuracy will inevitably result in model imperfections that can be detrimental to policy or value function learning. Using an imperfect model iteratively by composing its predictions can lead to compounding error [LPC22]. Prior work has attempted several avenues to tackle this problem. The first general idea is to plan with imperfect models more strategically. For example, an agent can estimate a model's predictive uncertainty and avoid planning in states where there is a significant risk of model imperfection [ASTW20]. The second general idea is to learn a model for planning in a compressed or latent space [HS18, HLF$^+$19], thus reducing the risk of imperfection as there is less to predict. The third general idea is to shift the objective of models altogether from being accurate in environment dynamics to being accurate in the corresponding expected returns. Models learned with such objectives are formalized as *value equivalent models* [GBSS20a, SAH$^+$20].

Advances in value equivalent models do show that a model can be highly performant and be inaccurate in simulating the world. Yet, these models are still grounded in real world observations and are focused on accuracy in returns. Our work is an exploration of pushing grounding to an extreme. Could there be an unexplored space of models that can produce useful experience if we removed any constraints on how model inputs or outputs should resemble reality?

In this thesis, we question the need for accuracy focused model learning objectives. In a complex or changing world, a perfectly accurate model is not practically achievable, is not necessary and it might not be the most useful model for policy or value function learning anyways. A model really only needs to be *useful*; it needs to aid in policy or value function learning. We explore these ideas in the context of Dyna-style planning [Sut91], where model-generated transitions are used to update a value function with the same learning rule used for updates with real transitions. We use a simple maze with changing reward placements in our investigations. In Chapter 3, we introduce formal model properties to clarify what model accuracy and inaccuracy mean

in the general context of a large, complex, or seemingly changing world. We disentangle model accuracy from other implicitly associated model properties that we define formally. We also provide motivating examples of handcrafted, inaccurate, but still useful models that can surpass handcrafted accurate models in performance. In Chapter 4, we introduce SynthDynaPrime to answer a *model existence question*: Does there exist *learnable* useful models whose learning objective does not constrain them to have any of the model-properties we defined? We design a novel model interface and a meta-learning objective that is focused on usefulness to learning. Assuming an idealized learning procedure with offline model training and privileged access to certain training data, we show that it is possible to learn a model that beats an accurate model. In Chapter 5 we introduce SynthDyna which relaxes assumptions from Chapter 4 by operating in a single experience stream and without privileged access to any information. We show that this algorithm can match an accurate model in aggregate performance. We conclude with a summary and discussion of future work in Chapter 6. Our contributions are twofold:

(1) *We motivate raising a fundamental question: should models be accurate?*

(2) *We present a novel model learning loss given by a meta-learning optimization. We show that a useful but inaccurate model can be learned with it so that it matches or surpasses accurate models in performance.*

Chapter 2

Background

In this chapter, we highlight the relevant foundations for exploring model-learning objectives in sequential decision making agents interacting with a complex or changing world. As humans, we can achieve a goal by interacting with the world to learn associations between actions and rewarding events. In Section 2.1, we provide some background on Reinforcement Learning (RL), a formulation for such learning problems. When we interact with the world, we often build a mental model of how the world works (e.g. how a car moves when we turn the steering wheel) and we act based on it. In Section 2.2, we highlight relevant Model-based Reinforcement Learning (MBRL) algorithms, where a sequential decision making agent learns a model of how the world works. In Section 2.3, we build connections to advances in meta-learning that can help us with our goal: creating model-learning objectives for MBRL with a focus on usefulness rather than accuracy in order to plan effectively in a complex or changing world.

2.1 Reinforcement Learning

Reinforcement Learning is a problem formulation for situations that require sequential decision making. In this formulation, an agent first observes the environment. Then, the agent takes an *action*. As a consequence, the agent receives a scalar *reward* and it observes the environmental change that its action induced. An agent's goal is to learn a *policy*, a way of behaving, in order to maximize expected cumulative discounted rewards.

Many RL problems can be thought of as Markov Decision Processes (MDPs). Let's define some MDP to be $M = \langle \mathcal{S}, \mathcal{A}, R, P \rangle$. Here, $\mathcal{S}$ is the set of possible *states*. In the case of a grid-world, a state might be the position of the agent in the grid. $\mathcal{A}$ is the set of possible *actions* that can be taken in the environment at any point in time. In a gridworld, an example of an action might be a vector encoding of "up", "down", "left", or "right". $P = P(s', r|s, a)$ is a true probability of transitioning to state $s' \in \mathcal{S}$ and getting a reward of $r \in \mathbb{R}$, given state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$. From now on, whenever we use a vector, like a state or action, we mean a column vector.

Since an agent cannot maximize an infinite sum of rewards into the future, a discount factor $\gamma \in [0, 1)$ is used to allow for the sum of rewards to remain finite. The higher the discount factor, the more weight is given to longer-term rewards. We call the discounted sum of rewards at an arbitrary time t, the *return*, and formalize it as:

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

A policy, $\pi : \mathcal{S} \to \mathcal{A}$, is a function that maps a state to an action. One simple example of a policy is an ϵ-greedy policy, where an agent can take either an exploratory random action with ϵ probability or a greedy action with probability $1 - \epsilon$. A greedy action in a particular state is the action that maximizes the estimated expected discounted sum of future rewards if the agent were to follow the same policy. This estimated expectation is often described using a state-action value function or Q-value function:

$$Q^{\pi}(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a, \pi]. \tag{2.2}$$

In this context, a greedy action is selected with:

$$\pi(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a). \tag{2.3}$$

Let $\prod$ be the set of all policies. When an agent has an optimal Q-value function $Q^*(s, a) = \underset{\pi \in \prod}{max} Q^{\pi}(s, a)$, we say that an agent has an optimal policy,

π^*, such that:

$$\pi(s) = \pi^*(s) = \arg\max_{\pi} Q^{\pi}(s, a) \tag{2.4}$$

One simple example of an RL algorithm that estimates $Q*$ and uses an ϵ-greedy policy is Q-learning [WD92]. This algorithm is an example of a *model-free* algorithm: it performs an incremental Q-value update based on direct interaction with the environment, which in turn improves an agent's policy. The Q-value update is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \arg\max_{a'} Q(s', a') - Q(s, a)) \tag{2.5}$$

Here, α is a step size dictating the magnitude of the update based on error on a single sample (s, a, r, s'). This update is a single sample update that bootstraps the value of $Q(s, a)$. Q is guaranteed to converge to $Q*$ under a slowly decaying sequence of step-sizes and as long as every state-action pair can always be visited. In the next subsection, we explore *model-based* algorithms which improve an agent's policy with interactions with a model of the world in addition to direct experience.

2.2 Model-based Reinforcement Learning

2.2.1 What is a model?

In the most basic sense, we think of a model as an internal agent artifact that supports policy or value function learning. Model-based RL algorithms are simply RL algorithms that use such a model. Traditionally, a "model" is thought of as a function, $\mathbf{m} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathbb{R}$, that provides a next state s' and a reward r, given a state s and an action a. $P(s', r|s, a)$ is regarded as the probability distribution from which the model draws samples of (s', r) given (s, a) [1].

We define models in a slightly broader sense than the traditional definition. Our definition requires us to introduce slightly different notions of states and

[1] In the literature, sometimes, the traditional definition of a model is broken down into two models: a dynamics model (for predicting a next state given a state and action) and a reward model (predicting a reward given a state). Sometimes, the dynamics model is referred to as the model.

actions because they can be helpful for describing models in the context of a large, complex, seemingly changing world. Considering the true underlying MDP of an RL problem, we call its set of states, $\mathcal{S}$, *environment states* and we call its set of actions, $\mathcal{A}$, *environment actions*. In the case of a grid-world with a non-stationary reward, an environment state will include the position of the agent *and* underlying variables dictating the non-stationarity. To capture the generality of models, we want to allow for them to be defined over a space of states and actions that may or may not reflect true environment states and actions. In addition, in a large, complex, seemingly changing world, the agent may never be able to infer the true environment state. So, we call the actions that an agent uses internally *agent actions*, and we denote them with the set $\hat{\mathcal{A}} \in \mathbb{R}^k$. We call the states that an agent uses internally *agent states*, and we denote them with the set $\hat{\mathcal{S}} \in \mathbb{R}^n$. From now on, we mean an agent state when we mention a state without mentioning whether it is an agent or environment state. Agent states can be states created by the model or states created using some function of the history, h_t, of all past observations, actions and rewards as in $h_t = o_0, a_0, r_1, o_1, a_1, r_2 ..., r_{t-1}, a_{t-1}, o_t$. The observations in the experience stream are the agent's raw experience of the world as it transitions through true environment states. To formally facilitate conversions between environment and agent versions of states and actions, we define two functions, $\psi_{\mathcal{S}} : \mathcal{S} \to \hat{\mathcal{S}}$ and $\psi_{\mathcal{A}} : \mathcal{A} \to \hat{\mathcal{A}}$. Considering a grid-world, an example of $\psi_{\mathcal{S}}$ can be a function that takes in a grid cell index and outputs a vector of bit features each giving an indicator of a wall in each of the cardinal directions. An example of $\psi_{\mathcal{A}}$ can be a function that takes in an action index (0 for up, 1 for right, 2 for down, or 3 for left) and returns a one-hot vector representation of the given index. The $\psi_{\mathcal{S}}$ and $\psi_{\mathcal{A}}$ functions are not assumed to be accessible to the agent. The agent does not always have access to the environment state outside the history of observations. Both $\psi_{\mathcal{S}}$ and $\psi_{\mathcal{A}}$ will be useful later in Chapter 3 as we motivate this thesis with simple but effective violations to formalizations of common assumptions on world models.

We formally define a **model** as an entity, $\mathbf{m}$, that provides a tuple, $(\hat{s}, \hat{a}, r, \hat{s}') \in \hat{\mathcal{S}} \times \hat{\mathcal{A}} \times \mathbb{R} \times \hat{\mathcal{S}}$ drawn according to a joint probability distribution:

$$\hat{P}(\hat{s}, \hat{a}, r, \hat{s}') = \hat{P}(\hat{s}, \hat{a})\hat{P}(\hat{s}', r|\hat{s}, \hat{a}) \tag{2.6}$$

Here, $\hat{P}(\hat{s}, \hat{a})$ is the probability of selecting an experience tuple $(\hat{s}, \hat{a}, r, \hat{s}')$ that has state $\hat{s}$ and action $\hat{a}$. The selection process with this probability is called *search control* in planning. We refer to $\hat{P}(\hat{s}', r|\hat{s}, \hat{a})$ as the *transition probability distribution*, which is analogous to the traditional definition of a model. Our definition combines the traditional notion of a model with search control. This will become important later as we tackle model-learning objectives with a focus on usefulness rather than accuracy. It allows us to talk about various MBRL algorithms using a unified language.

2.2.2 Background Planning with Dyna

Planning is the process in which an agent uses its model to support policy or value function learning. Planning methods generally fall under two categories: background planning and decision-time planning. Background planning is the act of using model-generated experience to gradually improve a policy or value function while interacting with the world [SB18]. Decision-time planning is the act of using model-generated experience in the process of selecting an action for the current state [SB18]. In this thesis, we focus on Dyna [Sut91], a background planning architecture.

The Dyna architecture interleaves planning and learning. Its main idea is in treating both model-generated and real experience in the same manner while updating a value function. A basic instantiation of the Dyna architecture is shown in Algorithm 1 as *All Experience Dyna*. Here in line 15, a non-parametric model, $\mathbf{m}$, samples uniformly at random a tuple of experience from a set of past real experience tuples $\mathcal{D}$. Note that if the number of planning steps is $k = 0$, then All Experience Dyna will just be Q-learning [WD92], the model-free algorithm mentioned earlier in Section 2.2.1. θ_t represents parameters of the state-action value function or Q-value function. The Q-learning update in lines 11 and 17 is an equivalent update to the one previously introduced in Equation 2.5 but we simply rewrite it here in a different form to allow for

linear value function approximation. This particular algorithmic instantiation of Dyna is presented here to provide an intuition for MBRL algorithms. Later in Section 3, we will use this as a template algorithm to introduce a variety of Dyna algorithms that provide insight on our research questions around what useful model experience can be.

Algorithm 1 All Experience Dyna

input: state feature transform f_s, action feature transform f_a
initialize: $\boldsymbol{\theta}_0 \in \mathbb{R}^{m \times n}$, $\mathcal{D} \leftarrow \{\}$, $\hat{\mathbf{s}}_0, \mathbf{a}_0, \mathbf{o}_0$
for $t = 1, 2, \cdots$ **do**
    Take an action $\mathbf{a}_{t-1}$ given by the ϵ-greedy policy, $\pi_{\boldsymbol{\theta}_{t-1}}(\hat{\mathbf{s}}_0) = \arg\max_{a \in \mathcal{A}} (\boldsymbol{\theta}_t \hat{\mathbf{s}}_{t-1})^T \hat{\mathbf{a}}_{t-1}$
    Observe $\mathbf{o}_t$ and r_t from the environment.
    $\hat{\mathbf{s}}_t \leftarrow f_s(\hat{\mathbf{s}}_{t-1}, \mathbf{a}_{t-1}, \mathbf{o}_t)$
    $\hat{\mathbf{a}}_{t-1} \leftarrow f_a(\mathbf{a}_{t-1})$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\hat{\mathbf{s}}_{t-1}, \hat{\mathbf{a}}_{t-1}, r_t, \hat{\mathbf{s}}_t)\}$

    // Update action-state value parameters, θ, with real experience.
    $\delta \leftarrow r_t + \gamma max(\boldsymbol{\theta}_t \hat{\mathbf{s}}_t) - (\boldsymbol{\theta}_t \hat{\mathbf{s}}_{t-1})^T \hat{\mathbf{a}}_{t-1}$
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \alpha \delta (\hat{\mathbf{a}}_{t-1} \otimes \hat{\mathbf{s}}_{t-1})$

    // Update action-state value parameters, θ, with internal experience.
    for $1, \cdots, k$ **do**
        $\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}' \sim \mathcal{D}$
        $\tilde{\delta} \leftarrow \tilde{r} + \gamma max(\boldsymbol{\theta}_t \tilde{\mathbf{s}}') - (\boldsymbol{\theta}_t \tilde{\mathbf{s}})^T \tilde{\mathbf{a}}$
        $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \beta \tilde{\delta} (\tilde{\mathbf{a}}_t \otimes \tilde{\mathbf{s}})$

2.2.3 The Landscape of Planning

In a complex or seemingly changing world, having a model that produces imperfect transitions is inevitable. Planning with such models can lead to adverse effects on policy learning. In this section, we reflect on past literature explaining and tackling this problem. To do so easily, when we say "model" in this section, we mean the traditional definition of a model as given in 2.2.1.

