Analyzing the Prediction Accuracy of Trajectory-Based Models with High-Dimensional Control Policies for Long-term Planning in MBRL

by Howard Zhang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

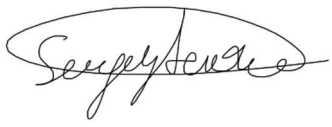
Committee:

Professor Kristofer S.J. Pister
Research Advisor

5/10/2021

(Date)

* * * * * * *



Professor Sergey Levine
Second Reader

5/10/2021

(Date)

Analyzing the Prediction Accuracy of Trajectory-Based Models with High-Dimensional Control Policies for Long-term Planning in MBRL

Howard Zhang

April 2021

1 Acknowledgment

Thank you to Nathan Lambert and Roberto Calandra for providing so much help and guidance throughout the year. I also want to express my gratitude towards Professor Kristofer Pister, whose insights and advice benefited not only my research, but my academic career as a whole. Lastly, I wanted to thank my family and my friends Leon Idelchik, Varan Nimar, Michael Wang, and Andre Yu for supporting me in a year that was difficult for all of us.

2 Abstract

Learning effective policies with model-based reinforcement learning is highly dependent on the accuracy of the dynamics model. Recently, a new parametrization called the trajectory-based model was introduced, which takes in an initial state, a future time index, and control policy parameters, and returns the state at that future time index [3]. This new method has demonstrated improved prediction accuracy in long horizons, increased sample efficiency, and ability to predict the task reward. However, this model has limited transferability to MBRL due to the limited expressivity of its low-dimensional control policy parameter inputs. In this work, we look at the effectiveness of the trajectory-based model at predicting environment dynamics with higher-dimensional and expressive neural network control policies. The trajectory-based model has demonstrated some capability in learning from these neural network policies, and still outperforms the traditional state-action one-step model due to less compounding error.

3 Introduction

In this section, we will take a look at the current landscape of reinforcement learning, and where trajectory-based models fit into that landscape. Then, we will take a look at the formalization of the reinforcement learning objective as a Markov Decision Process (MDP), which will be used in future sections to describe the different RL algorithms, control policies, and dynamics models.

3.1 Reinforcement Learning

Reinforcement learning is a form of machine learning that is concerned with the maximization of a reward of an agent in an environment. In recent years, it has been used extensively for finding optimal policies for robotic control. There are two primary branches of reinforcement learning: model-free and model-based. Model-free reinforcement learning procedures typically directly interact with the environment to find the optimal policy [9,10,11]. However, these methods typically require copious amounts of trajectory data to find an optimal policy. This data collection step can sometimes be limited, especially when performing experiments on actual hardware rather than simulation. Model-based reinforcement learning is structured on the idea of first learning the dynamics of a particular environment, then using this dynamics model to find the optimal control policy [2,1]. The benefit of this method is that it is much more sample efficient, since we can use the model to generate more samples. However, policies that are learned from model-based reinforcement learning are highly dependent on the accuracy of the learned model. Often, this means that the learned policies are handicapped by this prediction accuracy. This is why it is extremely important to develop high-accuracy dynamics models.

3.2 Trajectory-based Dynamics Models

A majority of dynamics models used in model-based reinforcement learning are made to learn state transitions. In other words, these models typically take a current state and action distribution, and output the distribution of the next state [2,1]. However, these methods often suffer from the issue of compounding errors [5,6], and have also been shown in the past to have objective mismatch with the reinforcement learning objective of learning an optimal controller [4]. Both of these issues will be expounded upon further in the "Related Works" section. Lambert et al. [3] has introduced a new parametrization on the model-based reinforcement learning problem: a "trajectory-based model" that takes in an initial state, a future time index, and control policy parameters, and outputs the state at that future time index. This new time-dependent trajectory-based model seeks to learn from the entire trajectory, rather than independent state-action pairs. These models have been shown to have higher prediction accuracy at longer horizons and improved sample efficiency compared to the traditional one-step model. However, due to the control policy input, the network has limited expressivity. In this paper, we aim to test the prediction accuracy and

learning potential of the trajectory-based model on simulated gym environments with high expressivity, complex neural network control policies.

3.3 Markov Decision Processes

The Markov Decision Process (MDP) was first formalized by Bellman [7], and is a mathematical framework used to describe decision-making based on probabilistic distributions. It is commonly used in reinforcement learning literature to formulate common objectives and problems. In Figure 1, we see a typical Markov Decision Process with 3 time steps. In the MDP formulation, we use states $s \in \mathbb{R}^{D_s}$ and actions $a \in \mathbb{R}^{D_a}$, with D_s and D_a equal to the sizes of the state dimension and action dimension, respectively. We get the action from a control policy $\pi_\theta(a_t|s_t) : \mathbb{R}^{D_s} \mapsto \mathbb{R}^{D_a}$, which gives a distribution over the action space given a state. The control policy itself is also dependent on parameters θ. This control policy can be anything from PID or LQR with very few θ parameters to a complex neural network with hundreds of θ weight and bias parameters. Each next state is gotten from the state transition distribution $p(s_{t+1}|s_t, a_t) : \mathbb{R}^{D_s+D_a} \mapsto \mathbb{R}^{D_s}$, which gives a distribution over the next state given the current state and action. In reinforcement learning, there is also commonly a reward associated with each time step that we get from a reward function $r(s_t, a_t) : \mathbb{R}^{D_s+D_a} \mapsto \mathbb{R}$. It should be noted that, when running in real environments, we frequently do not have access to the state s_t, only an observation, in which case we have a Partially Observed Markov Decision Process. In this case, we would need to use a model representing the probability distribution over the observation given the state, and the control policy π_θ would be a distribution over actions given the observation, not the state. For the purposes of this paper, we will be assuming that we have full access to the state for simplicity. Additionally, we will assume the Markov Property, which means that each next state can be fully determined by the current state and action, and will not be dependent on any previous states.

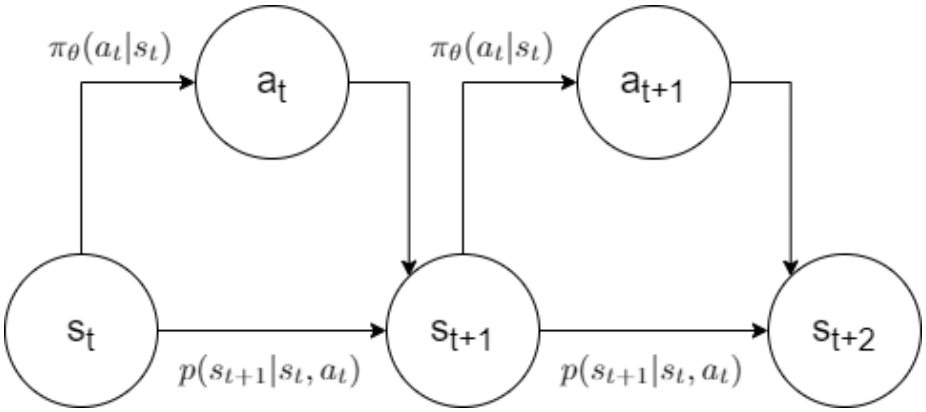


Figure 1: A diagram depicting a typical Markov Decision Process used for reinforcement learning.

4 Related Work

In this section, we will discuss a few of the popular model-free and model-based reinforcement learning methods. Additionally, we will discuss how common model-free RL algorithms deal with the issue of sample efficiency, and how the different model-based methods work with the one-step model.

4.1 Model-free Reinforcement Learning

4.1.1 Proximal Policy Optimization

As discussed previously, model-free reinforcement learning aims to learn a control policy directly from the agent interacting with the environment. Proximal Policy Optimization uses a variant of the vanilla policy gradient method [11]. The vanilla policy gradient method optimizes the control policy by running gradient ascent on the objective function in equation (1), with the optimization parameter being the θ parameters of the control policy $\pi_\theta(a_t|s_t)$. $\hat{A}_t$ is known as an advantage function, and is a measure of the expected cumulative reward from that time step onward. The expected value means that this advantage function is estimated using an average over some finite batch of samples using the policy $\pi_\theta(a_t|s_t)$. Simple models for the advantage function include Monte Carlo estimates, which simply just sum up all of the rewards from that time step onward for that trajectory. Often, this results in single sample estimates of the advantage function. This is unless a simulator is involved, in which case you could simply rewind back to the previous time step and see multiple paths that the agent could have taken and average together all of the cumulative rewards of each path. Vanilla policy gradient is on-policy, meaning gradient updates can only be made when the trajectories used to estimate the advantage function are gotten with the policy used in that update. In other words, with a single

sampled trajectory, we can only make one gradient ascent update on the policy parameters θ. In practice, multiple updates can be made per trajectory, but this is not well-justified and making too many updates will result in a bad policy.

