Improving model-based RL with Adaptive Rollout using Uncertainty Estimation

Nhat M. Nguyen
University of Alberta
Alberta T6G 2R3
`nmnguyen@ualberta.ca`

Abhineet Singh
University of Alberta
Alberta T6G 2R3
`asingh1@ualberta.ca`

Kenneth Tran
Microsoft Research
WA 98052, USA
`ktran@microsoft.com`

Abstract

Recently, incorporating a learned dynamic model in generating imagined data has been shown to be an effective way to reduce sample-complexity of model-free RL. Such model-free/model-based hybrid approaches usually require rolling out the dynamic model a fixed number of steps into the future. We argue that such fixed rollout is problematic for several reasons. We propose a simple adaptive rollout algorithm to improve the model-based component of these approaches and conduct experiment on CartPole task to evaluate the effects of adaptive rollout.

1 Introduction

Reinforcement learning (RL) is a general framework where an agent interacts with the environment to solve specific tasks through trial-and-error. There are two prominent paradigms in RL: model-free and model-based RL. They both have distinct strengths and weaknesses.

Model-free RL learns the state/state-action values only from the rewards and does not explicitly exploit the rich information underlying the dynamic transitions data. Model-free RL algorithms have been shown to achieve impressive feats in many high dimensional problems [17, 19]. Despite their recent successes, one main limitation of model-free RL algorithms is that they often require massive amounts of data samples for training. For example, simplest RL tasks like mountain-car or cart-pole usually require tens or hundreds of episodes to learn. This data-inefficiency problem makes it almost impossible for these algorithms to be as effective in control/robotic domains where the costs associated with real system interactions are high.

In contrast to model-free RL, model-based algorithms require models of the underlying system dynamics to be learned. Theoretically, when a good model is available, the agent can use it as a simulator to generate future observations and thus accelerate learning without having to interact with the real system as much. This leads to much better sample-efficiency than model-free RL. In practice, however, the assumption about the accuracy of the learned dynamic model is usually not satisfied, especially in real-world systems where the dynamic models are often complex and the environment is not fully observable. This kind of model error makes model-based RL less preferable to model-free RL in a wide range of tasks and is a crucial subject to address when employing model-based RL since even small biases in the learned model can lead to strongly-biased sub-optimal policies.

Many solutions have been proposed to compensate for the weaknesses of model-free and model-based RL. One of the preferred techniques is to integrate the learned dynamic model as a component of

Preprint. Work in progress.

model-free algorithms. For example, the Dyna-Q family [21, 11] learns the dynamic model to generate imagination rollouts for Q-learning in addition to the ones obtained from interacting with the real environment. A recent work [18] uses the policy learned by a model-based RL algorithm as initial policy for a model-free learner. [1] use the learned dynamic model to compute the trajectory distribution corresponding to a given policy and consequently the estimated cost of the policy. This approximated cost is then used to guide policy exploration for the model-free learner using Bayesian optimization. [6] improve the sample-complexity of model-free learning by restricting the use of the learned dynamic model to a fixed depth while allowing wider usage of the dynamic model. Overall, model-based/model-free hybrid methods have been shown to achieve on-par performance with model-free algorithms while having lower sample-complexity.

Most of the model-based components of the aforementioned methods utilize fixed number of steps rollout: they use the learned dynamic model to simulate transitions exactly H steps into the future. The imagined transitions can help to improve value estimation because the agent can "look ahead" and see what future states might follow from whatever action it is taking. This prevents the agent from exploiting bad actions that have high short-term gain.

However, such fixed rollout strategies have several drawbacks. First, rolling out a fixed number of steps can lead to a situation where simulated transitions are no longer accurate enough due to errors in the dynamic model. This usually causes catastrophic drop in performance as the agent will be provided with states that are too different from the real environment or are simply unreachable. Second, fixed rollout does not take into account the fact that imagined transitions are usually more useful at the beginning of training when the value estimation functions have not started to converge yet and less useful at the end of training where real data provides much better training targets for the agent. An ideal rollout strategy would rollout just enough number of steps, usually more in the beginning and less or even none at the end of training. Finally, for many algorithms, the number of fixed rollout steps is a hyper-parameter that needs to be chosen during training. Automatic selection of rollout steps can reduce training time significantly.