Compounding error can come from using a model iteratively [LPC22]. Given a state and action, a next state and reward are produced. Then, that next state is fed back into the model along with an action to produce another next state and reward, and so on. The resulting model-generated sequence of experience is a *rollout*. That last state and reward will have accumulated errors

associated with them. This is because they are a function of multiple nested operations, each with some result uncertainty. Another issue that can arise with imperfect models is the generation of states that do not correspond to any environment state; they are *hallucinated*. The Hallucinated Value Hypothesis posits that updates towards hallucinated state values can lead to misleading state-action value functions which in turn leads to deteriorating policy learning [JIT+20]. We are specifically interested in planning methods that can deal with the problems arising from inevitable model imperfection.

One line of research aims for adjusting the planning process to explicitly account for model imperfection. Long model rollouts can be beneficial as they are more likely to provide unfamiliar experience to learn from, but such rollouts can also compound model error. One idea to combat this problem is to limit the length and number of rollouts in planning [HTB18]. Another idea is to estimate a model's predictive uncertainty in order to avoid planning in states where the model might be critically inaccurate [ASTW20]. Rather than focusing exclusively on model outputs, another approach called *hallucinated replay*, trains the model on both model-generated experience and real experience so that the model can correct itself when it receives its own samples as input [Tal14].

Another line of research accounts for model imperfection implicitly by learning a policy or selecting actions exclusively based on interaction with a learned low-dimensional model. These models are commonly referred to as *latent dynamics models*. The common theme in this style of work is the ability to simulate the environment at low computational cost by predicting a compact latent next state given an action and a compact latent current state obtained by encoding an observation. The model is trained to produce latent states with the constraint that they must contain the necessary information to reconstruct original observations. Having compact latent states allows for focusing model capacity on more informative parts of the world. By predicting less, there is less of a chance for the model to make mistakes. *World Models* in [HS18] shows that a parametric policy can be learned entirely in a latent dynamics model. Then, the policy can be transferred into the agent interacting

with the real world. PlaNet [HLF+19], demonstrates that a latent dynamics model can be learned and used for an online planning process that selects an action for the agent's current state. Both World Models and PlaNet are examples from a line of research that emphasizes a need for model-generated latent states that are compact yet still represent environment states. We therefore say that the models in this line of work and the first line of work introduced before this are *aimed at accuracy* in simulating the world. However, do models really need to be aimed at accuracy or can other objectives be equally or more useful?

A third line of research also accounts for model imperfection implicitly but by shifting the learning objective of models from accuracy in simulating an environment to just ensuring that returns are accurate. A model that satisfies this objective is said to be *value equivalent* to a perfect model [GBSS20a]. By aligning the objective of a model with the objective of an agent (maximizing expected return), the model's limited capacity can be focused on parts of the world that are actually going to matter for the agent's goal of maximizing return. MuZero is one example of an algorithm that learns a value equivalent model [SAH+20]. Starting from a state that represents an environment state, MuZero applies its model iteratively to imagine the long-term expected cumulative reward of various actions if the agent were to move through different sequences of latent states. Based on that, the model can provide the agent with a reward-maximizing action to take from a current state. Value Iteration Networks (VINs) contain another example of a value equivalent model [TWT+16]. A VIN is a network representing a policy. This network is a special Convolutional Neural Network formulated to embed Value Iteration [SB18], a standard planning algorithm. Typically, value iteration is introduced in a context where a perfect model is given, but here, the model is entirely based on learnable parameters that are a subset of the VIN's parameters. Given that Value Iteration is expressed in a differentiable Neural Network architecture, it can be trained like any Neural Network policy in a model-free algorithm. As a consequence of the model parameters being a subset of the VIN parameters, the model's learning objective is that of the overall policy. MuZero and

VINs are demonstrations of algorithms with value equivalent models that are so useful to action selection or policy learning to the extent of beating competitive model-free algorithms and accuracy focused model-based algorithms in changing and/or complex environments.

This naturally leads us to question whether models should be aimed at accuracy in simulating the world. To what extent should any input or output of a model represent the environment? While MuZero's model is aimed at accuracy in returns without accuracy in transitions, MuZero still requires that planning starts at states representing environment states. VINs on the other hand, have their whole planning process (along with the model) embedded in a Neural Network blackbox. This makes their embedded model not quite amenable to examination so that we can look at the nature of the model-generated data influencing planning.

2.3 Meta-learning

When it is not possible to fully observe the world, it will look as though it is changing from an agent's point of view. Fast adaptation to change is a desirable agent trait in such a situation. Adapting quickly is a matter of learning how to learn quickly in a new context. Meta-learning refers to a wide range of methods that enable such a capability. Some of these methods, such as MAML, allow for quick adaptation in new situations after exposure to experience across multiple agent lifetimes [FAL17]. In each lifetime, an agent is initialized and exposed to one configuration of the environment (i.e. a task). MAML focuses on learning an initialization of a subset of agent parameters so that they can be learned quickly when the agent is exposed to a new task that shares some similarities with previously seen tasks. Sometimes, however, agent parameters need to be adapted online within a single agent lifetime as the agent encounters changes in the world. One early example of tackling this setting is IDBD, a method that incrementally adapts individual learning rates for each input feature of a linear supervised learning system [Sut92].

Partially inspired by IDBD, Reinforcement Learning with *meta-gradients*

is one way of adapting a subset of an algorithm's parameters, called *meta-parameters*, within a single agent lifetime [XvHS18]. Lets say we want to optimize a sequence of gradient updates that are applied to some agent parameters θ. These updates are a function of meta-parameters η. We define an outer loss, $L_{outer}(\tilde{\theta}(\eta))$, for optimizing η [2]. We define an inner loss $L_{inner}(\theta; \eta)$ for optimizing θ. Assume that we perform a sequence of updates to agent parameters θ, as shown below for an arbitrary i-th Stochastic Gradient Descent (SGD) update[3]:

$$\theta(\eta) = \theta_{i+1} = \theta_i - \alpha \nabla_\theta L_{inner}(\theta; \eta) \tag{2.7}$$

We call this update an *inner update*. To optimize a sequence of these inner updates, we can then take the gradient of the outer loss with respect to η because $\theta(\eta)$ is differentiable:

$$\eta_{j+1} = \eta_j - \beta \nabla_\eta L_{outer}(\theta(\eta_j)) \tag{2.8}$$

This general framework of meta-gradients can be applied to adapt components of model-free RL updates like the discount factor or the target [XvHS18, XvHH$^+$20].

We later build on an instantiation of meta-gradients called *bootstrapped meta-gradients*. In this instantiation, the outer loss gradient update to η is defined as:

$$\eta_{j+1} = \eta_j - \beta \nabla_\eta \mu \left(\theta(\eta_j)^{(k)}, [\![\theta'(\eta_j)]\!] \right) \tag{2.9}$$

Here, a matching function, μ, measures the dissimilarity between two versions of agent parameters $\theta(\eta)$. The first version, $\theta(\eta_j)^{(k)}$, is the result of applying k updates to the agent parameters θ as in equation 2.7. The second version, $\theta'(\eta_j)$, is the result of applying a fixed number of updates to $\theta(\eta_j)^{(k)}$ also using 2.7. Since $\theta'(\eta_j)$ is an agent parameter estimate based on another agent

[2] Notation for meta-gradients here is based on [ZXV$^+$20]

[3] This does not need to be an SGD update. It just needs to be a differentiable function. We are just using SGD for simplicity.

parameter estimate $\theta(\eta_j)^{(k)}$, we say that $\theta'(\eta_j)$ is *bootstrapped* from it. Formally, $\theta'(\eta_j)$ is called the *target bootstrap*. Bootstrapped meta-gradients are appealing because the target bootstrap lets us include future θ learning dynamics without the need to backpropagate through them (note how there is a stop gradient $[\![\cdot]\!]$ on $\theta'(\eta_j)$). In contrast, to include more θ learning dynamics, meta-gradients without bootstrapping require increasing the number of inner updates, k, which are then backpropagated through (note how $\theta(\eta_j)^{(k)}$ does not have a stop gradient applied on it). With bootstrapped meta-gradients, agents are able to achieve better performance than previous meta-gradient approaches when tested in complex environments or seemingly changing environments. We later explore bootstrapped meta-gradients as a primary ingredient for creating useful models that facilitate fast learning in a seemingly changing world.

Chapter 3

Inaccurate Models Can be Useful

"Model accuracy" is a concept that often comes with assumptions apparent in the context of its usage in language. Conventionally, the context of "model accuracy" assumes a stationary environment. It also assumes that accurate models are a productive target to aim for and by extension, inaccurate models are undesirable. However, we argue that inaccurate models can be desirable. We define model accuracy in a general manner that allows us to talk about modelling worlds, whether these worlds are stationary or not. We define new model properties that provide richer descriptions of models beyond accuracy. Developing these properties is important as it gives us a unified framework for describing informative algorithmic baselines used throughout this thesis. We then give an empirical demonstration of the common expectation model as a simple example of a useful inaccurate model. This highlights our intuition on why allowing for model inaccuracy can in fact be a productive design choice.

3.1 Model Properties Beyond Accuracy

The concept of "accurate models" can be an unclear. For example, an accurate model can mean a model that always produces a restricted set of all true transitions or it can mean a model that produces *all* true transitions in the world. This distinction is especially important in a complex, large or seemingly changing world as the latter can be hard to guarantee. In addition, describing

models only in terms of accuracy is insufficient for us to characterize and understand the wide variety of useful inaccurate models. While there has been some work in describing models beyond accuracy [GBSS20b], there is a lack of general descriptive properties that (a) can stand alone without assumed agent artifacts outside of models like value functions and (b) are general enough to describe models in a potentially changing world from an agent's perspective. In Definitions 3.1.1, 3.1.2 and 3.1.3, we introduce three properties of models to provide a clear a descriptive language while satisfying (a) and (b). A model can be *accurate*, *complete*, or *proper*.

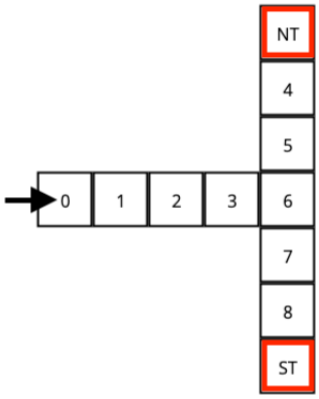


Figure 3.1: T-Maze environment. The arrow indicates the start state at 0. NT and ST indicate the North and South Terminal States.

Definition 3.1.1 *(Accurate Model) A model is accurate iff*
$\forall(s, a, r, s' \in \mathcal{S} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S})[\hat{P}(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a)) > 0 \implies$
$\hat{P}(\psi_{\mathcal{S}}(s'), r|\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a)) = P(s', r|s, a)]$

Definition 3.1.2 *(Complete Model) A model is complete iff*
$\forall((s, a) \in \mathcal{S} \times \mathcal{A})[\hat{P}(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a)) > 0]$

Definition 3.1.3 *(Proper Model) A model is proper iff*
$\forall(\hat{s}, \hat{a}, r, \hat{s}' \in \hat{\mathcal{S}} \times \hat{\mathcal{A}} \times \mathbb{R} \times \hat{\mathcal{S}})[\hat{P}(\hat{s}, \hat{a}, r, \hat{s}') > 0$
$\implies \exists(s, a, s' \in \mathcal{S} \times \mathcal{A} \times \mathcal{S})\ \psi_{\mathcal{S}}(s) = \hat{s},\ \psi_{\mathcal{A}}(a) = \hat{a},\ \psi_{\mathcal{S}}(s') = \hat{s}']$

In Definition 3.1.1, we describe an *accurate* model as a model with a transition probability that is equal to the true underlying true transition probability for any given $(\hat{s}, \hat{a})$ with a non-zero probability of being sampled in search control. If a $(\hat{s}, \hat{a})$ tuple has zero probability of being sampled, then it has no influence an agent's planning process. The model's transition probability can give the wrong probability on $(\hat{s}', r)$ given $(\hat{s}, \hat{a})$, but that does not matter and thus, we still regard the model as accurate. If a model's search control probability assigns non-zero probabilities to all $(\hat{s}, \hat{a})$ tuples that correspond to true environment (s, a) pairs, we call the model *complete* as in Definition 3.1.2. If a model's produced experience corresponds to some possible environment experience, we call the model *proper* as in Definition 3.1.3.