This is a major issue for policy gradient algorithms, since complex neural network control policies often require thousands of gradient updates to be trained properly, and there is often a limitation on the number of trajectories that can be run on an agent, especially with hardware and not simulation. The way Schulman et al. [11] alleviates this issue is with Trust Region Methods [12], which utilize a modified objective function, seen in equation (2) and (3). Equation (2) is a measure of the ratio between the current policy π_θ and the old policy $\pi_{\theta_{old}}$. The trajectory used to estimate the advantage function $\hat{A}_t$ is gotten with the old policy $\pi_{\theta_{old}}$, and gradient updates are done on the current policy π_θ. To ensure that the current policy does not deviate too far from the old policy that was used to estimate the advantage function, Schulman et al. [12] uses a constraint for the KL divergence of the two policies, which can also be modified to be a penalty in the actual objective function with a tuning hyperparameter. However, the Proximal Policy Optimization algorithm [11] opts for using (3), which takes the minimum between the Trust Region Method objective $f_t(\theta)\hat{A}_t$ and a clipped version of that objective to ensure that the ratio $f_t(\theta)$ does not venture outside $1-\epsilon$ and $1+\epsilon$. This new policy gradient method has proven that it is able to gain better performance compared to vanilla policy gradient with traditional Trust Region Methods [12], and many other continuous reinforcement learning methods at the time. Using trust region objectives is just one way that traditional model-free RL methods alleviate the need for extra sample efficiency.

$$L(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{1}$$

$$f_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(f_t(\theta)\hat{A}_t, clip(f_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{3}$$

4.1.2 Continuous Deep Q-Learning with Model-Based Acceleration

Another very common model-free RL method is Q-learning. To understand this, we first need to introduce value functions, Q functions, and advantage functions (note that these are slightly different than the "advantage functions" in the previous section). Value functions are a measure of the expected cumulative reward conditioned on a particular state. Q functions are a measure of the expected cumulative reward conditioned on a particular state and taking a particular action at that state. Advantage functions are the difference between the Q function and the value function, and represent the amount of "extra" reward we get by taking that particular action amongst all possible actions at that particular state. These terms are summarized in equations (4), (5), and (6). For infinite time horizon, we use a discount factor γ in order to make

the cumulative reward finite. The idea behind Q-learning is that we can simply choose the action at each time step that maximizes the Q function at that state, because that represents the action that will maximize our cumulative reward. In other words, follow the policy from (7). We estimate the Q function using dynamic programming. This means we train the Q function model towards the target (8). Note that this is not traditional gradient descent, since we update the Q function, which is in the target. Because of the moving target, there are no convergence guarantees and Q-learning is difficult to train. There are other methods of improving this algorithm, such as keeping track of two Q function models to improve stability in training, but these will not be discussed as they are not important to the discussion in this paper.

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | (s_0 = s)] \tag{4}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | (s_0 = s, a_0 = a)] \tag{5}$$

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{6}$$

$$\pi_\theta(a|s) = \arg\max_a(Q_\pi(s,a)) \tag{7}$$

$$Q(s_t,a_t) = r(s_t,a_t) + Q(s_{t+1},a_{t+1}) \tag{8}$$

Note that the Q-learning method above is primarily used for discrete action spaces, since the algorithm requires an argmax over the Q function. For continuous action spaces, the popular method of choice is called an actor-critic algorithm, which is essentially policy gradient using a learned advantage function (6) for the $\hat{A}_t$ in (1), which performs better than naive Monte Carlo estimates. However, actor-critic algorithms require two neural network models to be trained, one for the value function $\hat{A}_t$, and one for the policy itself $\pi_\theta(a|s)$. Two models require more data to train properly than a single model, and we again have an issue of requiring too many samples to properly train a model-free RL algorithm. Gu et al. [10] aims to improve RL in the continuous action space by creating a new Q-learning algorithm that does not require discrete actions. It does this by creating a neural network model that outputs the value function and the advantage function. Together, we can use (6) to get the Q function estimate $Q_\pi(s,a) = A_\pi(s,a) - V_\pi(s)$. The advantage function is parameterized as a quadratic function (9), and the neural network outputs the μ and P parameters. By doing this, the argmax of the Q function is simply given by $\mu_\theta(s)$. In creating a Q-learning function that is usable in the continuous action space, we only have to train one neural network model, which drastically decreases the amount of samples needed for training. Gu et al. [10] additionally improves the sample efficiency of the algorithm by using a simple model-based approach. The agent following a Q-learning procedure often needs to do ex-

ploration to learn an accurate Q-function. This is typically done with epsilon greedy or other exploration schedules that will not be discussed here. However, these exploration actions can sometimes make the agent move into potentially dangerous locations in the environment (if done on hardware). To fix this, Gu et al. [10] employs a simple local linear model to represent the dynamics of the environment. This not only allows the Q-learning algorithm to explore more effectively in potentially dangerous areas, but also helps with the sample efficiency of the algorithm. This paper not only highlights the issue of sample efficiency in model-free RL methods, but also demonstrates why model-based approaches solve this issue.

$$A_\theta(s,a) = -\frac{1}{2}(a-\mu_\theta(s))^T P_\theta(s)(a-\mu_\theta(s)) \tag{9}$$

4.1.3 Discussion of Model-Free Reinforcement Learning Algorithms

A common theme shows up in the discussion of these model-free RL algorithms: the necessity for higher sample efficiency algorithms. Because model-free algorithms solve for an optimal policy directly without prior knowledge of the environment, a massive number of trajectories are usually needed to train these policies. The methods in the papers above used to fix this include the use of trust regions (for more policy updates on the same trajectory) and training less models (Q-learning over actor-critic). Many modern methods also now have "pseudo-model-based" approaches by incorporating simple dynamics models such as locally linear models to improve sample efficiency and help agents explore in potentially dangerous environments.

4.2 Model-Based Reinforcement Learning

4.2.1 PILCO

Deisenroth et al. [2] developed the PILCO algorithm, which is a model-based RL algorithm that uses Gaussian Processes to capture the uncertainty in dynamics estimation. Gaussian processes are a different type of model to classic neural networks. Gaussian processes use different Gaussians and the covariances between a new input and these Gaussians to make a prediction on where the output is. The covariances are captured by kernel functions. The PILCO algorithm uses a squared exponential kernel. We will not go in-depth into the explanation of these models. The important thing about these Gaussian processes is that because they use the covariances of Gaussian distributions to model the input, output relations, the uncertainties of whatever function they are representing are captured. For the purposes of the PILCO algorithm, the Gaussian process represents the state transition dynamics. So the inputs are the current state and action, and the output is the delta state, which is the difference between the next state and current state. The Gaussian Process model captures the inherent uncertainty in the environment's state transitions $p(s_{t+1}|s_t, a_t)$, and outputs a probability distribution for s_{t+1}. The use of delta state outputs $(s_{t+1} - s_t)$

rather than simple next state outputs (s_{t+1}) increases the stability of the state transition dynamics model. The controller $\pi_\theta(a_t|s_t)$ in PILCO is directly tied to the Gaussian Process dynamics model (which takes in actions as an input). These controllers can also have many different parameterizations. For example, in the cartpole environment, the algorithm used a nonlinear RBF state feedback controller. The loss function for PILCO is simply the difference between the target state and the predicted state from the world model. For each data point, we then do standard backpropagation to learn the parameters θ for the controller.

4.2.2 Probabilistic Ensembles with Trajectory Sampling

The PETS paper by Chua et al. [1] recognized that, while the Gaussian processes in the PILCO algorithm [2] learned much faster and therefore had higher sample efficiency, Gaussian processes tend to fall off in predictive accuracy with more complex environments, due to their lower expressive power. Neural network models, however, have much higher expressive power compared to Gaussian processes. In order to incorporate the uncertainty-aware predictions of the Gaussian process dynamics model with the expressive power of neural networks, Chua et al. [1] used neural networks that output probability distributions. These probabilistic networks also model the state transition dynamics of the environment, similar to the Gaussian processes. They take in as input the current state and action and output a mean and variance for the delta state $s_{t+1} - s_t$. The use of probabilistic networks instead of traditional deterministic ones capture the aleatoric uncertainty in the model, which is the uncertainty tied to the random processes in the environment. Chua et al. [1] also use ensemble training to capture the epistemic uncertainty in the model, which is the uncertainty tied to the variance in the model parameters themselves. Ensemble training is the training of multiple neural network models, each with their own set of model parameters. Averaging the outputs of these models together decreases the variance of the output without biasing it.