In this paper, we propose an adaptive rollout algorithm for model-based RL based on the error estimate of the dynamic model and the state value function. In our algorithm, we only rollout until the estimated compound error of the dynamic model is higher than the estimated error of the next state value. This adaptive rollout strategy is similar to human planning: we can only predict a certain number of steps into the future until our forecasting model of the world becomes too unreliable and we have to rely on a rough value estimate of the last predicted state instead. This adaptive rollout algorithm should address the problems with fixed rollout. Adaptive rollout alleviates the effects of compound model errors as we explicitly use a heuristic to control when to stop rolling out based on compound model errors. Our rollout algorithm should also have fewer rollout steps into the future at the end of training, as the state value error estimate should be small during this period. Finally, adaptive rollout chooses the number of rollout steps automatically, thus removing the need for additional hyper-parameter tuning.

2 Background

2.1 Markov Decision Process

Following the notations in [6], a **deterministic** Markov Decision Process (MDP) is specified by the 5-tuple $(\mathcal{S}, \mathcal{A}, f, r, \gamma)$. Here, $\mathcal{S}$ and $\mathcal{A}$ are the sets of possible states and actions, $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the deterministic state transition function, $r : \mathcal{S} \to \mathbb{R}$ is the bounded reward function, and γ is the discount factor used to attach less importance to more distant rewards. The main objective in solving an MDP problem is to determine a deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ that can choose an action in each state so as to maximize the long-term γ-discounted cumulative reward. This work assumes continuous state and action spaces so that π is a function of real values parameterized by θ. This parameterization is assumed to be implicit in the definition of π for simplicity though might be explicitly indicated as π_θ when needed.

The cumulative reward can be represented by the action value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ defined as $Q^\pi(s_i, a_i) = \sum_{t=i}^{\infty} \gamma^t r_t$ where $r_t = r(s_t, a_t)$ and actions are chosen according to π so that $a_t = \pi(s_t)$ and $s_{t+1} = f^\pi(s_t) = f(s_t, \pi(s_t))\ \forall\ t \geq 0$. An alternative way to express the cumulative reward is through the value function $V^\pi : \mathcal{S} \to \mathbb{R}$ defined as $V^\pi(s) = Q^\pi(s, \pi(s))$. Since the true value functions Q^π and V^π are unknown, these are approximated using estimates denoted as $\hat{Q}$ and $\hat{V}$ respectively. The objective function to be maximized for finding the optimal policy can then be expressed as $J_{d_0}(\pi) = \mathbb{E}_{d_0}[V^\pi(S)]$ where $S \in \mathcal{S}$ is sampled from the initial state distribution d_0. Further, this work considers the off-policy setting where an exploratory policy is derived from π by random perturbation. Assuming the corresponding states to have been drawn from a distribution ν, the approximate objective is refereed to as J_ν. Finally, with a slight abuse of notation, f and r will be used to refer to learned models for the unknown state transition and reward functions respectively.

2.2 Neural networks with dropout as Bayesian Approximation

The dropouts technique was introduced by [20] to avoid over fitting while training deep networks. This involves selecting a random subset of units in some or all layers during each forward pass and turning them off, i.e. setting their outputs to zero. A fixed proportion of units is turned off for each layer and a different set of units is selected for each forward pass, though the same one is used for the corresponding back propagation too.

Let $\mathbf{W}_i$ and $\mathbf{b}_i$ denote the weight matrix of size $K_i \times K_{i-1}$ and the bias vector of size K_i for the i^{th} layer of the network with L layers. Dropout in this layer is implemented by sampling a binary vector $\mathbf{z}_i$ from a Bernoulli distribution such that $p_i \in [0, 1]$ is the probability of each element of this vector being 1 and the proportion of units turned off is thus $1 - p_i$. The output of this layer is then given as $\hat{\mathbf{y}}_i = \sigma((\hat{\mathbf{y}}_{i-1} \circ \mathbf{z}_i)\mathbf{W}_i + \mathbf{b}_i) = \sigma(\hat{\mathbf{y}}_{i-1}(\text{diag}(\mathbf{z}_i)\mathbf{W}_i) + b_i)$ where σ is the non-linearity (e.g. ReLU), $\circ$ denotes the element wise product while $\text{diag}(\cdot)$ is the diagonal matrix with its input vector in the diagonal.