These properties are best described with concrete examples of model-based agents in a simple environment. Imagine a T-Maze environment as in Figure 3.1. An agent starts at state 0. It can take actions to move up, down, left or right. When at the North terminal state, an agent receives a reward of 1 and at the south terminal state, the agent receives a reward of 0. We call this arrangement of rewards the first reward regime. After a fixed number of episodes, N, the environment changes. Now, the agent receives a reward of 0 at the North Terminal (NT) state and a reward of 1 at the South Terminal (ST) state. We call this reward arrangement the second reward regime. For every N episodes, the environment switches cyclically between the first and second reward regimes. To make the tasks more difficult, at any time, the environment can move the agent one step in a random direction with probability ρ. If the agent moves into a wall, it simply goes back to the same state it was in before. An environment state in the TMaze can be thought of as a one-hot encoded vector of a cell's index concatenated along with a bit to indicate the current reward regime and an index in order to indicate the environment's current

	Accurate	Complete	Proper
Oracle Model Dyna	✓	✓	✓
Null Model Dyna	✓	✗	✗
All Experience Dyna	✗	✓	✓
NoPTS Stable Experience Dyna	✓	✗	✓
NoTNS Stable Experience Dyna	✗	✓	✓
NoTNS Expectation Model Dyna	✗	✓	✗
NoPTS Expectation Model Dyna	✗	✗	✗
SynthDyna	✗	✗	✗

Table 3.1: The manifestation of model properties in different agents

episode (from which a reward switch can be inferred). An agent state in the TMaze is just a one-hot encoded vector of a cell's index. So here, $\psi_S : \mathcal{S} \to \hat{\mathcal{S}}$ is a an extractor of the one-hot encoded vector in an environment state. Agent action and environment actions are the same which means that $\psi_A : \mathcal{A} \to \hat{\mathcal{A}}$ is the identity function.

Considering the three model properties we introduced previously (Definition 3.1.1, 3.1.2, 3.1.3) and the TMaze environment described, we now have a concrete context through which we can understand Dyna agents with different models. We provide a summary of example agents and the properties that they satisfy in Table 3.1. Before going into why these agents have various property combinations, we need to understand how their models are constructed. Lets assume a Q-learning update for our Dyna agent instantiations and an ϵ-greedy policy. **Oracle Model Dyna** cheats; it has access to the true environment transition probability function at every point in time. Its search control probability distribution, $\hat{P}(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a))$, is a distribution that assigns a non-zero probability to every possible $(s, a) \in \mathcal{S} \times \mathcal{A}$. This distribution can, for example, be a uniform distribution. The extreme opposite to Oracle Model Dyna is **Null Model Dyna**. It has a model that can only produce zero vectors for $\hat{s}$, $\hat{a}$, and $\hat{s}'$. It is equivalent to model-free Q-learning.

Is there a middle ground between the extremes of having a perfect model and having effectively no model at all? Consider the TMaze in Figure 3.1. As in All Experience Dyna in Algorithm 1 in Chapter 2, a simple strategy can be to gather any experience tuple, $(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a), r, \psi_{\mathcal{S}}(s'))$, encountered through

interaction with the environment and sample uniformly from these tuples in planning. If a model is simply a set of experiences that the agent encounters, then the model's search control probability distribution is the uniform random distribution over the agent's set of experience, which will eventually cover all possible experiences. Given that rewards are switching between two reward regimes at terminals (NT and ST) in the TMaze, this agent's model is bound to gather experience tuples that are identical in agent states, actions and next states, yet different in rewards. In planning, the agent will sometimes sample terminating experience tuples with rewards reflecting a reward regime that the agent is not currently in, which will be misleading for the learning process, as we will demonstrate empirically later in Section 3.2.

How can we have a simple model that just has experience tuples but that avoids misleading the learning process? One idea is a model constructed with gathered interaction experience that excludes any experience where a state vector represents a pre-terminal state (PTS). In the TMaze, we call states 4 and 8 pre-terminal states. Eliminating any experience tuple with a PTS starting state prevents any learning updates on a PTS. This removes the possibility of sampling experiences with rewards that are not from the agent's current reward regime. Experience will look the same or be *stable* across reward regimes. We call an agent with such a strategy **NoPTS Stable Experience Dyna**. Since changing rewards can only be experienced in terminating transitions in the TMaze, an alternative experience elimination strategy can be to only filter out experience tuples with a terminating next state (TNS). We call an agent with this strategy **NoTNS Stable Experience Dyna**. Both, NoPTS Stable Experience Dyna and NoTNS Stable Experience Dyna are exploiting privileged knowledge of the TMaze domain, although not the unobservable knowledge indicating anything about upcoming reward regimes. The only non-stationarity not represented by these agents are rewards on termination.

All of the models that we have introduced so far, produce experience tuples with agent states that can be mapped back to actual environment states. Is there a simple agent that can plan well with states that do not map back to an environment state? This is part of our core goal in this thesis precisely

because we want to understand how model usefulness manifests without the need to simulate an environment's transitions. Expectation models are one answer [SS10]. In Stable Experience's model, a state is represented as the agent state is defined: a one-hot encoded vector representation of a TMaze cell. This representation is also part of the environment state. An **Expectation model**, as originally defined, is a model that gives an experience tuple as usual but the next state vector is an expectation of next state vectors that occur with that experience tuple's state and action. To handle an environment with changing rewards in terminal states, we can derive expectation model agents with strategies like NoPTS and NoTNS, which we use in Stable Experience agents. **NoPTS Expectation Model Dyna** is an agent that gathers experience tuples except for tuples that have pre-terminal states as starting states. Before putting an experience tuple in its set of experiences, it transforms its next state vector into an expectation next state vector, which makes its set of experience an expectation model. Similarly, NoNTS Expectation Model Dyna is an agent that gathers experience tuples except for tuples that have terminal states as next states. It does exactly the same transformation on next state vectors that **NoPTS Expectation Model Dyna** does. SynthDyna is an agent with a learned model that we will be introducing later in this thesis as our main contribution. In contrast to our Expectation Model and Stable Experience agents, SynthDyna does not assume knowledge of non-stationarities in the world, in advance.

The three model properties introduced previously are best understood in the context of the example agents we have provided so far. Oracle Model Dyna, Null Model Dyna, and NoPTS Stable Experience Dyna have accurate models satisfying Definition 3.1.1. An Oracle Model has an omniscient view of the environment's dynamics. It is accurate because it has access to the true environment transition probability function at every point in time. A Null Model does not even hold a representation of environment dynamics. Intuitively, it is accurate as it it claims to know nothing, and by extension, it does not claim to know something about the world's dynamics that is actually

incorrect [1]. The Null model is accurate since $\hat{P}(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a)) = 0$ for every environment state and action, (s, a). This is a consequence of the search control probability being $\hat{P}(\hat{s}, \hat{a}) = 1$ when $\hat{s} \in \mathcal{S}$ and $\hat{a} \in \hat{A}$ are zero vectors. NoPTS Stable Experience is an accurate model because it only gathers experience tuples with no pre-terminal starting states, and thus, the distribution of gathered tuples will always be the same regardless of the reward regime. In contrast, NoTNS Stable Experience is not accurate because it gathers experience while only eliminating terminating transitions. This makes the model's transition probability not equal to the true environment transition probability given pre-terminal states and their associated actions. Recall that the TMaze environment has some stochasticity. With some (usually small) probability, an action at a pre-terminal state that usually leads to a terminal state might be associated with a non-terminating next state in some experience tuples. So, with the elimination of terminating transitions, the agent transition probability increases for a non-terminating next state and a non-terminating reward, given a pre-terminal state and the action that usually leads to a terminating next state. Similarly, All Experience Dyna is inaccurate because mixing experiences from the TMaze's two reward regimes means that the agent transition probability is not equal to the true transition probability for any next state and reward given any pre-terminal state and action. Models in NoTNS Expectation Model Dyna and NoPTS Expectation Model Dyna are all inaccurate for the same reason; sampled experience tuples from them do not necessarily have agent states or next states that map back to environment states through $\psi_{\mathcal{S}}$.

Oracle Model Dyna, All Experience Dyna, NoTNS Stable Experience Dyna, and NoTNS Expectation Model Dyna have complete models satisfying Definition 3.1.2. Assume that these agents were to be run forever such that their models will eventually reach a point where every possible agent state-action pair is present. The mapping from these agent state-action pairs to environment state-action pairs is surjective. For every environment state-action pair (s, a), it is possible for these agents to sample an experience tuple with

[1] Assume that $\psi_{\mathcal{S}}$ and $\psi_{\mathcal{A}}$ will never map an environment state or action to a zero vector

$\psi_{\mathcal{S}}(s)$ and action $\psi_{\mathcal{A}}(a)$. Null Model Dyna, NoPTS Stable Experience Dyna and NoPTS Expectation Model Dyna are all incomplete models. Null Model Dyna does not have *any* experience tuples with agent states and actions that map back to environment states and actions, so it is impossible to sample such experience tuples. NoPTS Stable Experience Dyna and NoPTS Expectation Model Dyna both gather experience tuples yet eliminate experience tuples containing pre-terminal states as the starting state. So, it becomes impossible to sample those states.

Oracle Model Dyna, All Experience Dyna, NoPTS Stable Experience Dyna, and NoTNS Stable Experience Dyna have proper models satisfying Definition 3.1.3. Recall that in the TMaze, we define $\psi_{\mathcal{S}}$ to be a one-hot encoding extractor of an environment state's one-hot encoded cell index. We also define $\psi_{\mathcal{A}}$ to be a function that takes in environment action indicating a cardinal direction with an index (e.g. 0,1,2,3) and returns a one-hot encoded vector of that index. These agents have proper models because their models can only produce an experience tuple, $(\psi_{\mathcal{S}}(s), \psi_{\mathcal{A}}(a), r, \psi_{\mathcal{S}}(s'))$, with agent states and actions that map back to environment states and actions. In contrast, Null Model Dyna, NoTNS Expectation Model Dyna and NoTNS Expectation Model Dyna have improper models. By definition, the Null Model can only produce a zero vector state and action, and they do not map back to any environment state and action. Expectation Models produce expected next state vectors, which do not map back to any environment state vectors in a stochastic environment like the TMaze.

Through examples of agents in a stochastic and changing TMaze environment, we demonstrate how a model can be accurate, complete or proper, yet we are left with our core questions on the utility of model accuracy.

3.2 Empirical Evidence

Can inaccurate models beat accurate models in performance? To answer this question, we can compare agents with accurate models to agents with inaccurate models as they are labelled in Table 3.1. We will not be looking at Oracle

Model Dyna, since it is impossible to discover without assumptions on when the TMaze's terminals will switch into a new reward regime. We run all other agents in the TMaze environment for 30000 episodes. We evaluate agents on the last 1200 episodes, which we call the *evaluation period*. The environment switches between its two reward regimes every $N = 600$ episodes and it has a stochasticity degree of $\rho = 0.15$. We generally follow the result reporting recommendations from [ASC+21]. We use 30 samples per statistic and %95 (i.e. $\alpha = 0.05$) percentile bootstrapped confidence intervals with 100K repetitions. Such confidence intervals are constructed by first re-sampling the samples we have (with replacement) for 100K times. Then, we record the statistic of interest for each of the 100K re-sampled groups. After that, we can construct the confidence interval with the lower limit being the $\alpha/2$ quantile of the 100K statistics and the upper limit being the $1-\alpha/2$ quantile of the 100K statistics.

Figure 3.2 and Figure 3.3, demonstrate the Mean Squared Return Error across the episodes happening around the last two reward regime switches where evaluation period happens. The Return Error is the difference between the optimal discounted return and the actual discounted return in an episode. Similarly, Figure 3.4 and Figure 3.5, demonstrate the Inter-quartile Mean (IQM) Squared Return Error across the same set of episodes. The Interquartile Mean is useful for observing mean performance without extreme values. In Figure 3.6, we highlight aggregate agent performance with Mean Squared Return Errors (MSRE) over the evaluation period. We show the median, IQM and the mean of this performance metric across runs. An error here is the difference between the optimal discounted return and an agent's actual discounted return. We use grid search and choose hyperparameters according to the average MSRE in the last 1200 episodes. After selecting hyperparameters for any agent based on 30 runs per hyperparameter configuration, we always report results on 30 independent runs.

