Exploring Problems and Solutions in Model-Based Reinforcement Learning: Final Report

Thanard Kurutach, Yue (Andy) Zhang
CS 294-112

May 2017

1 Abstract

It has been a challenge to make model-based reinforcement learning (MBRL) worked in many applications. In this project, we investigate different MBRL problems such as overfitting to poor dynamics and avoiding local minima using Mujoco simulators [5] and OpenAI gym [1]. We begin our investigation by considering the setting where computing policy gradients from the true dynamics. Then, we transition to an oracle approach where we use the gradient from a learned dynamics model, but periodically check the true performance with the true dynamics. Finally, we move away from the true dynamics completely and use a dynamics validation model to stop the policy updates. We show that our proposed solution is promising compared to our baseline which is a vanilla model-based approach.

2 Introduction

One of the main challenges in reinforcement learning is it often requires a large number of samples in order to solve a task. In many domains where real-world data collection is expensive, e.g.robotics, collecting millions of samples is impractical. Using model-based reinforcement learning approach, we can extract useful information from the data much more efficiently, and make it possible to learn from pretrained dynamics model rather than having to start from scratch. The general framework of model-based rl is to first perform some roll-outs using an initial policy. Then, we use collected data to train a dynamics model. Then, given the trained dynamics model, we can compute the estimated policy gradients and update the policy. Then, we go to the next iteration which is collecting data from the current policy. We repeat until the task is learned.

However, in practice, it is difficult to know when to stop updating the policy and continue the next iteration of collecting data, training the dynamics and updating the policy again. If we make too many updates on the policy, then it can suffer from overfitting it. If we make too few updates then the task is never learned. This problem is also known as model-bias in reinforcement learning literature. In this project, we illustrate this problem in continuous control domain, we show that stopping at the right time is the key to make it work, and we propose a solution to the problem by bootstrapping dynamics models.

In this project, we use both deterministic policy and dynamics models, which are represented by feed-forward neural networks. We train the dynamics model using simple supervised-learning, and train the policy model using backpropagation through time. The training data fed to the dynamics model is a set of triplets of state, action, and next state (s, a, s'). The training data fed to the policy model is random initial states. We choose pendulum and swimmer environments in Mujoco.

In the first experiment, we assume the through dynamics in pendulum and we backpropagate through time to optimize the policy. We find 3 local optima in this setting. In the other experiments described below, we avoid local optima by initialize the policy with some structure from the expert.

We design our second experiment to identify overfitting issue. In this experiment, we use a vanilla model-based-rl approach. After each policy update, we compute the policy trajectory cost on the predefined validation initial states. If the cost goes, then we stop and move on to the next iteration. We find that training the policy using just the estimated validation cost, the policy overfits and exploits the learned dynamics in almost every iteration.

In the third experiment, we show that using backpropagation through time to solve model-based rl can potentially work if we can stop the updates at the right time. While we are still computing the policy gradients based on the learned dynamics, we allow an oracle access to the true dynamics in order to compute the true policy cost from the predefined validation initial states and suggest when stop training. Note that this setting works well, but it is not practical.

In the last experiment, we move away from the true dyanmics by learning two dynamics models. The first one is used to compute the policy gradient the same way as previous two experiments. The other is used as a proxy for the oracle performance. Unfortunately, this is not yet working robustly. Our future work is to bootstrap multiple training and validation dynamics models.

3 Related Work

Deisenroth and Rasmussen [2], [3] have investigated the model-bias problem in 2011 and proposed a probabilistic approach using gaussian process to represent dynamics. Even though this approach can maintain uncertainty through time, the main draw back is that the computation prohibitively slow and even slower than real-world data collections. Also, it requires a restrictive form of cost function and a lot of approximations in one-step prediction in order to keep the posterior computation tractable.

A similar line of work by Depeweg et al. uses a Bayesian neural network to maintain dynamics uncertainty. However, the approach is complicated, and only shown to work with batch RL problems, which avoids doing exploration.