The full algorithm uses model predictive control (MPC) to optimize the control policy. It trains the probabilistic ensemble model on an initial dataset, then uses a cross-entropy method on the model to optimize for the best set of actions to take to maximize the cumulative reward function. Cross-entropy methods (CEM), select random actions from a candidate distribution, then these distribution parameters are updated based on the actions that generated the best cumulative reward [13]. Once we have a set of actions that are optimized, we take one step in the environment with the first action, then re-optimize from the new state. This is done to correct for any mistakes arising from inaccurate dynamics, since "one-step" models that model the state transition dynamics usually have compounding error that lead to inaccurate predictions in horizons far away from the initial [5,6]. After finishing a trajectory, we add all of the state-action pairs from running this trajectory into the training dataset, and retrain the probabilistic ensemble model. This alleviates the distributional shift

issue, which arises when the trajectory we are predicting is not in the training set of the dynamics model. This is because the initial training dataset is generated using completely random policies, but the trajectory that we are following (and therefore generating predictions from the dynamics model) follows an optimal policy we get from MPC. If the random policy and the optimal policy are very different from each other, the predicted trajectory will fall outside of the distribution of training trajectories, and the prediction accuracy will drop greatly. By re-training after each trajectory, we train on additional trajectories that essentially help the model "explore" new parts of the environment and increase its prediction accuracy in those domains. Overall, this PETS algorithm led to results that are on par with popular model-free approaches such as Proximal Policy Optimization [11], but required much less samples to reach this point. For the Half-Cheetah simulated environment, it required roughly 125 times less samples to reach the same results as Proximal Policy Optimization.

4.2.3 Trajectory-Based Models

Lambert et al. [3] describes a new parameterization of the model-based RL problem. The previous methods described above both use models that represent the state transition dynamics, or the "one-step" dynamics. This model has been shown in the past to result in compounding errors [5,6]. Xiao et al. [5] notes the issue of compounding errors traditional one-step models, and mitigates the issue by adapting the planning horizon depending on the state. The reasoning behind this is that certain states in the environment have simple dynamics that can be learned easily at longer horizons without much compounding error, while other states have more complex dynamics, and the compounding error issue leads to much lower predictive accuracy. Asadi et al. [6] also notes the error magnification effects when the one-step model is composed onto itself (feeding the output of the network back into the input). It attempts to resolve this using multi-step models that can output the resulting state after taking a particular sequence of actions. Besides the compounding error issue, one-step networks have also been shown in the past to have an objective mismatch issue [4]. That is, the objective of improving the prediction accuracy of a one-step model is not correlated with the objective of finding the optimal control policy for an environment. Lambert et al. [3] expands upon previous attempts to fix compounding errors by introducing the "trajectory-based" model, which trains a time-dependent network from the entire trajectory as a whole. As seen in Figure 2, the model takes in the initial state s_0, a time index into the future t, the parameters θ that determine the control policy $\pi_\theta(a_t|s_t)$. It outputs the state at that future time index: s_t. Lambert et al. [3] has already demonstrated that with simple $\pi_\theta(a_t|s_t)$ control policies like PID or LQR, the trajectory-based model has higher prediction accuracy especially at longer horizons and increased sample efficiency. However, simple control policies like LQR have fewer parameters (4 for the cartpole environment). By training the model with an LQR control policy, we constrain the model to only be able to predict for trajectories that use LQR control policies. This severely limits the expressivity

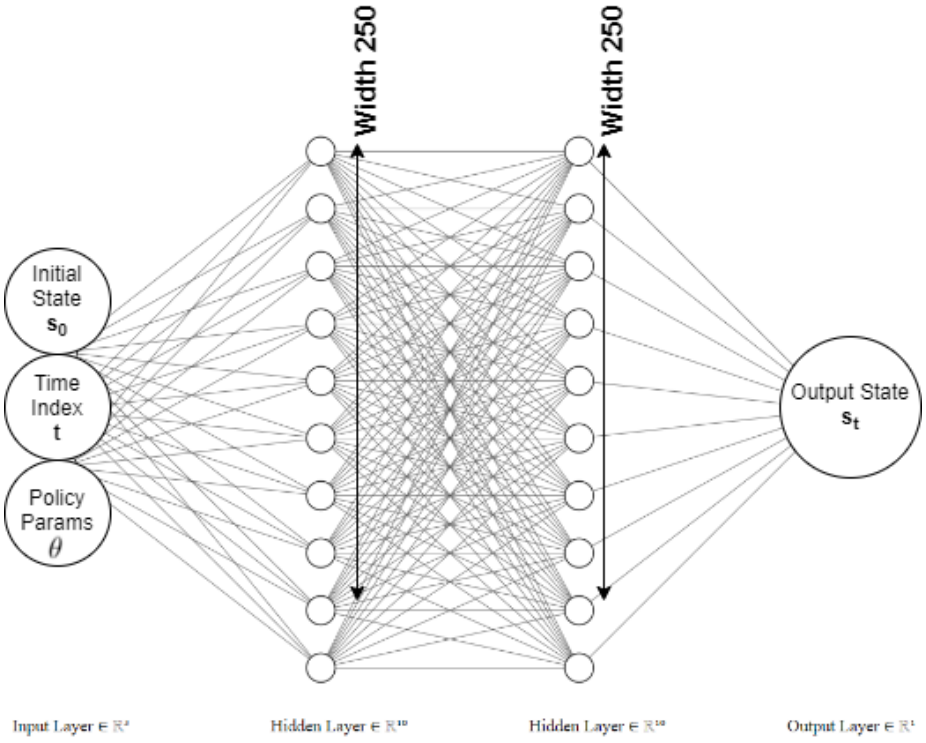


Figure 2: A diagram showing the architecture, inputs, and outputs of the Trajectory-Based Model.

of the model and the types of control policies that it can be used to optimize. In order to use this trajectory-based model to optimize a control policy with a similar algorithm as PETS [1], we need to train the model with much more complex and expressive policies, like neural networks, which can have upwards of 100 weight and bias parameters even for smaller architectures. In this paper, we look into the prediction accuracy of trajectory-based dynamics models with complex neural network policies of varying architectures and sizes, along with some of the issues that arise when training these networks.

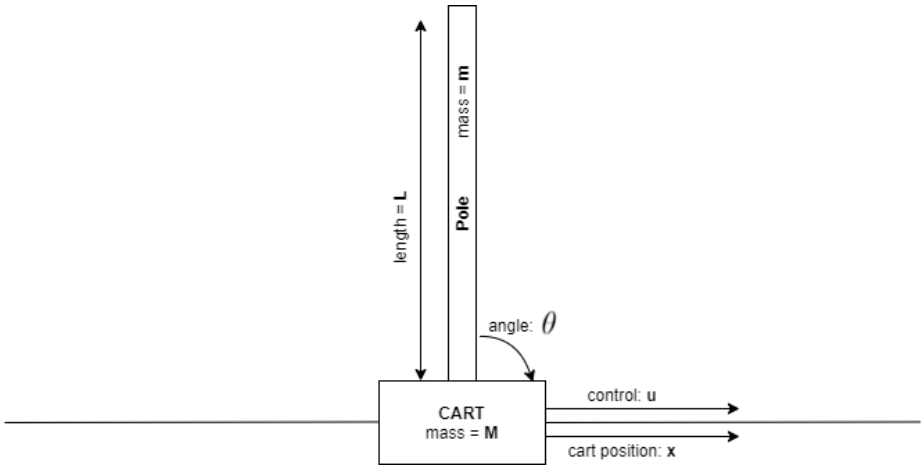


Figure 3: A diagram depicting the cartpole environment and its important parameters. The goal of the environment is to balance the pole at 90 deg to the cart.

5 Experimental Framework

In this section, we will give a description and analyze the Cartpole and Reacher gym simulation environments in which we will run these tests. Then, we will diagram and describe the methods used to generate training and evaluation trajectories, as well as the equations used to generate the evaluation metrics used.

5.1 Environment

5.1.1 Cartpole

While the Inverted Pendulum is a traditional control problem, the use of its dynamics in the cartpole environment for machine learning purposes was first introduced by Barto et al. [8], and consists of a cart on a track attempting to balance a pole. As we can see in Figure 3, the cartpole environment's important parameters include the mass of the cart M, the mass of the pole m, and the length of the pole L. The control input into this environment is a force pushing the cart u. The environment's state vector include the cart position x, the cart velocity $\dot{x}$, the pole angle θ, and the pole angular velocity $\dot{\theta}$. For our simulation, we use $M = 1kg$, $m = .1kg$, $L = 1m$, and a time step of $\tau = .02s$. These parameters lead to a time constant of $.319s$.