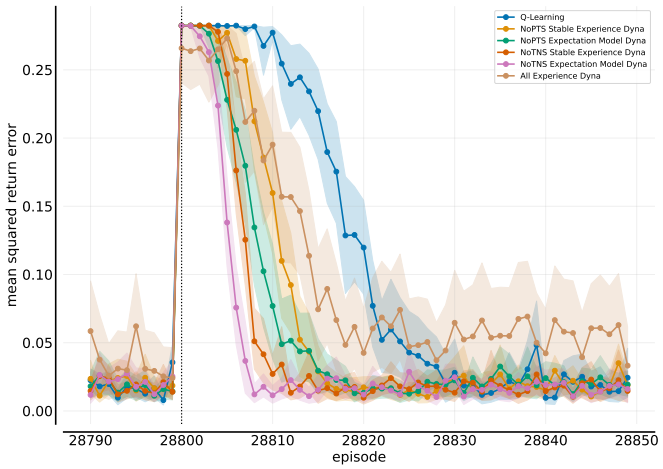


Figure 3.2: Around the first reward switch period in evaluation, we observe the Mean Squared Return Error across episodes, per baseline agent. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

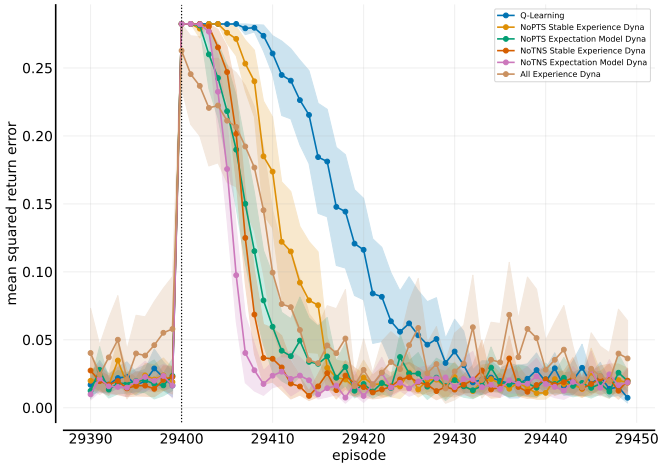


Figure 3.3: Around the second reward switch period in evaluation, we observe the Mean Squared Return Error across episodes, per baseline agent. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

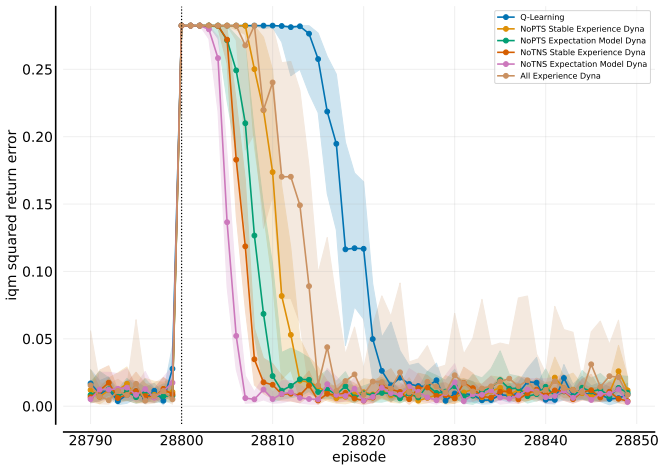


Figure 3.4: Around the first reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error across episodes, per baseline agent. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

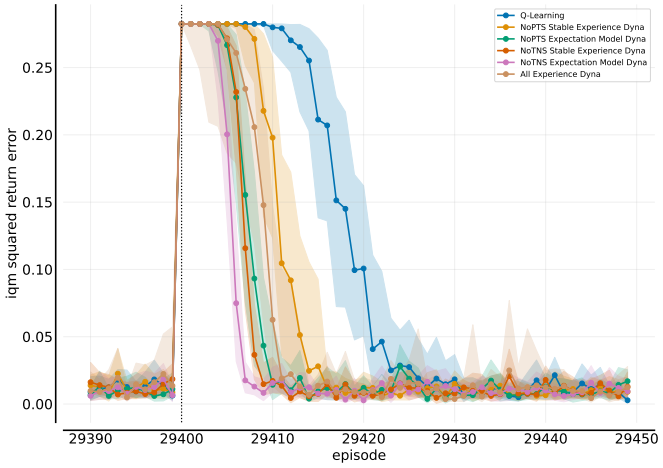


Figure 3.5: Around the second reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error across episodes, per baseline agent. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

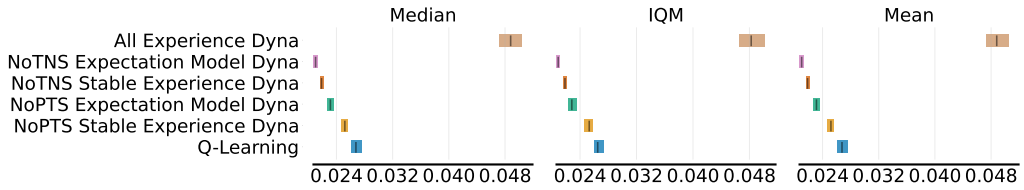


Figure 3.6: Evaluation performance per baseline agent summarized by the Mean Squared Return Error. We show the Median, Inter-Quartile Mean (IQM), and Mean of the performance metric over 30 runs. Intervals around performance point estimates are %95 percentile bootstrap confidence intervals constructed with 100K repetitions.

First of all, we clearly see from All Experience Dyna that when a model-based agent operates in the world without accounting for change, its model will be inevitably inaccurate. All Experience Dyna has large fluctuations in errors most clearly seen in Figures 3.2 and 3.3. In fact, sampling experience that is irrelevant for the current reward regime can be so harmful to the extent that Null Model Dyna, an effectively model-free agent, can beat All Experience Dyna in aggregate performance as seen in Figure 3.6. So model inaccuracy can be detrimental. Second, as seen in Figure 3.6, *inaccurate models can beat accurate models in aggregate performance*. NoTNS Stable Experience, NoTNS Expectation Model and NoTNS Expectation Model are all inaccurate models that perform better than an accurate model, NoPTS Stable Experience. Third, NoPTS Expectation Model Dyna in Figure 3.6, indicates that a model can be inaccurate, incomplete and improper, yet still beat an accurate model in aggregate performance. In this case, the accurate model is the model of NoPTS Stable Experience. Given that there exists handcrafted inaccurate models that can perform better than accurate models, this suggests that in the absence of knowledge on the scheduled nature of non-stationarity in the world, model accuracy is not necessarily the best objective for model learning. In the next chapter, we explore an alternative model learning objective with a focus on usefulness to value function learning without imposing accuracy as a goal and without restricting the space of models to complete or proper models.

Chapter 4

Performance Potential of Useful Models

Handcrafted expectation models show us how a model can be inaccurate yet more useful than an accurate handcrafted model. In addition, they show us that a model can be incomplete and improper yet still be useful. What still remains is this *model existence question*: Does there exist *learnable* useful models whose learning objective does not constrain them to be accurate, complete or proper? Can these models beat accurate models? In Section 5.1, we assume an idealized learning algorithm and we describe our attempt at designing a model-learning loss function to answer this existence question. In Section 5.2, we discuss empirical results for that algorithm.

4.1 SynthDynaPrime: Offline Model Training and Perfect Loss Targets

As mentioned in Section 2.3, recent meta-learning research highlights its potential to speed up learning in situations similar to ones seen in the past. When learning is sped up in a certain situation, we simply mean that policy learning happens faster than if no meta-learning took place beforehand. This means that the agent *adapts its behaviour* relatively quickly. For example, let us reconsider our TMaze in Figure 3.1 from Section 3.1. An agent is exposed to two situations manifested in the two reward regimes of the TMaze. If an ideal agent were to navigate this maze, we would expect it to understand that

terminal rewards are not going to be the same for all time. The agent would need to understand that its always productive to walk along the initial hallway of the maze but that the decision point of going north or east at the of the hallway is important to get right in order to maximize reward. As soon as the agent is exposed to a reward change on a certain terminal state, it needs to update their beliefs quickly to explore again and discover that the other terminal is the rewarding one now. Of course, in an environment as simple as this, the agent could potentially avoid exploring again by simply memorizing the pattern of reward switches. However, recall that we are assuming the agent has no knowledge on the consistent nature of the non-stationarity in the world, as we are not trying to design algorithms that are too specific to a certain pattern.

How can we induce similar behaviour in agents? One approach taken in the "meta-gradient" style of RL is to meta-learn more efficient update functions [XvHS18, XvHH$^+$20, FSZ$^+$22]. For example, an update rule can be completely parametrized or partially parametrized by components like its discount factor or its target. Then, parametrized quantities can be learned by taking a gradient through a meta-loss. As stated in our model existence question, we are interested in understanding whether useful models can be learned with a learning objective that does not constrain them to be accurate, complete or proper. In [SRL$^+$20], Generative Teaching Networks are introduced as a method for meta-learning training data that can speed up learning. So, one promising idea could be to parameterize data (i.e. experience) in planning updates and meta-learn a model to generate that data.

Algorithm 2 $\mathcal{L}$: SynthDynaPrime Meta Loss

input: $\boldsymbol{\theta} \in \mathbb{R}^{m \times n}$, $\boldsymbol{\theta}_{true}$
for $1, \cdots, k$ **do**
    $\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}' \sim m(\boldsymbol{\eta})$
    $\tilde{\delta} \leftarrow \tilde{r} + \gamma max(\boldsymbol{\theta}\tilde{\mathbf{s}}') - (\boldsymbol{\theta}\tilde{\mathbf{s}})^T\tilde{\mathbf{a}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \zeta\tilde{\delta}(\tilde{\mathbf{a}} \otimes \tilde{\mathbf{s}})$
return: $||\boldsymbol{\theta} - \boldsymbol{\theta}_{true}||_2^2$

In Algorithm 2, we introduce a metaloss for training a generative model

m with meta-parameters η. We allow this model to generate a whole tuple of experience $(\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}')$. In order to make the objective of this model aligned with the objective of the agent, we require that planning with the model (lines 2 to 5) from arbitrary value function parameters θ results in a value function closer to the target true value function θ_{true} (line 6), which in turn, should result in policy improvement. Choosing some proxy for a target value function can be tricky especially in a seemingly changing world. So for now in line 6, we assume that the target is given as the true value function (parameters given by θ_{true}) in order to explore the potential of models learned with the metaloss in Algorithm 2. This metaloss is somewhat related in form to the bootstrapped meta-gradient loss introduced in Chapter 2.3. Here, the outer loss is given by the matching function, $\mu(\mathbf{a}, \mathbf{b}) = ||\mathbf{a} - \mathbf{b}||_2^2$, which was previously studied in the context of Bootstrapped Meta-learning [FSZ+22]. However, bootstrapping the target is completely bypassed by our assumption that the target is the true value function. With these choices on the model's interface and the meta-loss, we are effectively giving our model the ability to generate any experience tuple as long as it is useful for value function learning. The learned model is allowed to be inaccurate, incomplete and improper.

SynthDynaPrime in Algorithm 3 is an idealization of the algorithm we aim to develop as it is specifically designed to allow us to focus on developing the loss function and answer our model existence question. In order to focus on developing a loss function, we train SynthDynaPrime's model offline. This eliminates the possibility of over-fitting on early data or training on non-iid data. We also make several simplifying assumptions that relate to the environment: (1) we have the ability to start an agent at a particular task configuration, (2) we have access to the environment dynamics which allows us to compute the true value function per task configuration, and (3) we have the ability to restart an agent arbitrarily many times in the environment.

In lines 4 to 6, SynthDynaPrime starts by computing the true value function per environment dynamics configuration in set $\mathcal{C}$. An environment configuration in our case is going to be a different placement of rewards in the environment. In lines 9 to 16, starting τ times from each environment config-

uration, we run model-free Q-learning (Null Model Dyna) to gather starting $\hat{\boldsymbol{\theta}}_t$ examples along with the corresponding true value function parameters indexed by the current environment configuration $\bar{c}$ as in $\Theta_{true}[\bar{c}]$. Tuples of ($\hat{\boldsymbol{\theta}}_t$, $\Theta_{true}[\bar{c}]$) are gathered in the collection $\hat{\Theta}$. Note that although gathering $\hat{\boldsymbol{\theta}}_t$ examples starts from different environment configurations, these configurations can change as the agent interacts with the environment, hence the need to index the true value function parameters by the current environment configuration. In lines 19 to 23, we use the data gathered in $\hat{\Theta}$ to train a generative model $m(\eta)$ using the SynthDynaPrime metaloss $\mathcal{L}$ in Algorithm 2 (in implementation, we hold out a validation data-set and use early stopping to avoid over-fitting). In lines 26 to 39, we run an agent with Dyna-style planning. The agent uses the learned model $m(\eta)$ in planning which updates the actual value function θ_t that parameterizes the agent's policy $\pi_{\theta_{t-1}}$.