4 Methods

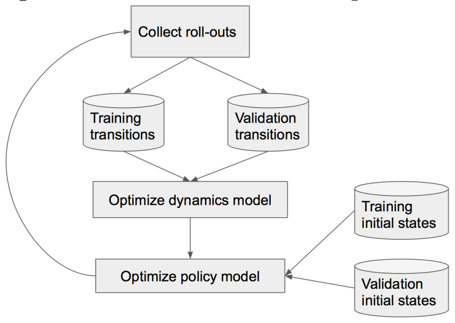
We consider a standard model-based reinforcement learning procedure. We initialize a policy $\pi_0(x_t) = u_t$. Using this policy, rollouts are collected from the environment, in the form $D = \{(x_t, u_t, x_{t+1})|t = 0, ..., n\}$. The set of rollouts is used to learn one or more dynamics functions, $f_i(x_t, u_t) = x_{t+1}$, and using these dynamics models, a new policy π_1 is trained until some stopping condition. This process of rollouts, dynamics learning, and policy learning is repeated for one or more iterations, until a sufficiently well-performing policy is learned for the task.

We compare 3 main variants of this model-based reinforcement learning procedure. In addition to the three model-based variants, we also explored problems in a simpler setting: giving access to true dynamics during policy training, rather than using a learned dynamics function.

4.1 Vanilla Model-Based

The vanilla model-based procedure directly uses the process described above. Rollouts are partitioned into training and validation transitions to train the dynamics model. Policy training is performed on a set of training initial states, and validated against a set of validation initial states. We move on to the next iteration of rollout collection when policy training converges.

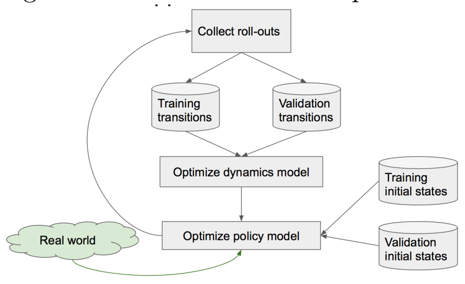
Figure 1: Vanilla model-based procedure



4.2 Oracle Model-Based

In this variant, we allow oracle access to the true environment/dynamics for policy validation. During policy training, a policy is trained on the learned dynamics, but validated against the true dynamics. This allows the policy training to distinguish when it is making policy improvements, and when it is overfitting to the learned dynamics model, not making policy improvements, and should move on to the next iteration of rollouts collection.

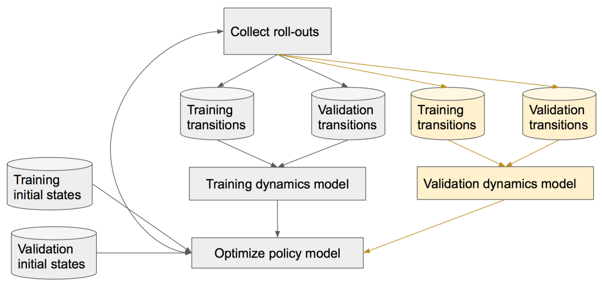
Figure 2: Oracle model-based procedure



4.3 Double Validation Model-Based

In the double validation variant of model-based reinforcement learning, we do not allow oracle access. Instead, multiple dynamics models are learned, each by subsampling transitions from the rollouts. Some dynamics models are held as validation dynamics models. The validation dynamics models serves as a proxy for oracle access – the policy is trained with the training dynamics, and the validation dynamics are used to identify when to end policy training and move on to the next iteration of rollout collection.

Figure 3: Double-validation model-based procedure



4.4 Implementation

Dynamics and policies are modelled as feed-forward neural networks. For policy optimization, we primarily use backpropagation through time, and also compare this against trust-region policy optimization (TRPO) [4].

For backpropagation through time, we optimize over T timesteps, given some cost function $c(x_t, u_t)$, dynamics function $f(x_t, u_t)$. We optimize the sum-over-time cost of policy $pi_\theta(x_t)$ parameterized by θ:

$$\begin{aligned} \min_\theta \sum_{t=0}^{T} c(x_t, u_t) \\ \text{subject to } x_{t+1} &= f(x_t, u_t) \\ u_t &= \pi_\theta(x_t) \end{aligned}$$

5 Experiments

We used the Pendulum-v0 and Swimmer-v1 environments from OpenAI gym.