We will now take a look at the dynamics of the cartpole system to gain a better understanding of the unstable or stable modes of the cartpole system. We will first solve for the nonlinear dynamics of the system using Lagrangian

Dynamics. In order to do this, we first calculate the kinetic energy and potential energy of the cartpole system. The cart's kinetic energy is defined by equation (10). The pole's kinetic energy is defined by equation (11). The pole's potential energy is defined by (12). We then use $L = KE_{total} - PE_{total}$ and (13) with $q = [x, \theta]$ to calculate the nonlinear dynamics of the cartpole system (14). We can linearize these dynamics around $\begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}$, which results in the state space representation $\dot{s} = As + Bu$ where s is the state vector $\begin{bmatrix}x & \dot{x} & \theta & \dot{\theta}\end{bmatrix}$. The A and B matrices are defined by (15) and (16).

$$KE_{cart} = \frac{M\dot{x}^2}{2} \tag{10}$$

$$KE_{pole} = \frac{m(\dot{\theta}^2L^2 + 2cos(\theta)\dot{\theta}\dot{x}L + \dot{x}^2)}{2} \tag{11}$$

$$PE_{pole} = mgLcos(\theta) \tag{12}$$

$$\frac{d}{dt}(\frac{\delta L}{\delta \dot{q}}) = \frac{\delta L}{\delta q} \tag{13}$$

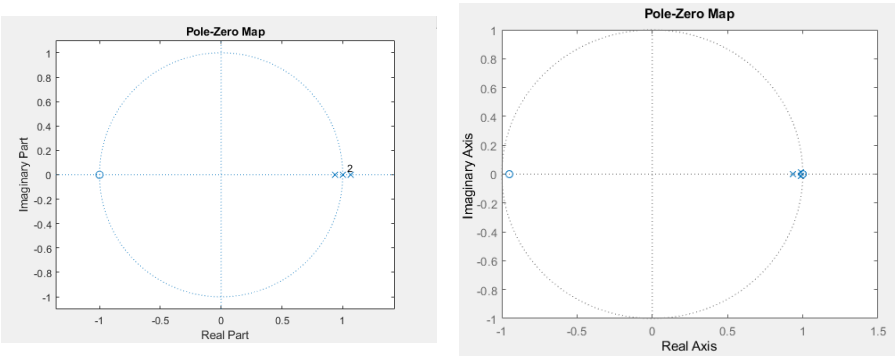
$$\begin{bmatrix}\dot{x}\\ \ddot{x}\\ \dot{\theta}\\ \ddot{\theta}\end{bmatrix} = \begin{bmatrix}\dot{x}\\ \frac{-mLsin(\theta)\dot{\theta}^2+u+mgcos(\theta)sin(\theta)}{M+m-mcos(\theta)^2}\\ \dot{\theta}\\ \frac{-mLcos(\theta)sin(theta)\dot{\theta}^2+ucos(\theta)+mgsin(\theta)+Mgsin(\theta)}{L(M+m)-mcos(\theta)^2}\end{bmatrix} \tag{14}$$

$$A = \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & 0 & \frac{mg}{M} & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & \frac{mg+Mg}{ML} & 0\end{bmatrix} \tag{15}$$

$$B = \begin{bmatrix}0\\ \frac{1}{M}\\ 0\\ \frac{1}{ML}\end{bmatrix} \tag{16}$$

We take these continuous dynamics, discretize them with zero-order hold with a time step of $\tau = .02s$, and then plot the poles and zeros using Matlab. Figure 4a displays these poles and zeros. For discrete systems, poles are unstable if they are outside of the unit circle, and are stable if inside the unit circle. In Figure 4a, we can see that the cartpole system has one stable, one unstable, and two marginally stable poles. The unstable pole corresponds to the pole angle.

We can also take a look at these poles and zeros when the discrete cartpole system is under an LQR policy. The LQR policy aims to use state feedback control to minimize the cost function J defined by (17). This is done by solving the Algebraic Ricatti equation (20) for the matrix P. Then, we use (19) to get the state feedback gain K, which gives our state feedback input u from (18). For our purposes we use a Q matrix that is a diagonal matrix where the diagonal is $\begin{bmatrix}.5 & .05 & 1 & .05\end{bmatrix}$ and $R = 1$. Using matlab, we can generate a pole zero



(a) Open loop

(b) LQR closed loop control

Figure 4: Maps of the poles and zeros of the discretized cartpole system, generated with Matlab. The left is the open loop system, and the right is under an LQR control policy.

plot of the system under this control policy. Figure 4b shows us that with LQR control, we stabilize all poles, as in all poles are now within the unit circle. This essentially tells us that the LQR policy balances the pole on top of the cart. We can see from the graph in Figure 5a that both the pole angle and cart position are stabilized by the LQR policy.

$$J = \int_0^\infty (s^T Q s + u^T R u) \tag{17}$$

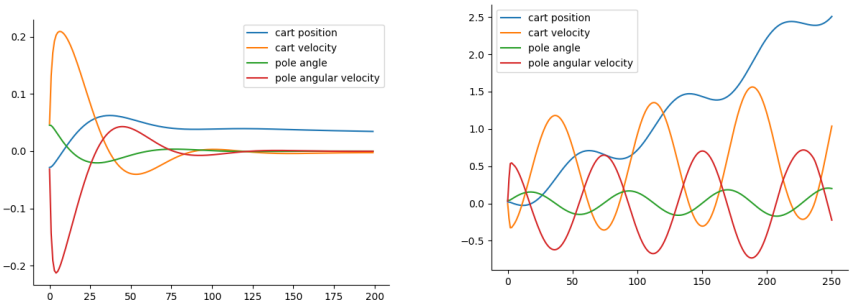
$$u = -Ks \tag{18}$$

$$K = R^{-1}(B^T P) \tag{19}$$

$$0 = A^T P + PA - (PB)R^{-1}(B^T P) + Q \tag{20}$$

5.1.2 Reacher

The Reacher environment is a five-jointed arm moving in a 3d space. It is reaching towards a 3d coordinate goal $g \in \mathbb{R}^3$. The states are determined by the joint angles $\theta \in \mathbb{R}^5$ and the target goal $g \in \mathbb{R}^3$: cosines of the joint angles $\cos(\theta)$, sines of the joint angles $\sin(\theta)$, the 3d coordinates of the goal g, the velocities of the joint angles $\dot{\theta}$, and the difference between the goal and current coordinates $g - x$ where x is the current 3d coordinate position. The actions of this environment are $a \in \mathbb{R}^5$ and represent the torques of the 5 joint angles. We use a PID control policy as one of the tests in this environment, which uses the difference between the target goal and the current position to do feedback control on the five joint angle torques.



(a) LQR Control Policy

(b) CMA-ES optimized neural network policy with default cartpole reward

Figure 5: Above are two plots showing the states of the cartpole system under different control policies. The left shows an LQR control policy. The right shows a neural network policy optimized with CMA-ES with the default cartpole reward, which results in an unstable trajectory.

5.2 System for Training and Evaluating the Dynamics Network

In this section we will be detailing the system used to generate stable trajectories for the training and test set. Then, we describe the system used for generating the evaluation metrics used. There are two key neural networks maintained in these systems: one neural network control policy (depicted in blue in Figure 6), and one dynamics model (depicted in green in Figure 7). The neural network control policy $\pi_\theta(a_t|s_t)$ is used to get actions a_t for our trajectory. The dynamics model is the model used to predict the dynamics and is a trajectory-based model as depicted in Figure 2.