This technique was shown by [9] to be equivalent to performing approximate variational inference on a deep Gaussian process (GP) [3] after marginalizing over the parameters of its covariance function. This equivalence is shown by first approximating the deep GP posterior $p(\mathbf{y}|\mathbf{x})$ using a variational distribution $q(\omega)$. The estimation of $q(\omega)$ through minimization of KL divergence is then further approximated using single sample Monte Carlo integration to show that the final objective to be optimized is identical to an L_2 regularized loss function corresponding to Eq. **??**. The detailed proof is too mathematically involved to be included here even in condensed form. The reader is referred instead to [9] and its separate appendix [8] for more details.

In addition to proving this equivalence, [9] also developed insights about how NNs with dropouts can be used to estimate uncertainty of the network output. As shown in [8][sec. 2.3], the predictive distribution is given as $q(\mathbf{y}^*|\mathbf{x}^*) = \int p(\mathbf{y}^*|\mathbf{x}^*, \omega)q(\omega)d\omega$. Uncertainty estimation is done by empirically computing the first two moments of this distribution. In theory, this is done by sampling a set of T binary vectors $\{\mathbf{z}_1^t, ..., \mathbf{z}_L^t\}$ from the Bernoulli distribution to get the corresponding weight matrices $\{\mathbf{W}_1^t, ..., \mathbf{W}_L^t\}$. These weights are then used to obtain T samples for $\hat{\mathbf{y}^*}$ and the mean and variance are computed over this sample set.

For the NN, this involves performing T stochastic forward passes through the network, each one with a different set of units turned off, and then computing the sample mean and variance of the results thus obtained. This is called **Monte Carlo Dropout**. Also, though, [9] mentioned that the inverse of the model precision needs to be added to the sample variance, it turns out that this is not needed for reinforcement learning applications, as shown, for instance, by [10].

3 Adaptive Rollout using Uncertainty Estimate

3.1 Adaptive Rollout

Consider a model-based RL algorithm that learns a state value function V, a dynamic model f and a reward function r. Denote $\tau = (\tau_0, \tau_1, ..., \tau_t)$ as the rollout trajectory using f and r starting from τ_0 with $\tau_i = (s_{i-1}, a_{i-1}, r_{i-1}, s_i)$ for $i = 0, ..., t$ when the agent follows a policy π which can be a greedy policy or a policy that is parameterized by θ. $\tau_0 = (s_{-1}, a_{-1}, r_{-1}, s_0)$ where s_0 denotes the starting rollout state and s_{-1}, a_{-1}, r_{-1} denote previous state, action and reward.

Assume that f has the capability to output the compound error estimate between the rollout trajectory and the true policy, i.e. the trajectory obtained when we follow the true dynamic model instead of f. This compound error estimate can be defined as the squared error between the final state of the rollout trajectory and the true dynamic model trajectory. One of the most common ways to model this

Algorithm 1 General Adaptive Rollout

Input: Starting transition $\tau_0 = (s_{-1}, a_{-1}, r_{-1}, s_0)$. Policy π. State value function V with error estimate. Dynamic model f with error estimate. Reward model r.
Output: Transitions $\tau_1, \tau_2, ..., \tau_t$

```
 1: $t \leftarrow 0$
 2: while true do
 3:     Generate an action $a_t$ according to the policy $\pi$ and current state $s_t$
 4:     Use $f$ and $r$ to generate the next transition $\tau_t = (s_t, a_t, r_t, s_{t+1})$. Also, record the compound
            error estimate $e_{t,f}$ of the next state.
 5:     Compute the value function for the next state $V(s_{t+1})$. Record the error estimate $e_{t,V}$.
 6:     if $any(e_{t,f} > e_{t,V})$ then
 7:         break
 8:     end if
 9:     $t \leftarrow t + 1$
10: end while
11: return $\tau_1, \tau_2, ..., \tau_t$
```

compound error estimate is to treat it as uncertainty in f and use probabilistic approaches like GP to model this uncertainty. There have been many works that follow this paradigm, for example, PILCO [5] and DeepPILCO [10]. The theoretical relationship between the variance of a GP and its mean prediction error is well-known [12]. Additionally, we assume that V also has the ability to output the estimated error for a state value. In this work, we only consider the case where the environment is **deterministic** and the error estimate output by out models account only for their belief about the accuracy of the predictions.