Algorithm 3 SynthDynaPrime

input: feature transform f, world dynamics configurations $\mathcal{C}$
initialize: $\boldsymbol{\theta}_0 \in \mathbb{R}^{m \times n}$, $\boldsymbol{\eta}$, $\Theta_{true} = dict()$, $\hat{\Theta} = []$ $\hat{\mathbf{s}}_0 \leftarrow f(\mathbf{h}_0)$
// Run Value Iteration for each change period, assuming knowledge of world dynamics
for every environment configuration of dynamics in $\mathcal{C}$, indexed by c **do**
    $\boldsymbol{\theta}_{true} \leftarrow$ value_iteration(c, $\mathcal{C}$, γ)
    $\Theta_{true}[c] = \boldsymbol{\theta}_{true}$

// Gather $\boldsymbol{\theta}$ data from model free Q-learning (Null Model Dyna)
for every environment dynamics configuration in $\mathcal{C}$, indexed by c **do**
    for $1, \cdots, \tau$ **do** // Generate multiple trajectories Initialize $\hat{\boldsymbol{\theta}_0} \in \mathbb{R}^{m \times n}$
        set_world_starting_config(c)
        for $t = 1, 2, \cdots$ **do**
            $\delta \leftarrow r_t + \gamma max(\hat{\boldsymbol{\theta}}_t \hat{\mathbf{s}}_t) - (\hat{\boldsymbol{\theta}}_t \hat{\mathbf{s}}_{t-1})^T \hat{\mathbf{a}}_{t-1}$
            $\hat{\boldsymbol{\theta}}_t \leftarrow \hat{\boldsymbol{\theta}}_t + \alpha \delta (\hat{\mathbf{a}}_{t-1} \otimes \hat{\mathbf{s}}_{t-1})$
            $\bar{c} \leftarrow$ get_current_world_config()
            $\hat{\Theta}$.append($(\hat{\boldsymbol{\theta}}_t, \Theta_{true}[\bar{c}])$)

// Train a model m for e epochs based on gathered data and loss $\mathcal{L}$ from Algorithm 2
for $1, \cdots, e$ **do**
    for each minibatch-sized part of $\hat{\Theta}$ **do**
        // Update SynthDynaPrime model with metaloss $\mathcal{L}$ (Alg 2).
        Sample minibatch $\mathcal{B}$ from $\hat{\Theta}$.
        $\boldsymbol{\eta} \leftarrow$ Optimizer($\boldsymbol{\eta}, \mathcal{B}, \mathcal{L}$)

// SynthDynaPrime's main loop
for $t = 1, 2, \cdots$ **do**
    Take an action $\hat{\mathbf{a}}_{t-1}$ given by the ϵ-greedy policy, $\pi_{\boldsymbol{\theta}_{t-1}}$
    Observe $\mathbf{o}_t$ and r_t from the environment.
    $\hat{\mathbf{s}}_t \leftarrow f(\mathbf{h}_t)$

    // Update action-state value parameters, θ, with veridical experience.
    $\delta \leftarrow r_t + \gamma max(\boldsymbol{\theta}_t \mathbf{s}_t) - (\boldsymbol{\theta}_t \hat{\mathbf{s}}_{t-1})^T \hat{\mathbf{a}}_{t-1}$
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \alpha \delta (\hat{\mathbf{a}}_{t-1} \otimes \hat{\mathbf{s}}_{t-1})$

    // Update action-state value parameters, θ, with internal experience.
    for $1, \cdots, k$ **do**
        $\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}' \sim m(\boldsymbol{\eta})$
        $\tilde{\delta} \leftarrow \tilde{r} + \gamma max(\boldsymbol{\theta}_t \tilde{\mathbf{s}_t}) - (\boldsymbol{\theta}_t \tilde{\mathbf{s}}_{t-1})^T \tilde{\mathbf{a}}$
        $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \beta \tilde{\delta} (\tilde{\mathbf{a}}_t \otimes \tilde{\mathbf{s}}_{t-1})$

4.2 Empirical Results

We test SynthDynaPrime (Algorithm 3) on the TMaze environment introduced in Section 3.1. We use the same environment settings, evaluation methodology, and hyperparameter selection strategy in Section 3.2. One key difference is that we select hyperparameters for SynthDynaPrime based on running it with a fixed model for 3600 episodes (instead of 30000) and evaluating it on the last 1200 episodes. This is just to facilitate more searching over hyperparameters since performance does not differ too much at 30000 episodes when the model is fixed. After hyperparameter selection, we report the final performance by running SynthDynaPrime for 30000 episodes with a fixed model. We use 3600 episodes for each run of the data gathering Q-learning algorithm in SynthDynaPrime. We build a model as a fully connected 2-layer Neural Network with ReLU activations. The model takes in a Gaussian noise vector and gives an experience tuple to be used in planning. We train the model with an Adam optimizer.

In Figures 4.1 and 4.2, we examine the performance of SynthDynaPrime (along with baselines defined in Section 3.1) in terms of the Mean Squared return Error at every episode near the last two reward regime switches. In Figures 4.3 and 4.4, we repeat the same plots but with the Inter-quartile Mean Squared Return Error as the performance metric, to look at performance when measurements in the top and lower quartiles are eliminated. These plots indicate that SynthDynaPrime can facilitate adaptation to changes in the environment. It is faster in adaptation than Q-learning and unlike All Experience Dyna, it can roughly settle at a low error. There is some variability in its performance as highlighted in the difference between Mean Squared Return Error plots in comparison with Inter-quartile Mean Squared Return Error plots. We think this can be improved with more data coverage for gathered θ data. SynthDynaPrime's relationship to other handcrafted (i.e. cheating) model-based algorithms is less clear from these plots, so we examine this in terms of aggregate performance plots.

In Figure 4.5, we summarize performance by first computing the Mean

Squared Return Error per run over the last 1200 episodes, and then compute an aggregate metric over runs. The last 1200 episodes cover the last two reward switches in the TMaze. SynthDynaPrime yields an aggregate Mean Squared Return Error of 0.02350 with 95% bootstrapped percentile confidence interval of (0.02304, 0.02396). NoPTS Stable Experience Dyna, the model-based agent with an accurate model from Chapter 3, yields a Mean Squared Return Error of 0.02520 with a 95% bootstrapped percentile confidence interval of (0.02472, 0.02566). Recall that NoPTS Stable Experience Dyna is an agent whose model consists of a set of every experience tuple encountered except for experiences where the starting state is a pre-terminal state. This allows NoPTS Stable Experience Dyna to avoid updating pre-terminal states in planning since these are the states that can occur before a changing final reward is reached. Therefore, in terms of aggregate performance, *SynthDynaPrime can learn an inaccurate model that beats an accurate model that is informed of the non-stationarity in the environment.* We can also see this when considering other metrics that are robust to skewed distributions like the IQM and the median. These results also show that *there exists such a useful model that can be reached with a learning objective that makes no restrictions on a model being accurate, complete or proper.*

In Appendix A, we include examples of generated experiences from the SynthDynaPrime model and contrast them to examples of experiences that can be seen in AllExperienceDyna's model and StableExperience models. The fact that synthetic transitions do not correspond to any transitions that can happen in the TMaze world means that the generative model of SynthDynaPrime is improper. It is not possible for us to formally claim with complete certainty that the model is in fact incomplete or inaccurate. This is because the model is a generative model with continuous outputs representing full experience tuples (state, action, reward, next state). It is not possible to examine every possible tuple to check for transitions with elements that map to real transitions, given the continuous nature of this model's output. Thus, it becomes impossible to say with complete certainty if this model is inaccurate or incomplete. However, no matter how much we tried in our attempts to generate experiences, we

never obtained any experience tuples with any elements that could potentially be even close to real experiences.

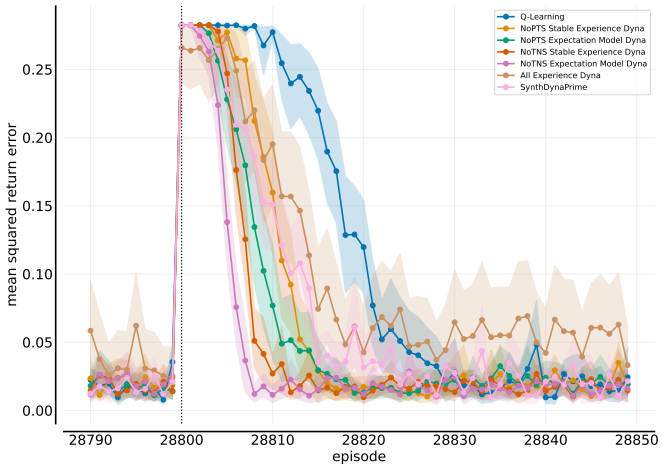


Figure 4.1: Around the first reward switch period in evaluation, we observe the Mean Squared Return Error, per agent, across episodes, in comparison to SynthDynaPrime. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

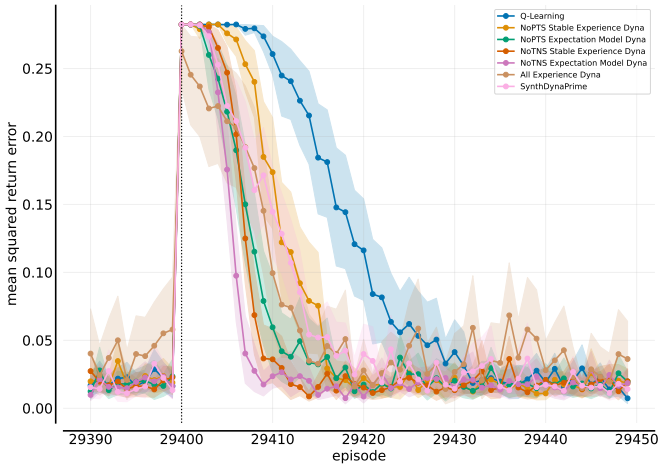


Figure 4.2: Around the second reward switch period in evaluation, we observe the Mean Squared Return Error, per agent, across episodes, in comparison to SynthDynaPrime. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

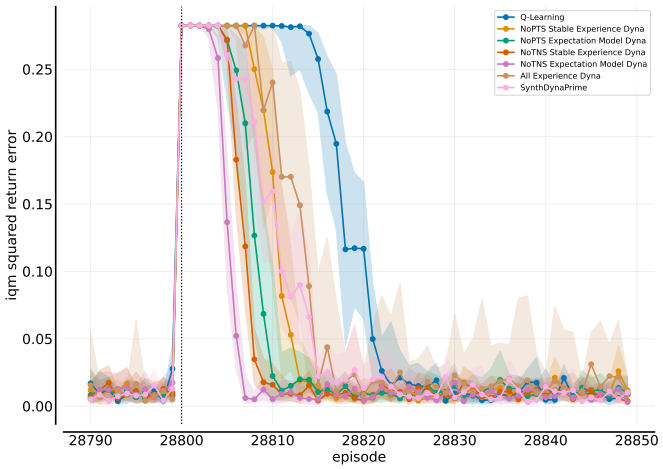


Figure 4.3: Around the first reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error, per agent, across episodes, in comparison to SynthDynaPrime. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

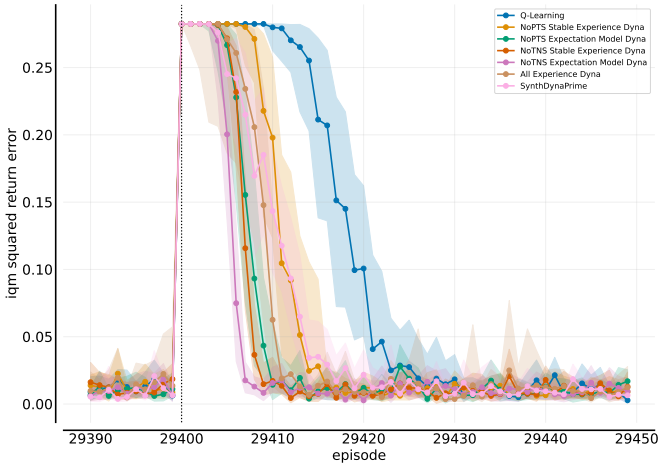


Figure 4.4: Around the second reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error, per agent, across episodes, in comparison to SynthDynaPrime. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

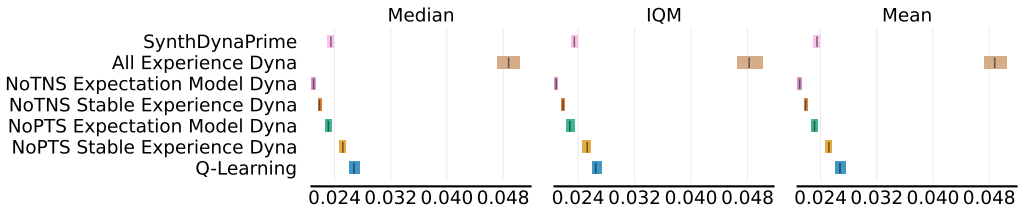


Figure 4.5: Evaluation performance per agent in comparison with SynthDynaPrime, summarized by the Mean Squared Return Error. We show the Median, Inter-Quartile Mean (IQM), and Mean of the performance metric over 30 runs. Intervals around performance point estimates are %95 percentile bootstrap confidence intervals constructed with 100K repetitions.

Although our results answer our questions for this section, it is interesting to also observe here that SynthDynaPrime did not learn a model that can beat other competitive handcrafted inaccurate models like NoPTS Expectation Model Dyna, NoTNS Expectation Model Dyna and NoTNS Stable Experience. Why is that? One hypothesis can be that our model cannot express the data-generating distributions needed given its limited capacity. In that case, we need to increase our model's capacity and observe whether that prompts it to learn even more powerful models. Another hypothesis can be that SynthDynaPrime's model does not have enough transition data to learn the more intricate generative distribution for transitions occurring with preterminal states. In that case, we need to observe performance as a function of increasing data gathered by Q-learning. We should also consider gathering data with a policy that has more randomness, to introduce more data diversity. We leave the investigation of these hypotheses for future work. The model existence question powered by SynthDynaPrime's intentionally idealized design is a fascinating question that can help us explore what performance is possible in an expanded space of models that is not traditionally fully considered in MBRL.

Chapter 5

Useful Model Learning with a Single Experience Stream

Our results in Chapter 4 confirm that we have a loss function that can help with finding useful models that can beat our handcrafted accurate models. The loss function does not place any constraints on the model being accurate, complete or proper. However, the model training procedure we use with this loss function is offline and it assumes privileged access to perfect loss targets. In this Chapter, we consider the idea of training the model within an agent's single experience stream and eliminating access to perfect loss targets.