5.2.1 Generating Stable Trajectories

We want to use asymptotically stable trajectories, example shown in Figure 5a, to train the dynamics model. This is because the space of all stable (especially asymptotically stable) trajectories have lower variance than the space of all trajectories. There is an argument to be made that constraining the space of trajectories like this makes the model ineffective for MPC control policy training like in the PETS algorithm [1] due to the distributional shift issue. The dynamics model will be trained on only stable trajectories, and will be inaccurate when predicting unstable trajectories. However, the purpose of this experiment is to determine the prediction accuracy of the trajectory-based model when looking at trajectories run using complex neural network control policies compared to trajectories run using LQR control policies. Since LQR control policies are inherently stable as shown in Figure 5a, the most fair way to make a comparison is to train and predict on stable trajectories from neural network policies. It

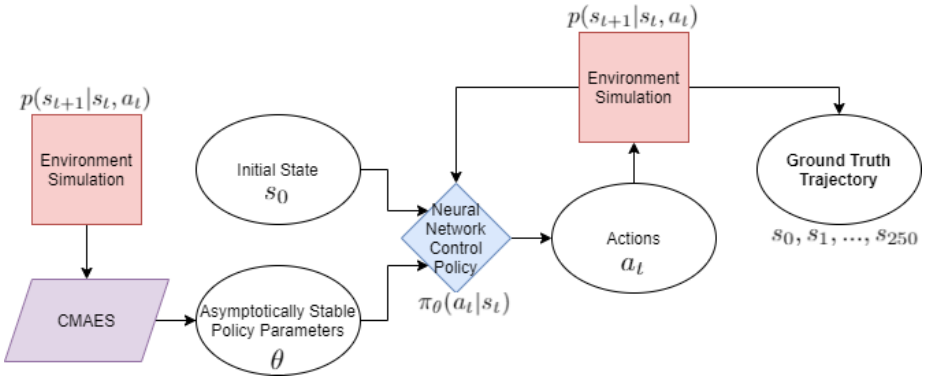


Figure 6: A flow chart showing the process we use to generate stable trajectories with neural network control policies to train and evaluate the trajectory-based dynamics model.

should be noted, however, that because of the above reason, the prediction accuracy of the trajectory-based model when trained and evaluated on both unstable and stable trajectories should be looked into, and will be discussed later, though it is not a central point of this paper.

We find neural network control policy parameters that lead to asymptotically stable trajectories by using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [14]. This is a method commonly used to optimize non-linear non-convex objective functions. It employs an evolutionary strategy, where each iteration "parents" generate "children" stochastically (the parent vector plus some random vector perturbation). Then, we select the children that become the new parents based on which child produced the best values for the objective function. In doing so, with each iteration, we move closer to an optimal objective function value. In the case of our problem, we are trying to find policy weight and bias parameters θ that will lead to a high cumulative reward. Since this objective function includes optimizing through a neural network, it is highly non-convex and non-linear, which is why we choose to use CMA-ES for this problem. The CMA-ES optimizer uses the environment simulation dynamics directly to find asymptotically stable policy parameters θ. At each iteration, from the "parent" it generates random "child" weight and bias parameters θ_{rand}, then run a simulation with that control policy $\pi_{\theta_{rand}}(a_t|s_t)$, and selects the best the cumulative reward to be the new parents. After we get the asymptotically stable policy parameters θ_{opt}, we use these as the weight and bias parameters for the neural network control policy $\pi_{\theta_{opt}}$, then run through the environment simulation to generate a trajectory. We then save pairs of the parameters θ_{opt} and the states of the trajectory. We do this for multiple trajectories, which are then split up into training sets and test sets. For each dynamics model we train and evaluate, we have 100 different policies generating

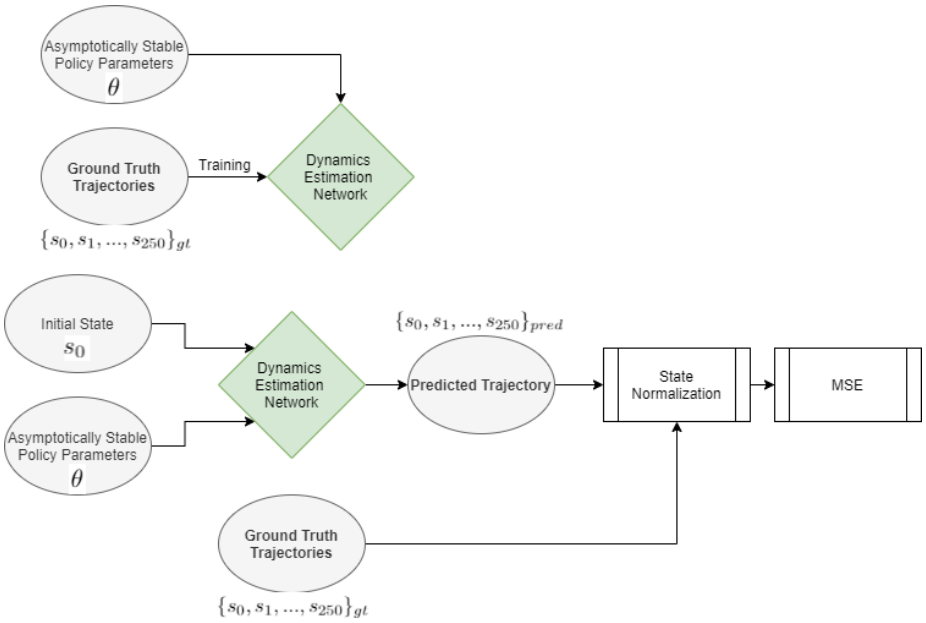


Figure 7: A flow chart showing the process we use to evaluate the prediction accuracy of the Dynamics Model Neural Network. We predict trajectories using the trained dynamics model, normalize the states, then calculate the MSE.

100 different trajectories in the training set and similarly 100 policies for 100 trajectories in the evaluation set.

The choice of reward function used for the objective function in the CMA-ES optimizer is important. Initial tests were done using the default cartpole reward, which is simply equal to the number of time steps before the simulation ends. The simulation ends when the cart fails to balance the pole. Specifically, it is when the pole angle exceeds 24 deg, or the cart position moves more than $9.6m$. So, if the pole angle and cart position stay within that threshold for the full run (250 time steps), then the objective function would be maximized at 250. However, this does not generate asymptotically stable trajectories due to plateaus in the reward surface. It only guarantees trajectories that are within a threshold for the full run of 250 time steps. The state plot of a trajectory generated using a control policy optimized with this reward function is shown in Figure 5b. We changed this reward function to be the LQR cost function (17), where Q is $\begin{bmatrix}.5 & .05 & 1 & .05\end{bmatrix}$ and $R = 1$.

5.2.2 Training and Evaluating the Dynamics Model

As we can see from Figure 2, we use control policy parameters θ and an initial state s_0 as inputs into the network. From the last step, we have saved pairs

of control policy parameters θ and a list of states for the trajectory generated using the corresponding control policy $\pi_\theta(a_t|s_t)$. We train the dynamics model by inputting the first state (the initial state s_0) for the trajectory, control policy parameters θ, and varying the time index t from 1 to the horizon, for this experiment, 250. The target output we use to train the model against is the rest of the trajectory from s_1 to s_{250}.

As seen from Figure 7, we take a trained dynamics network and evaluate against a test set to get the prediction accuracy in the form of a mean-squared error. We use a similar process as the training process, except with a test set. We use the asymptotically stable policy parameters θ and the initial state s_0 from the corresponding trajectory. We vary the time index t from 1 to 250 to generate a predicted trajectory from s_1 to s_{250}. We then compare this against the ground truth trajectory. For a fair comparison where one particular state is not weighted more than the other, we normalize the state vectors in the predicted and ground truth trajectories between 0 and 1. We then use the standard MSE formula shown in (21), where N is the number of trajectories we have evaluated. We take the mean with respect to all N trajectories evaluated. In (21), $s_{gt,i}$ is the ground truth state of the ith trajectory, and $s_{pred,i}$ is the predicted state of the ith trajectory. In the end, we get a vector of length 250, which includes the MSE at each time step. Included in most of our result graphs are also errorbars depicting the 66th percentile of errors amongst all trajectories (transparent highlighted area), and a baseline showing MSE prediction accuracy compared to a naive prediction of all 0s (solid line with no markings).

$$MSE_t = \frac{1}{N}\sum_{i=1}^{n}\|(s_{gt,i} - s_{pred,i})\|^2 \tag{21}$$

5.3 Problem Formulation

In the following results, we will be comparing the MSE prediction accuracy of the dynamics model with various control policies. We will test LQR policies in the Cartpole environment, PID policies in the Reacher environment, and neural network policies of varying depths.

Type	Number of Parameters	MSE
One-step	-	3.97e11
Trajectory-based with NN control	61	1.30
LQR	4	0.0125
LQR(train)	4	0.0132

Table 1: Table detailing the MSE of four different evaluations. The trajectory-based models (rows 2 through 4) perform much better than the one-step, which suffers from compounding error. The two LQR policy trained models (one evaluated on a test set and the other the training set) perform better than the neural network policy trained models.