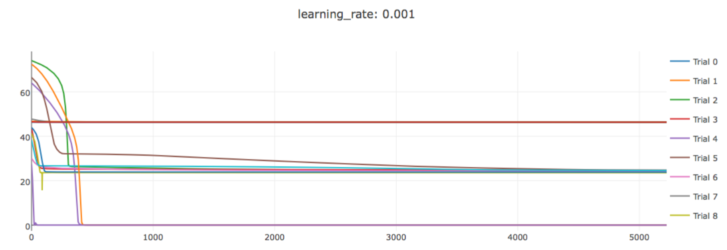
5.1 Full Access to True Dynamics

We used the pendulum environment for these experiments. The goal of this environment is to train a policy that stabilizes the pendulum at the vertical state. The true dynamics were provided during policy training. We reimplemented the true dynamics function of the pendulum environment based on the OpenAI gym implementation, so that they can be backpropagated through. The policies were modelled either as a linear policy or a feed-forward neural network policy with 2 hidden layers, each size 8.

Rather than using the cost function defined in OpenAI gym for BPTT optimization, we defined a different cost function based on the height of the end-effector, rather than the angle of the arm. The original cost function involved mod operations on the pendulum angle, which resulted in a very discontinuous cost function that is difficult to optimize over.

We compared linear and neural network policies trained to stabilize the pendulum. It turns out, the backpropagation-through-time method for this task is highly sensitive to policy weight initializations. Policies with poor weight initializations often converged to well-defined local minima during training, and resulted in policies unable to stabilize the pendulum.
Thus, the distribution of initial weights was a hyperparameter we had to tune. We compare the success rate (number of trials converging to a good policy) and final performance of linear and neural network policies trained with optimal hyperparameters found.

Figure 4: Example of policies converging to minima during policy training, plotting training loss over iterations. Some policies are quickly able to converge to a low loss (and eventually reach a good policy), but many converge to local minima at loss around 24 and 46



Pendulum policy comparison

	Linear Policy	Neural Network Policy
Success rate	8/15 trials	3/15 trials
Final training loss	0.000377	0.000377
Test reward	$-0.08224 \pm (5.6 \cdot 10^{-6})$	$-0.08224 \pm (6.7 \cdot 10^{-6})$

Linear policy hyperparameters: Weights initialized by sampling from normal distribution with $\mu = 0, \sigma^2 = 1$; learning rate 1e-2
Neural network policy hyperparameters: Weights initialized by sampling from normal distribution with $\mu = 0, \sigma^2 = 1$; learning rate 1e-3

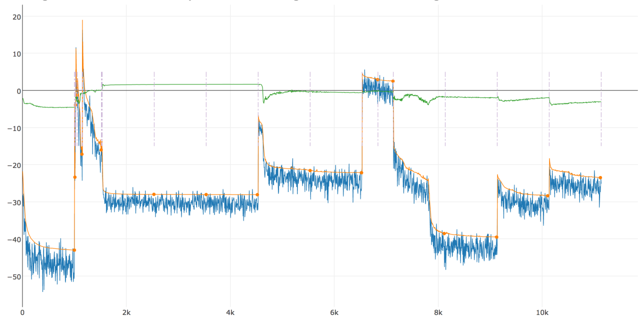
Linear and neural network policies resulted in almost identical performance for successful policies, both at training and during test time. This is not surprising since the pendulum is initialized close to the goal state, in a locally-linear region of the dynamics function. However, the rate of successful policy convergence is higher for the linear policy, which has much fewer weights, than the neural network policy.

As an example of the policy optimization's sensitivity to initial weights, we noticed that simply changing the initial weights of the linear policy as sampled from $Normal(\mu = 0, \sigma^2 = 1)$ to sampled from $Normal(\mu = 0, \sigma^2 = 0.1)$ decrease the rate of successful policy training from about 50% to 6%.

5.2 Vanilla, Oracle, and Double-Validation

In this section, we design our experiments to invest overfitting problem in model-based rl. We used swimmer environment for these experiments. In order to avoid local optima, we learned the task using TRPO and initialize the policy parameters with 0.3 times the expert parameters. We iterate between collecting roll-outs, optimizing dynamics models, and optimizing policy.