5.1 SynthDyna Algorithm

To remove SynthDynaPrime's (Algorithm 2) dependence on perfect loss targets (i.e. the true value function parameters θ_{true}), we need to find a proxy for these. One idea is to construct a bootstrapped target as in bootstrapped meta-gradients described in Chapter 2 and in [FSZ+22]. After imagining planning in the metaloss from some arbitrary starting θ to obtain θ', we can apply a fixed number of additional updates to θ' with the model in order to construct a target. This is problematic in our particular case. Since we are learning the data of the planning update, these additional updates with the model will be completely un-grounded by any experience from the world. So the targets can quickly become misleading. In Algorithm 4, we describe our new metaloss. Here, we adopt a safer idea for constructing targets. We apply

the additional updates using temporally coherent real experience, V, from the world rather than using model-generated experience. We require that experiences in V happen sequentially in the world after the input θ is recorded. Real experience updates ensure that the targets constructed are productive to aim for in learning. The sequential nature of θ and V experience ensures that the real updates are done with experiences that occur as close as possible to θ. Thus, these experiences can be as relevant as possible to the environment configuration in which θ occurs. There is an interesting "middle-ground" idea of including some additional model updates along with real experience updates in the construction of targets. We leave this as an avenue for future work.

In Algorithm 5, we introduce SynthDyna, an algorithm that uses the loss in Algorithm 4 to train a model within an agent's single experience stream without assumptions of access to perfect loss targets. This algorithm is exactly like All Experience Dyna in Algorithm 1 except in the construction of the set of experience $\mathcal{D}$ and in the usage of a learned model $m(\eta)$. We maintain two temporally coherent queues of length v: Q_e for experience tuples and Q_θ for value function parameters. As in line 13, this allows us to build $\mathcal{D}$ with tuples of (a) temporally coherent experiences and (b) the value function parameters that occur before them. In lines 17 to 25, we have steps identical to All Experience Dyna except in the fact that planning happens with a learned model in line 23. In lines 27 to 29, we sample a minibatch of tuples from $\mathcal{D}$ and use an optimizer with the metaloss in Algorithm 4 to update model parameters η. In our implementation, we do this every fixed number of time-steps. We also use reservoir sampling to keep the experience set's capacity fixed, but allow the set to hold a uniform random sample of all past experience [Vit85].

Algorithm 4 $\mathcal{L}$: SynthDyna Meta Loss

input: $\boldsymbol{\theta} \in \mathbb{R}^{m \times n}$, $V = [(\hat{\mathbf{s}}_0, \hat{\mathbf{a}}_0, r_0, \hat{\mathbf{s}}'_0), ..., (\hat{\mathbf{s}}_v, \hat{\mathbf{a}}_v, r_v, \hat{\mathbf{s}}'_v)]$
for $1, \cdots, k$ **do**
    $\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}' \sim m(\boldsymbol{\eta})$
    $\tilde{\delta} \leftarrow \tilde{r} + \gamma max(\boldsymbol{\theta}\tilde{\mathbf{s}}') - (\boldsymbol{\theta}\tilde{\mathbf{s}})^T \tilde{\mathbf{a}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \zeta\tilde{\delta}(\tilde{\mathbf{a}} \otimes \tilde{\mathbf{s}})$
$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
for each tuple $(\hat{\mathbf{s}}, \hat{\mathbf{a}}, r, \hat{\mathbf{s}}')$ in V **do**
    $\delta \leftarrow r + \gamma max(\boldsymbol{\theta}'\hat{\mathbf{s}}') - (\boldsymbol{\theta}'\hat{\mathbf{s}})^T \hat{\mathbf{a}}$
    $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}' + \zeta\delta(\hat{\mathbf{a}} \otimes \hat{\mathbf{s}})$
return: $||\boldsymbol{\theta} - \boldsymbol{\theta}'||_2^2$

Algorithm 5 SynthDyna

input: feature transform f
initialize: $\boldsymbol{\theta}_0 \in \mathbb{R}^{m \times n}$, $\boldsymbol{\theta}_p \in \mathbb{R}^{m \times n}$, $Q_e = queue([])$, $Q_\theta = queue([])$, $\boldsymbol{\eta}$, $\mathcal{D} \leftarrow \{\}$, $\hat{\mathbf{s}}_0 \leftarrow f(\mathbf{h}_0)$
for $t = 1, 2, \cdots$ **do**
    Take an action $\hat{\mathbf{a}}_{t-1}$ given by the ϵ-greedy policy, $\pi_{\boldsymbol{\theta}_{t-1}}$
    Observe $\mathbf{o}_t$ and r_t from the environment.
    $\hat{\mathbf{s}}_t \leftarrow f(\mathbf{h}_t)$
    Q_e.append($(\hat{\mathbf{s}}_{t-1}, \hat{\mathbf{a}}_{t-1}, r_t, \hat{\mathbf{s}}_t)$)
    Q_θ.append($\boldsymbol{\theta}_p$)

    // Maintain queues of last v experiences and $\boldsymbol{\theta}$ matrices
    if len(Q_θ) == v **then**
        $\boldsymbol{\theta} \leftarrow Q_\theta.pop(0)$ // Add earliest $\boldsymbol{\theta}$ and the v experiences after it
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{\theta}, Q_e)\}$
        $Q_e.pop(0)$

    // Update action-state value parameters, θ, with veridical experience.
    $\delta \leftarrow r_t + \gamma max(\boldsymbol{\theta}_t \hat{\mathbf{s}}_t) - (\boldsymbol{\theta}_t \hat{\mathbf{s}}_{t-1})^T \hat{\mathbf{a}}_{t-1}$
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \alpha \delta (\hat{\mathbf{a}}_{t-1} \otimes \hat{\mathbf{s}}_{t-1})$
    $\boldsymbol{\theta}_p \leftarrow \boldsymbol{\theta}_t$ // Save $\boldsymbol{\theta}_t$ for model training

    // Update action-state value parameters, θ, with internal experience.
    for $1, \cdots, k$ **do**
        $\tilde{\mathbf{s}}, \tilde{\mathbf{a}}, \tilde{r}, \tilde{\mathbf{s}}' \sim m(\boldsymbol{\eta})$
        $\tilde{\delta} \leftarrow \tilde{r} + \gamma max(\boldsymbol{\theta}_t \tilde{\mathbf{s}}) - (\boldsymbol{\theta}_t \tilde{\mathbf{s}})^T \tilde{\mathbf{a}}$
        $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \beta \tilde{\delta} (\tilde{\mathbf{a}}_t \otimes \tilde{\mathbf{s}})$

    // Update SynthDyna model with metaloss $\mathcal{L}$ (Alg 4).
    Sample minibatch $\mathcal{B}$ from $\mathcal{D}$.
    $\boldsymbol{\eta} \leftarrow$ Optimizer($\boldsymbol{\eta}, \mathcal{B}, \mathcal{L}$)

5.2 Empirical Results

We run SynthDyna (Algorithm 5) on the TMaze environment introduced in Section 3.1. We use the same environment settings, agent state definitions, and evaluation methodology in Section 3.2. Using grid search, we select hyperparameters for SynthDyna by running it for 30000 episodes and evaluating it on the last 1200 episodes. We maintain the same model architecture from SynthDynaPrime (Algorithm 3) and train the model with an RMSProp op-

timizer. We use a fixed capacity of 30000 for $\mathcal{D}$. In Figures 5.1 and 5.2, we observe the performance of SynthDyna (along with SynthDynPrime and baselines). We highlight performance in terms of the Mean Squared return Error at every episode near the last two reward regime switches. In Figures 4.3 and 4.4, we provide the same plots but with the Inter-quartile Mean Squared Return Error as the performance metric. Once again, these plots indicate that SynthDyna can adapt to changes in the environment. However, this adaptation is (unsurprisingly) slower than SynthDynaPrime. SynthDyna can be faster in adaptation than Q-learning in some stretches of time, but it is not always definitively so. In some stretches of time it can roughly settle at a low error unlike All Experience Dyna. Nevertheless, we can see that it sometimes has error spikes in later periods where we would like it to settle.

In Figure 5.5, we summarize performance for all algorithms in this thesis with aggregate metrics over 30 runs per agent. For each agent run, we assess performance with the Mean Squared Return Error (MSRE) over the last 1200 episodes out of 30000 episodes. SynthDyna roughly matches an algorithm with a handcrafted accurate model (NoPTS Stable Experience Dyna) in the Inter-quartile Mean (IQM). SynthDyna obtains an IQM of 0.02496 and a %95 boostrapped percentile confidence interval of (0.02435, 0.02572). NoPTS Stable Experience Dyna obtains an IQM of 0.02530 and a %95 boostrapped percentile confidence interval of (0.02469, 0.02582). Looking at individual run MSRE scores for SynthDyna we find that some runs skew the aggregate Mean MSRE into higher error. This is why we see larger confidence interval bands for the Mean in Figure 5.5. The IQM, however, is more robust to skews in data. Overall, these results indicate that SynthDyna can learn a useful model that at least matches a handcrafted accurate model if we can control for some runs with a higher error skew in performance.

Learning a generative world model in a single stream of data with no special phases (like pre-training) is a non-trivial task. We hypothesize that SynthDyna's performance can be improved further by utilizing tricks from the literature within or adjacent to continual learning. For example, we can try Neural Network layer resetting techniques designed to tackle over-fitting

on early data in RL [NSD+22]. We can also try aggressive L2 regularization or Shrink and Perturb both of which result in better model generalization in situations where new data comes periodically [AA20].

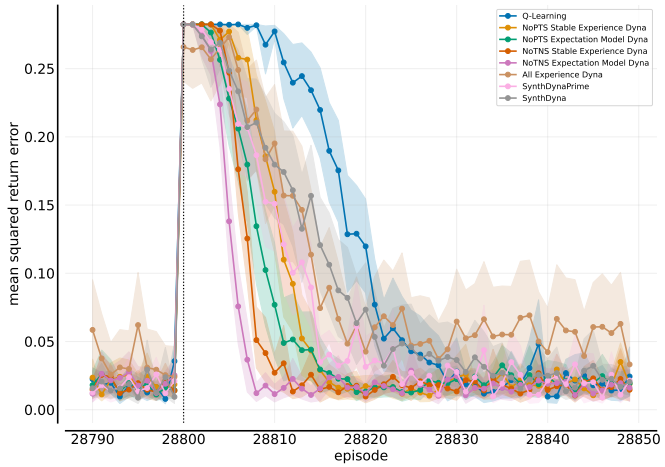


Figure 5.1: Around the first reward switch period in evaluation, we observe the Mean Squared Return Error, per agent, across episodes, in comparison to SynthDyna. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

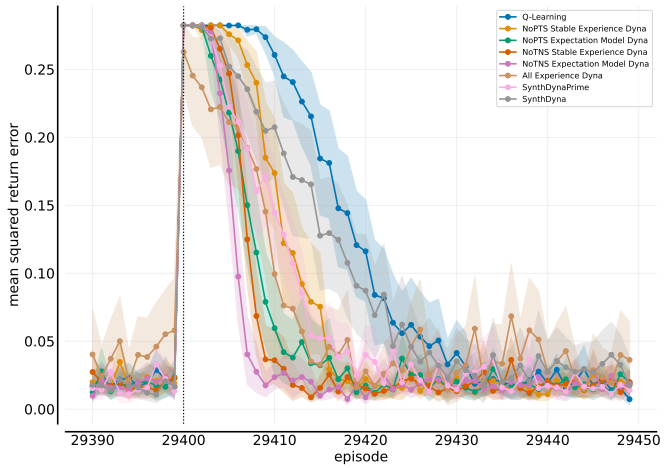


Figure 5.2: Around the second reward switch period in evaluation, we observe the Mean Squared Return Error, per agent, across episodes, in comparison to SynthDyna. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

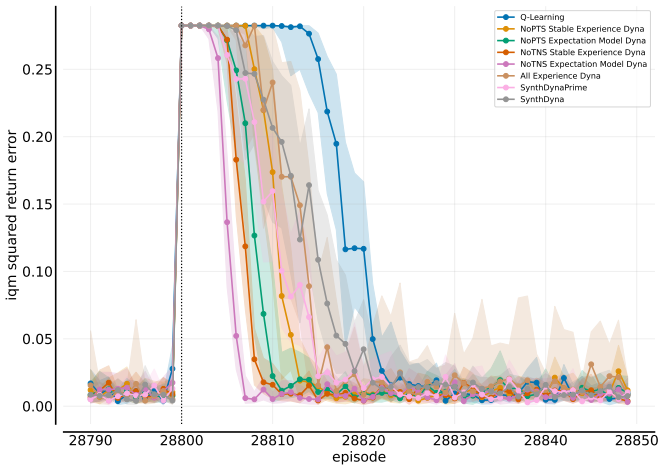


Figure 5.3: Around the first reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error, per agent, across episodes, in comparison to SynthDyna. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 28800-th episode indicated by a dotted line.

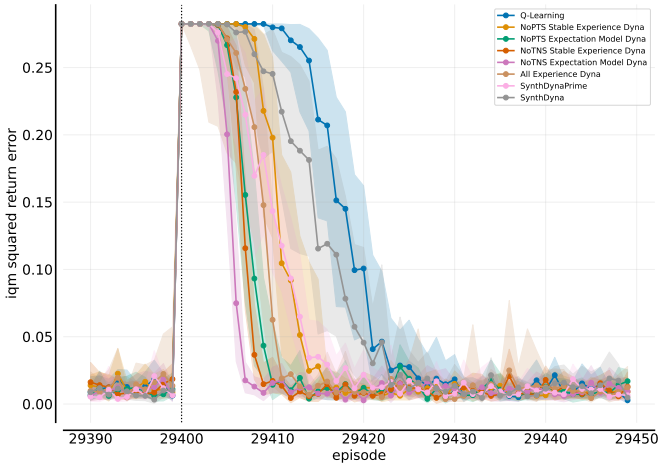


Figure 5.4: Around the second reward switch period in evaluation, we observe the Inter-Quartile Mean (IQM) Squared Return Error, per agent, across episodes, in comparison to SynthDyna. A dot summarizes 30 agent runs in the TMaze in environment. A shaded region is a %95 percentile bootstrap confidence intervals constructed with 100K repetitions. The reward regime switch happens at the 29400-th episode indicated by a dotted line.