6 Results and Discussion

6.1 Comparison of One-step, LQR, and Neural Network Control Policies

We first take a look at the MSE prediction accuracy of the one-step network (with delta state $s_{t+1} - s_t$ outputs), the trajectory-based network with LQR control policy, and the trajectory-based network with a neural network control policy (1 hidden layer, width 5). Note that for the one-step network, we directly fed each next-state back into the input, which led to serious compounding error. The results are listed in Table 1. The one-step models performed the worst due to compounding error. In Figure 8, we can see that after time step 50, the network reached an unexplored part of the state transition dynamics (the network was not trained for this part of the environment), and began giving outputs which were wildly inaccurate. Notice that this is not an issue with the trajectory-based models. However, the less complex LQR policy with less policy parameters not only had a lower MSE error, but also did not vary greatly between the evaluations on the training set and the test set, which suggests no overfitting. As we will see in later sections, overfitting is an issue with more complex neural network policies. It is likely not an issue with the LQR policy because it only has 4 parameters, and therefore varies less, and the training set of 100 trajectories can more accurately capture the entire space of trajectories and control policy parameter inputs.

6.2 Neural Network Control Policies of Varying Depths

We now take a look at the MSE prediction accuracy when varying the number of layers in the neural network control policy. More layers in the policy means more weight and bias parameters to be input into the dynamics model. For these networks, we use neural network control policies of hidden layer width 5. As we can see from both Figure 9 and Figure 10, there is a general trend of more policy parameters leading to more error. Another important thing to

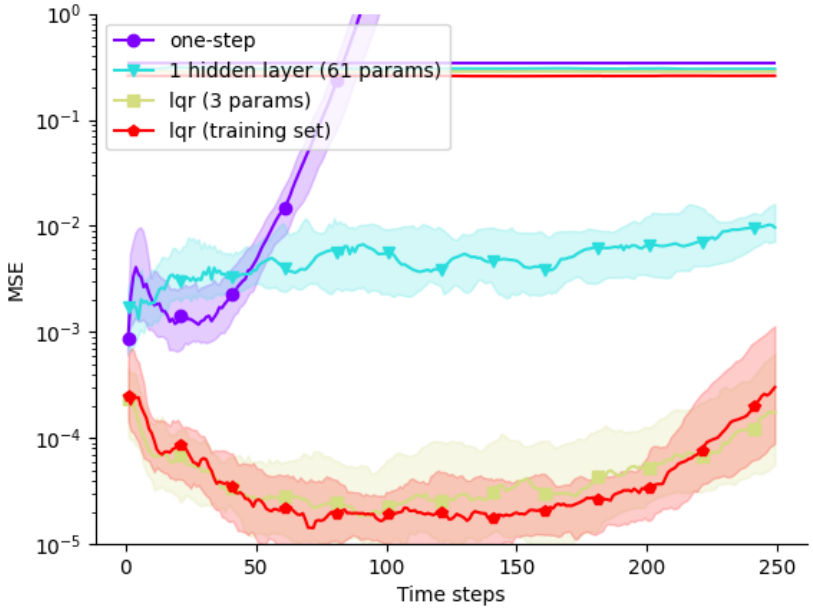


Figure 8: A plot comparing the MSEs of the One-step dynamics model, the trajectory-based dynamics model with an LQR control policy (both evaluated on test and training sets), and the trajectory-based dynamics model with a NN control policy (1 hidden layer, width 5). The one-step model's error compounds after 50 time steps. Both LQR policy models perform better than the neural network policy model. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

Network Depth	Number of Parameters	Evaluation Set	MSE
0	31	Test	0.839
1	61	Test	1.30
3	121	Test	1.54
5	181	Test	1.32
0	31	Training	0.00820
1	61	Training	0.00654
3	121	Training	0.00978
5	181	Training	0.0177

Table 2: The table details the MSE results from neural network policies of 4 different depths. We see a general trend that more parameters used (the more layers used), the higher the MSE.

note is the difference between evaluations on the training set (Figure 10) and evaluations on the test set (Figure 9). This big difference suggests overfitting to the training set. We hypothesize that this is because the training set did not accurately capture all the different types of trajectories there could be. This could be because with neural network control policies, there is a higher variance due to the higher number of input parameters. This issue could be potentially alleviated by training on a higher number of trajectories (1000 instead of 100). These same trends can be seen in Chart 2, which show the MSEs summed across the 250 time steps.

6.3 Testing with Control Policy Inputs as Zero

We now look a different type of evaluation where we zero out the θ control policy parameters before inputting them into the dynamics model. This is so we can see exactly how much the network is learning from the policy. As we can see in Figure 11 and Chart 3, the model learns more from the policy parameters the less policy parameters there are. For the LQR policy (with only 4 policy parameters), the MSE drastically increases when zeroing out the policy parameters θ. For the 0 hidden layer neural network (with 31 policy parameters), it made a minimal difference. For the 5 hidden layer neural network (with 181 policy parameters), it made no difference at all. Therefore, we can conclude that the more complex the control policy we use, the less the dynamics model itself learns from the control policy parameters θ.

6.4 Reacher Environment

6.4.1 Reacher: Comparison of One-step, PID, and Neural Network Control Policies

For the Reacher environment, we see in Figure 12 and Table 4 that the one-step model, similar to the Cartpole environment, performs worse than both

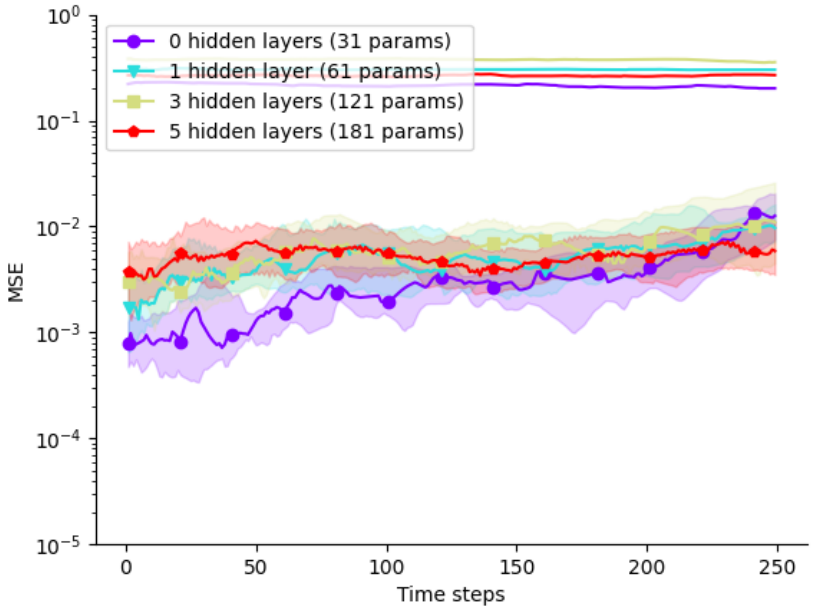


Figure 9: A plot comparing the MSEs of different neural network control policy depths in the Cartpole environment. This evaluation is done on test sets. We can see the general trend that more parameters means higher MSE. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

Policy Type	No-Policy	Number of Parameters	MSE
LQR	False	4	0.0125
LQR	True	4	0.478
NN 0 hidden layers	False	31	0.839
NN 0 hidden layers	True	31	0.965
NN 5 hidden layers	False	181	1.32
NN 5 hidden layers	True	181	1.25

Table 3: The table details the tests with three different policies: LQR, neural network with 0 hidden layers, and neural network with 5 hidden layers. We do tests with the control policy parameters zeroed out. We can see that the difference between the "no-policy" and normal tests are biggest in the LQR tests.