Figure 5: Policy learning curve using vanilla approach.



In roll-outs collection, we randomly perturb the policy weights by a Gaussian noise whose standard deviation proportional to the latest weight update. We also perturb predicted action by a Gaussian noise (std from 0 to 1). We add 10% of validation data cap to validation data E. If the total size exceeds validation data cap, do first-in-first-out. We add the rest of triplets to training data D (No data cap).

In dynamics optimization, we compute the gradients of the dynamics model parameters from random minibatch from dataset D, and update dynamics model parameters. We also compute validation loss on validation dataset E. When the loss increases more than 10%, we stop dynamics model updates and revert back to the minimum.

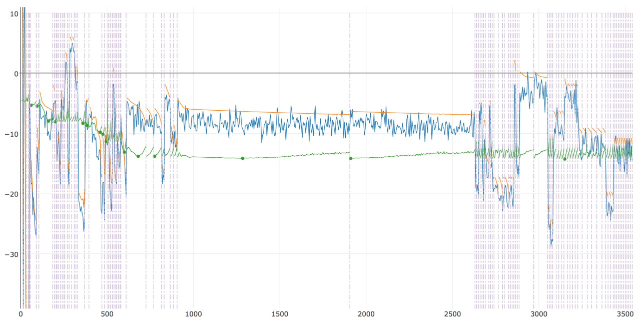
In policy optimization, we compute the gradients of the policy model parameters from random initial states under the learned dynamics model, and update policy model parameters. In the oracle case, we compute real validation cost from the same validation initial states using true dynamics. In vanilla MBRL, we compute estimated validation cost from fixed validation initial states (pre-sampled from the environment) under training dynamics model. In double-validation MBRL, we compute estimated validation cost from both training and validation dynamics models. If the validation cost increases more than 10%, we stop policy updates and revert back to the best validation cost.

The vanilla result is shown in figure 5, where the green curve is the true validation cost, orange curve is the estimated validation cost, and the blue curve is the training cost. The x-axis is the number policy updates. The y-axis is cost. The vertical lines separate between iterations. We can see that in most iterations the model is overfit when the green goes up but the orange goes down.

The oracle result is shown in figure 6 with the same legends as figure 5. We can see that The final performance is about -15, which is the best it can get using TRPO.

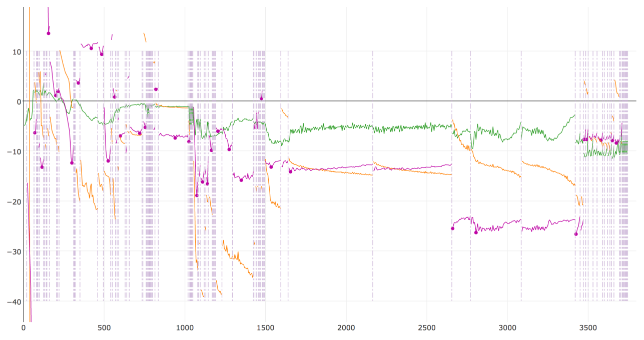
Double-validation result is shown in figure 7. Purple segments are the validation cost on a validation dynamics model. We can see that in some iterations when the policy is trying to overfit. The validation cost from the validation model help prevent that and can revert back the overfitting updates. Unfortu-

Figure 6: Policy learning curve using oracle approach.



nately this works one out of five seeds. We are still exploring ways to make this work more reliably.

Figure 7: Policy learning curve using double validation approach.



6 Conclusion

In our research, several problems encountered during model-based reinforcement learning are addressed. We observed that even when given access to true dynamics, policy optimization using backpropogation-through-time is very sensitive to how policy weight are initialized. For simple domains, it was sufficient to rely on multiple policy weight initializations, and hand-tune weight initialization distributions to reduce the chance of converging to local minima. However, for more complex domains, we recognize that more sophisticated methods are necessary, such as cross-entropy methods on the initial weight distribution.

For overfitting problems, we are looking into using bootstrapped training and validation models to better estimate gradients and stop policy learning.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[3] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.

[4] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

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