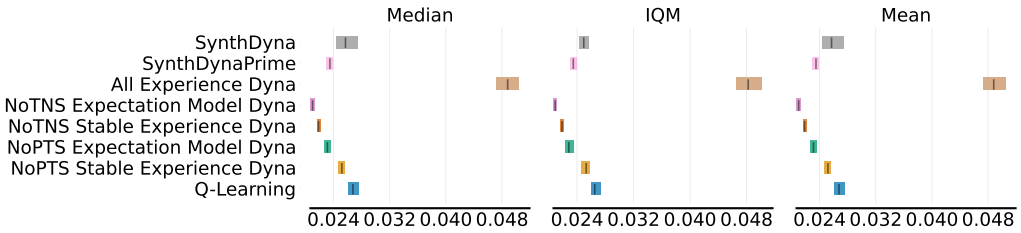


Figure 5.5: Evaluation performance per agent in comparison with SynthDyna, summarized by the Mean Squared Return Error. We show the Median, Inter-Quartile Mean (IQM), and Mean of the performance metric over 30 runs. Intervals around performance point estimates are %95 percentile bootstrap confidence intervals constructed with 100K repetitions.

Chapter 6

Conclusion

Our investigations start by asking: "Should models be accurate?" . This is an important question for Model-based RL as it pushes the limits of our understanding on what useful model inputs and outputs can look like, thus expanding our perception of the accessible space of useful world models. In Chapter 3, our motivating investigations with handcrafted Expectation Models show that there exists inaccurate models that can be more useful than analogous handcrafted accurate models. In fact, there even exists inaccurate, incomplete and improper models that can beat accurate models. In Chapter 4, we introduce SynthDynaPrime to consider the *useful model existence question*: Does there exist *learnable* useful models better than accurate models and whose learning objective does not constrain them to be accurate, complete or proper? Inspired by meta-gradients research, we design a model learning objective that is focused on usefulness to learning rather than accuracy. We also give the model full control over planning data by making it a generative model of full transition tuples. We investigate the potential of the model learning objective with this model design by creating a model training procedure with offline model training and privileged access to true value functions to use as loss targets. Our experiments support answering the useful model existence question in the affirmative. In Chapter 5, we introduce SynthDyna where we eliminate special training phases and consider model-learning within an agent's single stream of experience. We also develop a proxy for true value functions to eliminate the need for them. Our experiments show that SynthDyna can learn a model that

matches an accurate model when using a performance metric that is robust to skewed distributions. Ultimately, our work provides evidence that MBRL can be better served by designing models and learning objectives that allow models to simply be useful and not necessarily accurate or even complete or proper.

As mentioned in various chapters, we think that there are various immediate avenues for future work to improve SynthDyna's performance. First for SynthDynaPrime (a stepping stone to SynthDyna), we think performance can be studied with both data scaling and model scaling, to understand if we can push useful models to the performance of some other inaccurate handcrafted models. Second, we think that outlier lower performance SynthDyna runs can be addressed with some tools recommended for better continual learning like layer weight resets [NSD+22], aggressive L2 regularization or Shrink and Perturb [AA20]. These strategies can help us in avoiding over-fitting to early experience.

There also more long-term avenues of future work that we think are particularly exciting. One idea is to meta-learn the discount factor along with data for a planning update. This can allow for learning model abstractions at multiple timescales. Another idea is to make the model more contextually sensitive by conditioning it on some summary of the current history or the current state. A third idea is to combine the metaloss presented here with better model architectures like Dreamer [HLNB20] or like Score-based Diffusion Models. Overall, our investigations represent an initial exploration of an expanded space of models that are not accessible with traditional model designs and model learning objectives focused on accuracy. We believe that there is tremendous potential for developing competitive MBRL algorithms when the focus is on model usefulness rather than model accuracy.

References

[AA20] Jordan Ash and Ryan P Adams. On warm-starting neural network training. *Advances in Neural Information Processing Systems*, 33:3884–3894, 2020.

[ASC+21] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34:29304–29320, 2021.

[ASTW20] Zaheer Abbas, Samuel Sokota, Erin Talvitie, and Martha White. Selective dyna-style planning under limited model capacity. In *International Conference on Machine Learning*, pages 1–10. PMLR, 2020.

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[FSZ+22] Sebastian Flennerhag, Yannick Schroecker, Tom Zahavy, Hado van Hasselt, David Silver, and Satinder Singh. Bootstrapped meta-learning. In *International Conference on Learning Representations*, 2022.

[GBSS20a] Christopher Grimm, André Barreto, Satinder Singh, and David Silver. The value equivalence principle for model-based reinforcement learning. *Advances in Neural Information Processing Systems*, 33:5541–5552, 2020.

[GBSS20b] Christopher Grimm, André Barreto, Satinder Singh, and David Silver. The value equivalence principle for model-based reinforcement learning. *Advances in Neural Information Processing Systems*, 33:5541–5552, 2020.

[HLF+19] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pages 2555–2565. PMLR, 2019.

[HLNB20] Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2020.

[HS18] David Ha and Jürgen Schmidhuber. World models. *Advances in Neural Information Processing Systems*, 2018.

[HTB18] G Zacharias Holland, Erin J Talvitie, and Michael Bowling. The effect of planning shape on dyna-style planning in high-dimensional state spaces. *arXiv preprint arXiv:1806.01825*, 2018.

[IFKC16] Nursultan Imanberdiyev, Changhong Fu, Erdal Kayacan, and I-Ming Chen. Autonomous navigation of uav by using real-time model-based reinforcement learning. In *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1–6, 2016.

[JIT+20] Taher Jafferjee, Ehsan Imani, Erin Talvitie, Martha White, and Micheal Bowling. Hallucinating value: A pitfall of dyna-style planning with imperfect environment models. *arXiv preprint arXiv:2006.04363*, 2020.

[LPC22] Nathan Lambert, Kristofer Pister, and Roberto Calandra. Investigating compounding prediction errors in learned dynamics models. *arXiv preprint arXiv:2203.09637*, 2022.

[MBT+18] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[NSD+22] Evgenii Nikishin, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. The primacy bias in deep reinforcement learning. In *International Conference on Machine Learning*, pages 16828–16847. PMLR, 2022.

[SAH+20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[SMK+22] Esra'a Saleh, John D. Martin, Anna Koop, Arash Pourzarabi, and Michael Bowling. Should models be accurate? *The Multidisciplinary Conference on Reinforcement Learning and Decision Making*, 2022.

[SRL+20] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth Stanley, and Jeffrey Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. In *International Conference on Machine Learning*, pages 9206–9216. PMLR, 2020.

[SS10] Jonathan Sorg and Satinder Singh. Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 31–38, 2010.

[Sut91] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[Sut92] Richard S Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, volume 92, pages 171–176. Citeseer, 1992.

[Tal14] Erik Talvitie. Model regularization for stable sample rollouts. In *UAI*, pages 780–789, 2014.

[TWT+16] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in Neural Information Processing Systems*, 29, 2016.

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[WEH+23] Philipp Wu, Alejandro Escontrela, Danijar Hafner, Pieter Abbeel, and Ken Goldberg. Daydreamer: World models for physical robot learning. In *Conference on Robot Learning*, pages 2226–2240. PMLR, 2023.

[XvHH+20] Zhongwen Xu, Hado P van Hasselt, Matteo Hessel, Junhyuk Oh, Satinder Singh, and David Silver. Meta-gradient reinforcement learning with an objective discovered online. *Advances in Neural Information Processing Systems*, 33:15254–15264, 2020.

[XvHS18] Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[ZXV+20] Tom Zahavy, Zhongwen Xu, Vivek Veeriah, Matteo Hessel, Junhyuk Oh, Hado P van Hasselt, David Silver, and Satinder Singh.

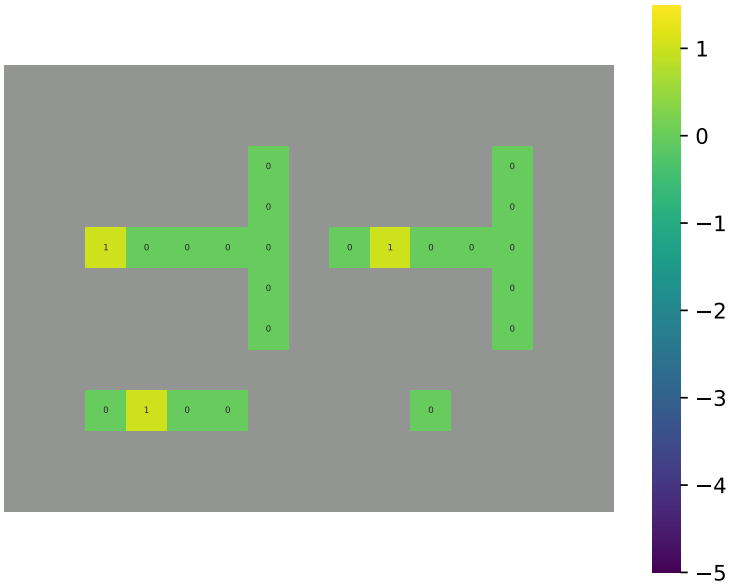
A self-tuning actor-critic algorithm. *Advances in Neural Information Processing Systems*, 33:20913–20924, 2020.

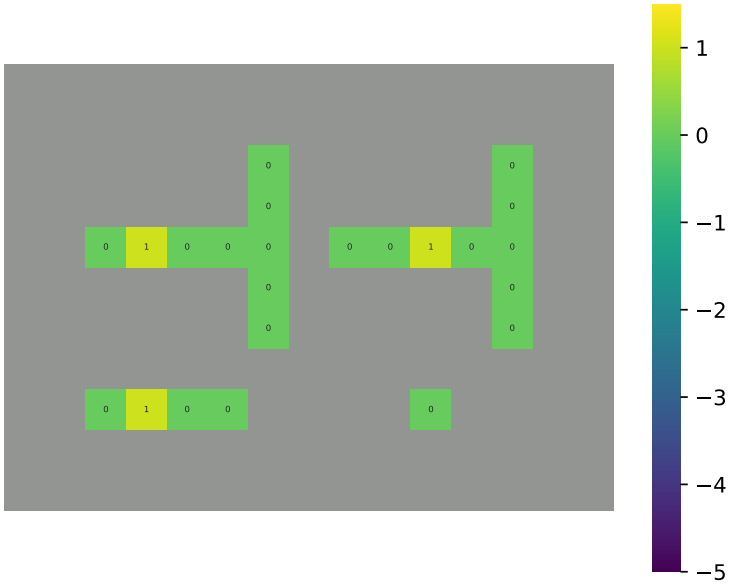
Appendix A

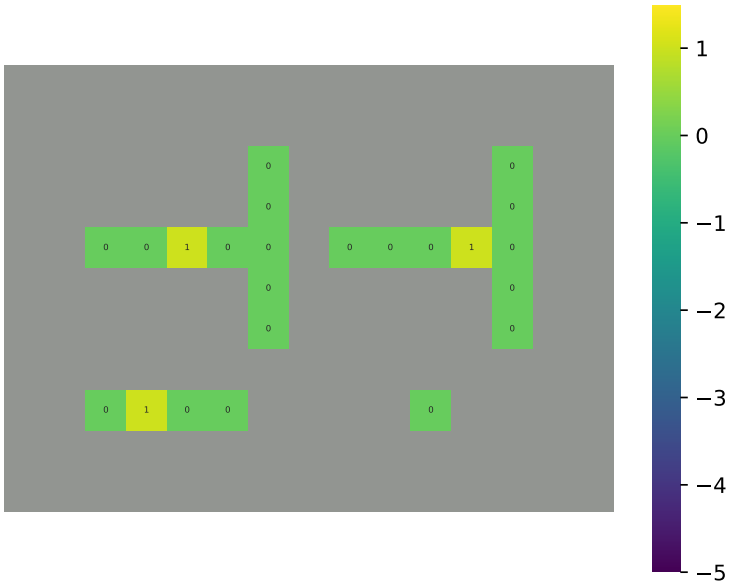
Examples of Synthetic and Real Experience

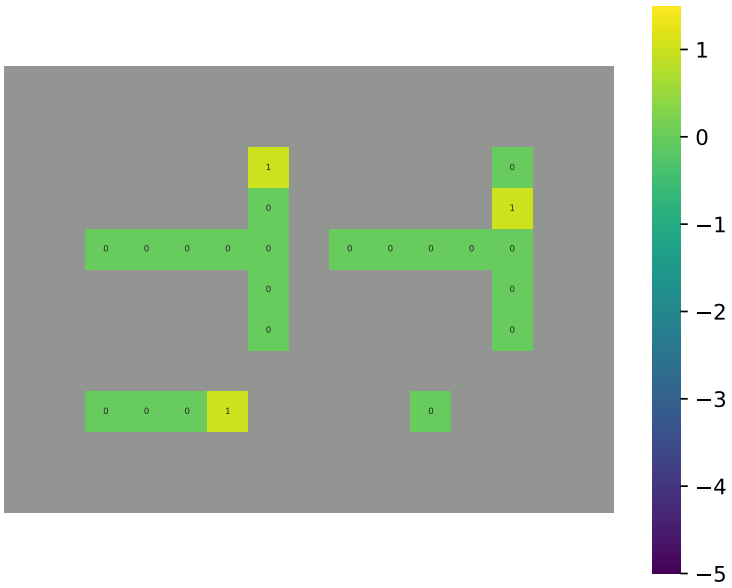
Below, we provide heatmaps of synthetic experience generated by SynthDynaPrime's model (See Chapter 4). If we contrast these synthetic experience examples to "real" experience as given by AllExperienceDyna's model or as given by models of StableExperience agents, we can see how these synthetic experience examples do not map to any transitions that can happen in the world. Each plot below summarizes experience examples by showing values of a state vector (mapped onto a TMaze to the top left), action vector (the four boxes to the bottom left, where each box represents a cardinal direction), reward (the single box on the bottom right), and next state vector (mapped onto a TMaze to the top right).