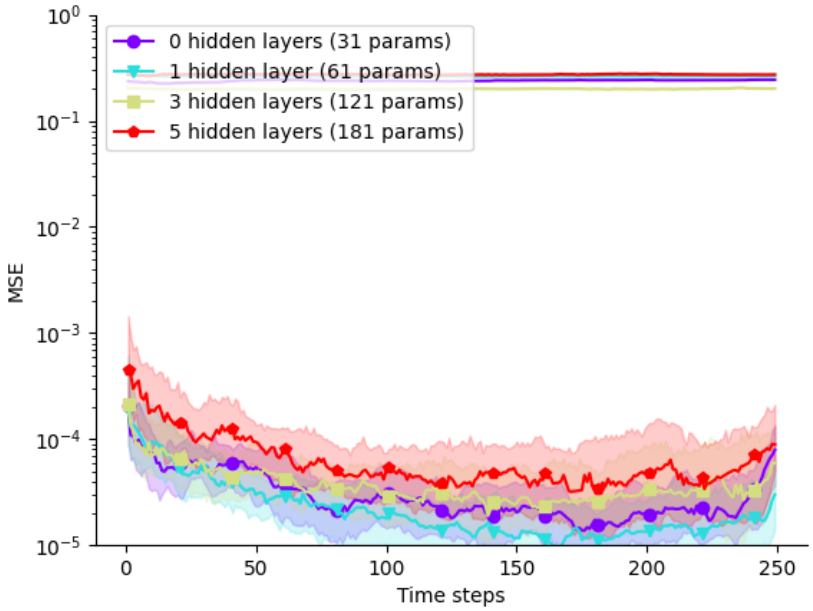


Figure 10: A plot comparing the MSEs of different neural network control policy depths in the Cartpole environment. This evaluation is done on the original training set. We can see the general trend that more parameters means higher MSE. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

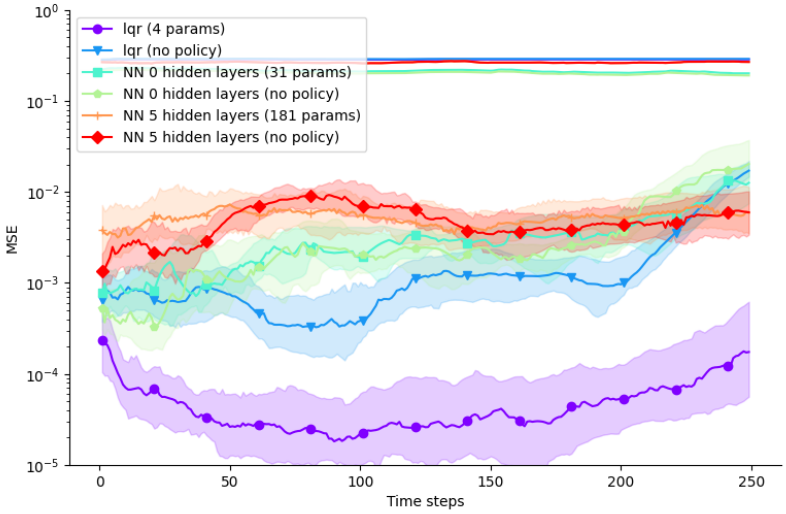


Figure 11: This graph shows the LQR policy, the NN policy with 0 hidden layers, and the NN policy with 5 hidden layers with the policy parameter inputs zeroed out. We can see that the dynamics model trained on the LQR policy suffers more from losing the control policy parameter inputs. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

Policy Type	Number of Parameters	MSE
One-step	-	144
Trajectory-based with NN control	146	11.8
PID	15	23.7

Table 4: The table above displays the MSE results of evaluations on a one-step dynamics model, a trajectory-based model trained on trajectories with neural network control policies, and one with PID control policies. We can see that the two trajectory-based model have better accuracy, and that the neural network control policy is slightly more accurate.

trajectory-based models. However, unlike the Cartpole environment, the simple PID control policy performs worse than the neural network control policy. We can see, however, from the baselines, that the small error between the two policies could be attributed to differences in the ground truth trajectories used to train them. It seems that with the PID policy generated ground truth trajectories that were further away from the environment origin than the neural network control policy. To discover why this is would require further looks into the workings of the PID policy in the Reacher environment, and is left for future work.

6.4.2 Reacher: Neural Network Control Policies of Varying Depths

For the Reacher environment, we see from Figure 13 and Table 5 that there is still a general trend that a higher number parameters means lower prediction accuracy. There is also still a drastic difference between the training set and test set, suggesting overfitting. However, we see in Figure 14, for the training set evaluations, there is no prediction accuracy trend present whatsoever. We hypothesize that overfitting to the training trajectories set has caused this, and further work into fixing the overfitting issue is needed to look into this.

6.4.3 Reacher: Testing with Control Policy Inputs as Zero

We see in the Reacher environment "no-policy" tests the same trends we saw in the Cartpole environment. In Figure 15 and Table 6 that there is a big difference between "no-policy" and original evaluations for the PID policy, but less of a difference the more parameters we add to the control policy. This suggests that the dynamics model has a more difficult time learning from the control policy parameters the more parameters there are.

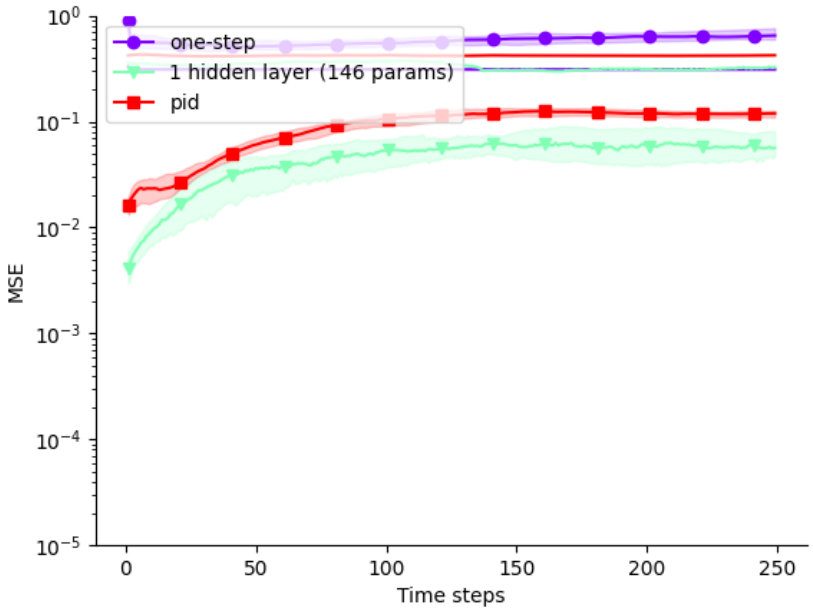


Figure 12: This graph shows the one-step dynamics model, the trajectory-based dynamics model with a neural network control policy, and with a PID control policy on the Reacher environment. The one-step model still suffers from compounding error, while the trajectory-based models perform better. The PID policy, however, ends up worse than the neural network control policy. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

Network Depth	Number of Parameters	Evaluation Set	MSE
0	116	Test	10.6
1	146	Test	11.8
3	206	Test	14.9
5	266	Test	13.3
0	116	Training	0.144
1	146	Training	0.103
3	206	Training	0.0989
5	266	Training	0.123

Table 5: The table above shows results for evaluations of a trajectory-based dynamics model with control policies of varying depths on the Reacher environment. We can see that, while for the evaluations on a test set, the accuracy generally decreases with the number of parameters, this is not true for the evaluations on a training set.

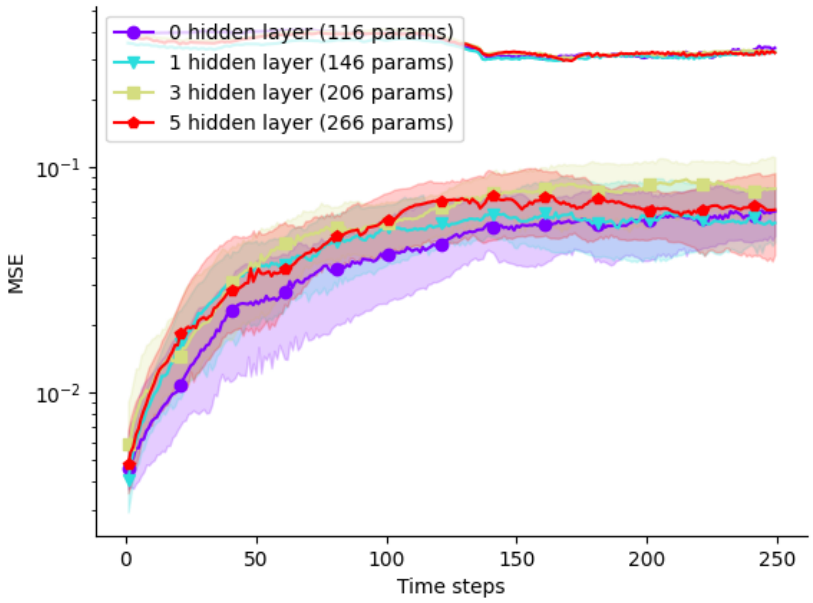


Figure 13: The graph above shows results for evaluations of a trajectory-based dynamics model with control policies of varying depths on the Reacher environment. We can see that there is a general trend that the accuracy decreases when increasing the number of parameters. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

Policy Type	No-Policy	Number of Parameters	MSE
PID	False	15	23.7
PID	True	15	32.3
NN 0 hidden layers	False	116	10.6
NN 0 hidden layers	True	116	13.2
NN 5 hidden layers	False	266	13.3
NN 5 hidden layers	True	266	13.7

Table 6: The table above shows the the PID policy, the 0 hidden layer neural network policy, and the 5 hidden layer neural network along with tests where the control policy parameter inputs into the trajectory-based dynamics model are zeroed out. We can see the biggest difference between the "no-policy" and original evaluations for the PID policy compared to the other policies.

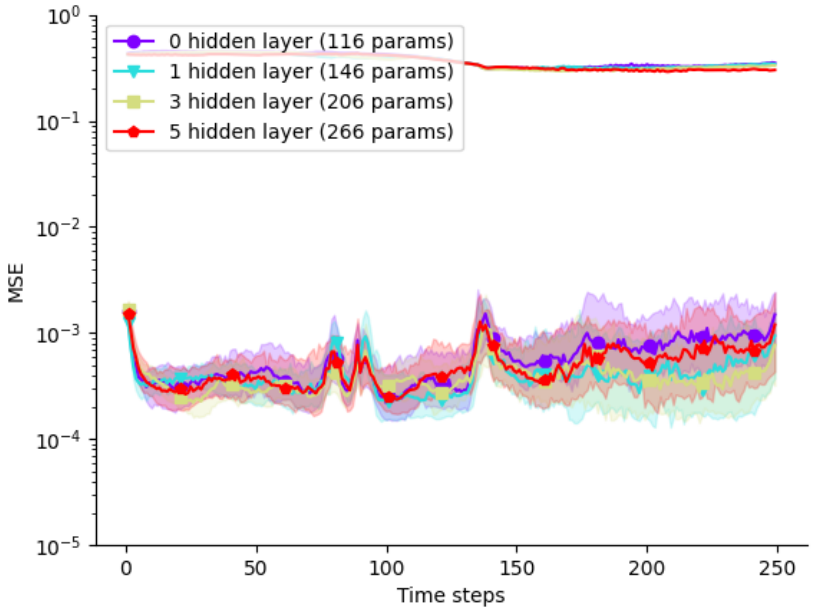


Figure 14: The graph above shows results for evaluations of a trajectory-based dynamics model with control policies of varying depths on the Reacher environment. These evaluations are done on the original testing sets. There is no general trend in the training set evaluations. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

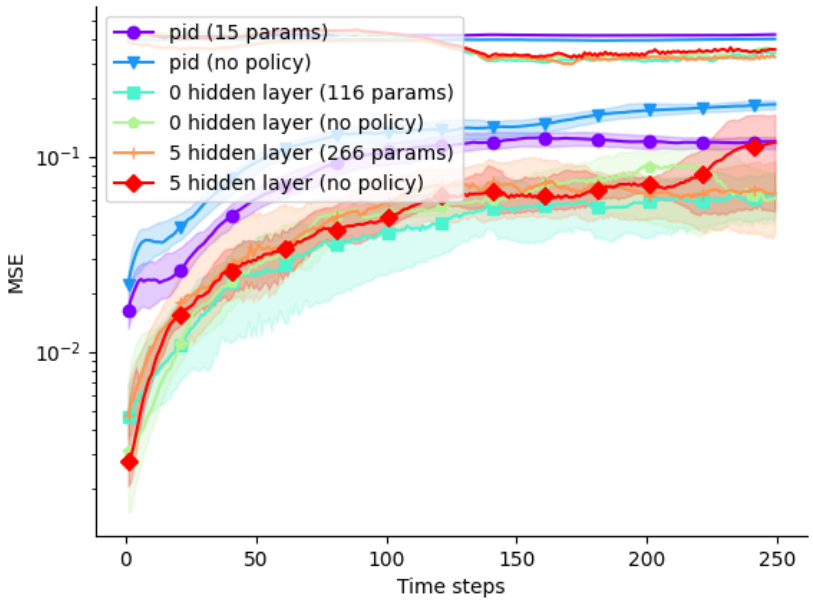


Figure 15: The graph above shows the the PID policy, the 0 hidden layer neural network policy, and the 5 hidden layer neural network policy. It also includes tests with the control policy parameter inputs into the trajectory-based dynamics model zeroed out. We can see the biggest difference between the "no-policy" and original evaluations for the PID policy compared to the other policies. We have also included in this graph 66th percentile errors as well as baseline predictions (solid lines).

7 Future Work

7.1 Overfitting

Due to the length of the research period, there are still many avenues to go down to discover more about the workings of the trajectory-based model and how it interacts with control policies with a high-dimensional parameter space. One particularly interesting one is the overfitting issue. It seems from this research that there is a positive correlation between the number of parameters used in a control policy and the variance of the trajectories generated by that control policy. This makes sense because the higher the dimension of the control policy parameter space, the more the control policy can vary for each trajectory. Higher variance training trajectories can lead to overfitting to the training set and loss of generalizability to a test set. This issue can be alleviated with the use of more training trajectories, or perhaps ensemble training, which is the training of multiple dynamics models to average together their outputs for a non-biased, lower-variance output.

7.2 Testing with Control Policies with More Parameters

Due to the time constraints of generating these trajectories, we were unable to run tests with massive neural networks, with over thousands of parameters. However, pushing this boundary could be very useful in solidifying our hypotheses on the prediction accuracy of the dynamics model. We believe that more parameters means less prediction accuracy, due to the dynamics model having difficulty learning from the control policy if it has too many parameters. Training a dynamics model with trajectories with control policies that have thousands of parameters could show us if this trend continues.

7.3 Uncertainty

Similar to the PETS algorithm [1], we want to test these dynamics models with the added benefit of uncertainty-aware predictions. We can account for the aleatoric uncertainty of the environment by using probabilistic loss functions for the trajectory-based dynamics model. We can account for the epistemic uncertainty of the model itself by implementing ensemble training, where we train multiple dynamics models, run an input through all of the models, and average together the outputs to get the final uncertainty-aware output. We would like to test the prediction accuracy of these uncertainty aware models on trajectories with neural network control policies with many parameters.

7.4 MBRL

The end goal of this research is to apply the trajectory-based model to model-based reinforcement learning algorithms. We could use a system similar to the one described by Chua et al. [1]. We use a trained dynamics model to simulate

the dynamics of the environment for an optimizer to determine the best control policy in an MPC algorithm. Then, we can test this optimized control policy to see how it compares to other MBRL algorithms such as PILCO [2] or a similar MPC algorithm with the one-step model. We can test this MPC approach with re-optimizing the control policy at each time step, and re-training the model after each trajectory similar to PETS [1]. This paper looks into the prediction accuracy of the trajectory-based model on trajectories using neural network control policies. This is necessary for future applications in MBRL, since we would like to optimize for expressive neural network control policies in these algorithms, rather than only simple (low number of parameters) control policies like LQR or PID.

References

[1] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Neural Information Processing Systems*, pp. 4754–4765. 2018.

[2] M. P. Deisenroth and C. E. Rasmussen, "PILCO: A Model-Based and Data-Efficient Approach to Policy Search," in *International Conference on Machine Learning*, pp. 465–472. 2011.

[3] Lambert, N.O., Wilcox, A., Zhang, H., Pister, K.S. and Calandra, R., Learning Accurate Long-term Dynamics for Model-based Reinforcement Learning. *arXiv preprint arXiv:2012.09156*. 2020.

[4] N. Lambert, B. Amos, O. Yadan, and R. Calandra, "Objective mismatch in model-based reinforcement learning," *arXiv preprint arXiv:2002.04523*, 2020.

[5] C. Xiao, Y. Wu, C. Ma, D. Schuurmans, and M. Muller, "Learning to ¨combat compounding-error in model-based reinforcement learning," *arXiv preprint arXiv:1912.11206*, 2019.

[6] K. Asadi, D. Misra, S. Kim, and M. L. Littman, "Combating the compounding-error problem with a multi-step model," *arXiv preprint arXiv:1905.13320*, 2019.

[7] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[8] A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," in *IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, no. 5*, pp. 834-846, Sept.-Oct. 1983.

[9] Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S., Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pp. 1861-1870. PMLR. 2018.

[10] Gu, Shixiang, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. "Continuous deep q-learning with model-based acceleration." In *International Conference on Machine Learning*, pp. 2829-2838. PMLR, 2016.

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

[12] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust region policy optimization". In *CoRR, abs/1502.05477*. 2015.

[13] Z. I. Botev, D. P. Kroese, R. Y. Rubinstein, and P. L'Ecuyer. The cross-entropy method for optimization. In *Handbook of statistics*, volume 31, pages 35–59. Elsevier, 2013

[14] Hansen, Nikolaus. "The CMA evolution strategy: A tutorial." *arXiv preprint arXiv:1604.00772*. 2016.