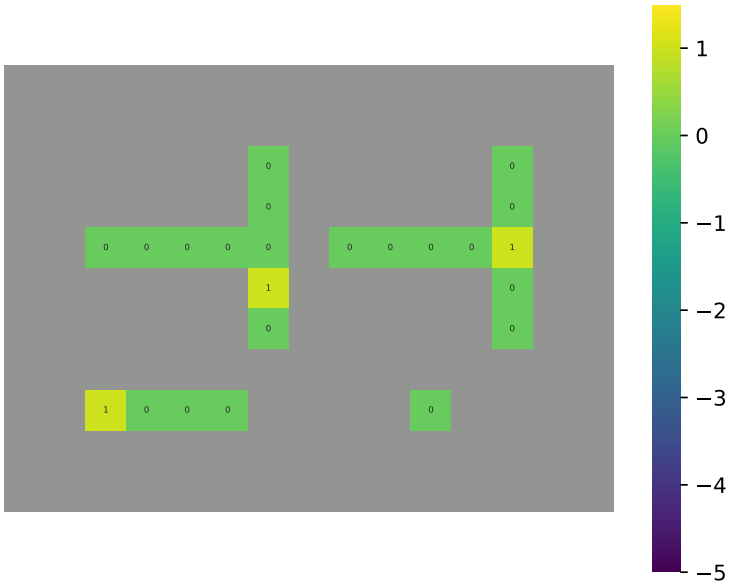
A.1 Real Experience

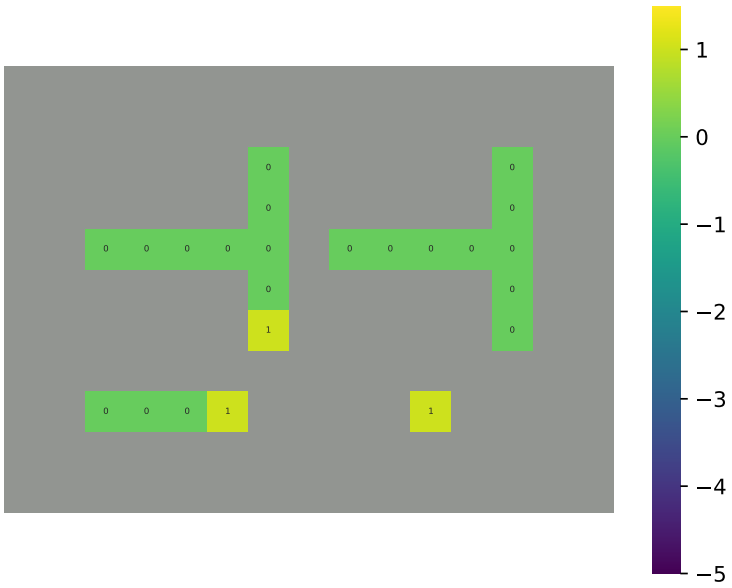




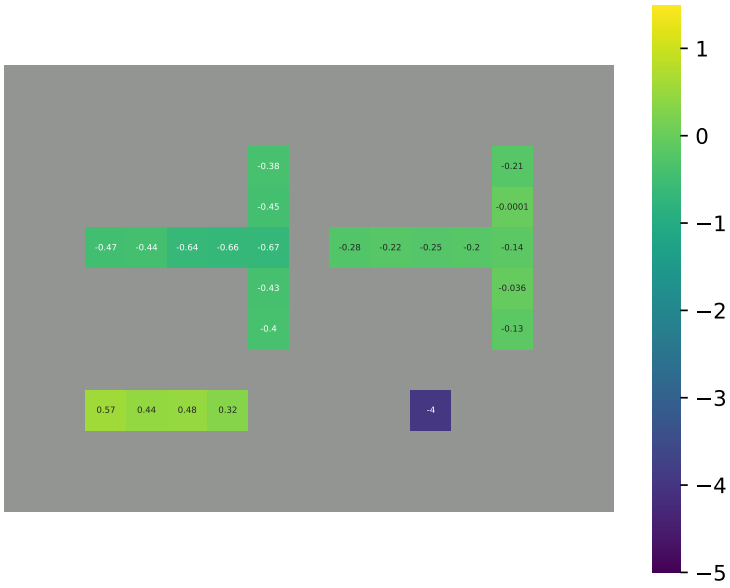

1
0
−1
−2
−3
−4
−5

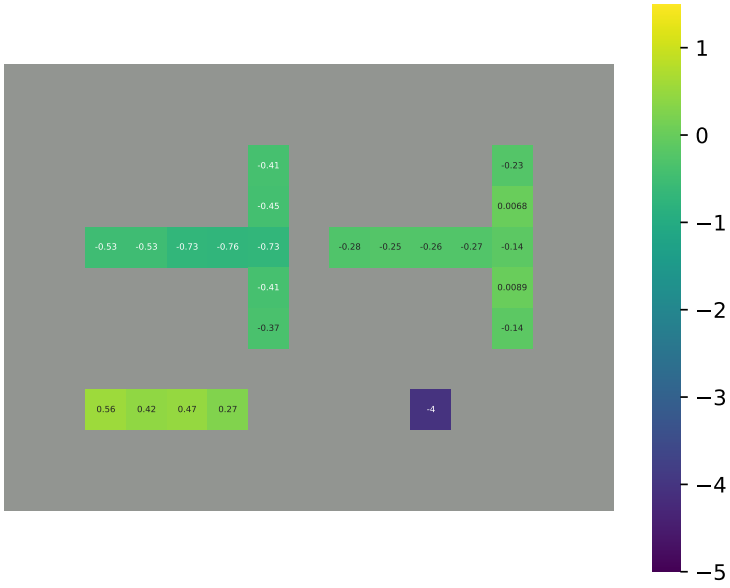

1
0
−1
−2
−3
−4
−5

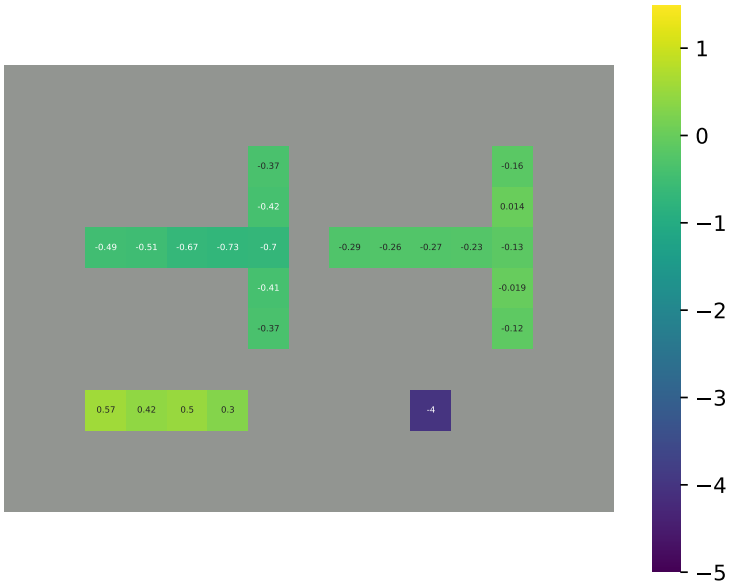

1
0
−1
−2
−3
−4
−5

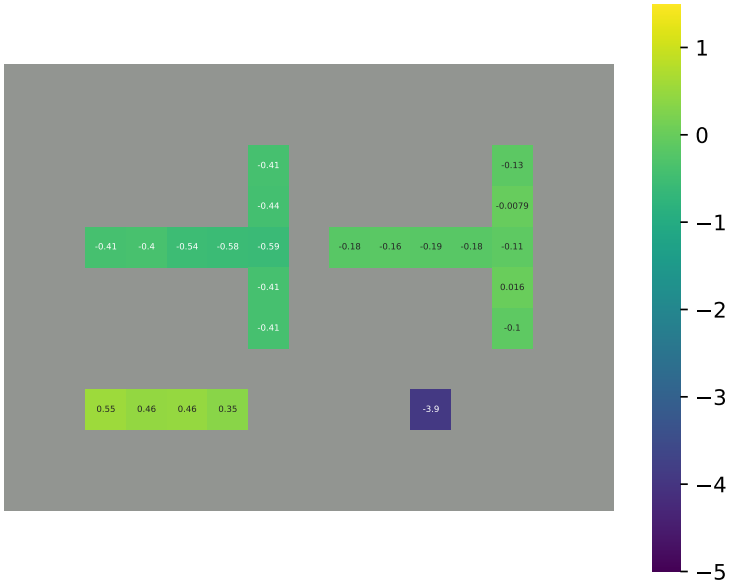

1
0
−1
−2
−3
−4
−5

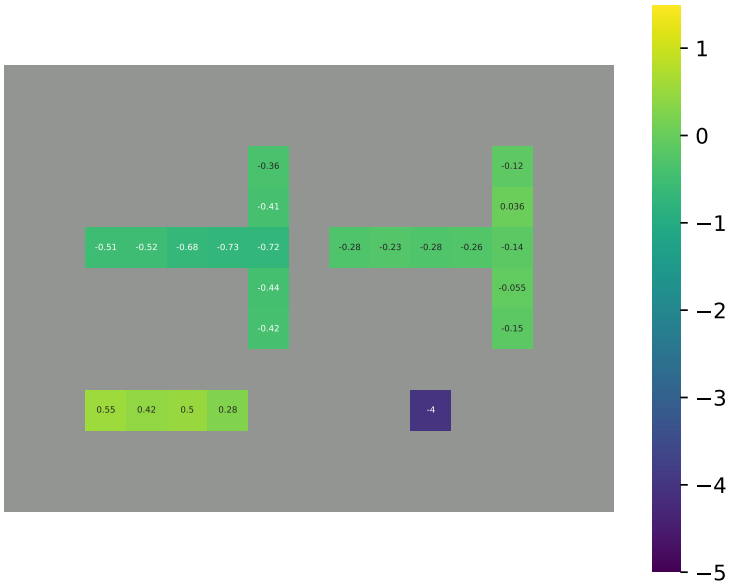
A.2 Synthetic Experience

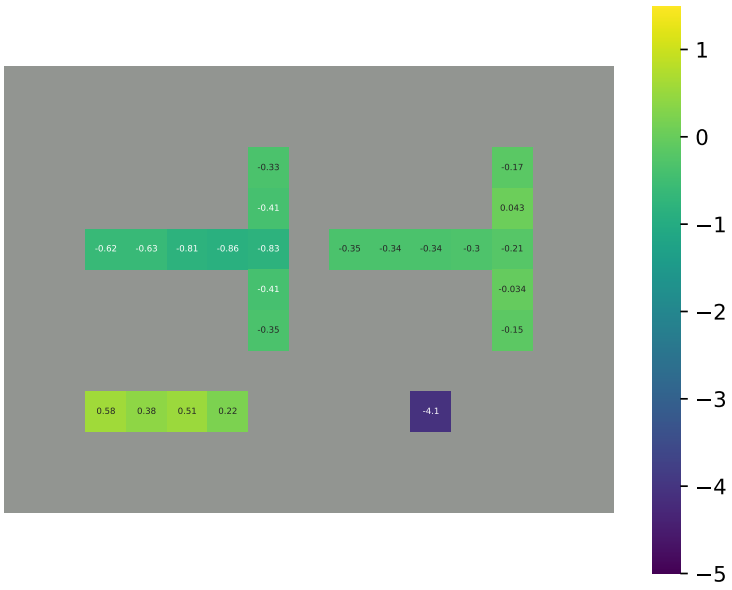





-0.37
-0.42
-0.49
-0.51
-0.67
-0.73
-0.7
-0.41
-0.37
0.57
0.42
0.5
0.3
-4
-0.16
0.014
-0.29
-0.26
-0.27
-0.23
-0.13
-0.019
-0.12
1
0
−1
−2
−3
−4
−5


-0.41
-0.44
-0.41
-0.4
-0.54
-0.58
-0.59
-0.41
-0.41
0.55
0.46
0.46
0.35
-3.9
-0.13
-0.0079
-0.18
-0.16
-0.19
-0.18
-0.11
0.016
-0.1
1
0
−1
−2
−3
−4
−5


-0.36
-0.41
-0.51
-0.52
-0.68
-0.73
-0.72
-0.44
-0.42
0.55
0.42
0.5
0.28
-4
-0.12
0.036
-0.28
-0.23
-0.28
-0.26
-0.14
-0.055
-0.15
1
0
−1
−2
−3
−4
−5


-0.33
-0.41
-0.62
-0.63
-0.81
-0.86
-0.83
-0.41
-0.35
0.58
0.38
0.51
0.22
-4.1
-0.17
0.043
-0.35
-0.34
-0.34
-0.3
-0.21
-0.034
-0.15
1
0
−1
−2
−3
−4
−5