Starting from the transition τ_0, the Adaptive Rollout algorithm proceeds as follows: at every step t with current state s_t, the agent takes an action a_t according to the current policy π, gets a reward r_t from the reward model r and a next state s_{t+1} with its compound error estimate $e_{t,f}$ from the dynamic model. After that, the agent computes the next state value $V(s_{t+1})$ and its error estimate $e_{t,V}$. The agent then compares the two error estimates and if $e_{t,f} > e_{t,V}$ at any step t, the agent stops the rollout. This process continues as a loop until it is terminated using the aforementioned condition.

Note that, because the error estimate $e_{t,f}$ of f is a vector that has the same dimensionality as the states and $e_{t,V}$ is a scalar, we cannot compare them directly. Instead, we compare each value in $e_{t,f}$ with $e_{t,V}$ and if any of them is greater than $e_{t,V}$, we stop the rollout process. We denote this kind of comparison as $any(e_{t,f} > e_{t,V})$. Another issue is that, for the comparison between each value in $e_{t,f}$ with $e_{t,V}$ to make sense, $e_{t,V}$ and these values must have the same scale. In the general Adaptive Rollout algorithm, we assume that both f and V output error estimates such that this scale condition is satisfied. The general Adaptive Rollout algorithm is shown as Algorithm 1.

3.2 Adaptive Rollout using Neural Networks with Dropout

In this section, we discuss a special case of the general adaptive rollout algorithm where V and f are represented by neural networks with dropout (Algorithm 2). Traditionally, model errors have been treated as variance of GPs. However, GPs do not scale well to high dimensional problems as they rely on Euclidean distance to define their input-space correlation which becomes uninformative in higher dimensional problems [2]. Furthermore, GPs are slow too as their runtime scales cubically with the amount of input data. Neural networks with dropout have been shown to approximate Deep GPs [4] which are hierarchical extensions of GPs. They are also faster than GPs, as their time complexity only scales linearly with the number of input samples and they can be computed very quickly on GPUs. For a summary of neural networks with dropout, see section 2.2. In order to model f and V thus, we need to address three issues: (1) How to compute the error estimate of V, (2) How to compute the compound error estimate of the rollout trajectory, and (3) How to scale the error estimates for f and V to the same scale.

Issue (1) is trivial as we can just do a couple of stochastic forward passes through our V once we have the next state s_{t+1} and use the variance of the output as our error estimate. To resolve issue (2), we noticed that, after the first imagination rollout step, there is uncertainty in our states. Namely, our state estimation is a probability distribution rather than a single point in the state space. To compute

Algorithm 2 Adaptive Rollout using Neural Network with dropout

Input: Starting transition $\tau_0 = (s_{-1}, a_{-1}, r_{-1}, s_0)$ using particles method. K: number of particles/sampling time for reward and state value functions. Policy π and state value function V which are parameterized by θ and φ ($\theta = \emptyset$ in case π is a greedy policy) . f and r: state dynamic and reward models.

Output: Transitions $\tau_1, \tau_2, ..., \tau_t$

```
 1: Initialize a set of $K$ particles $s_t^k$, $1 \leq k \leq K$ to $s_0$
 2: for $k = 1$ to $K$ do
 3:     Sample dropout weight for $k^{th}$ particle $W^k$
 4: end for
 5: $t \leftarrow 0$
 6: while true do
 7:     $a_t \leftarrow \pi_\theta(s_t)$
 8:     for each particle $s_t^1$ to $s_t^K$ do
 9:         Evaluate $f(s_t^k, a_t)$ with dropout weight $W^k$ and input particle $s_t^k$, obtain output $f_t^k$
10:     end for
11:     Compute the mean $\mu_{t,f}$ and standard deviation $\sigma_{t,f}^2$ of $\{f_t^1, f_t^2, ..., f_t^K\}$
12:     $s_{t+1} \leftarrow \mu_{t,f}$
13:     Evaluate the next state value function $V_\varphi(s_{t+1})$ $K$ times.
14:     Compute the mean $\mu_{t+1,V}$ and standard deviation $\sigma_{t+1,V}^2$ of the sampled state values.
15:     if $any(normalize\_state(\sigma_{t,f}^2) > normalize\_state\_value(\sigma_{t+1,V}^2))$ then
16:         break
17:     end if
18:     Evaluate the reward model $r(s_t, a_t, s_{t+1})$ to get $r_t$.
19:     $\tau_{t+1} = (s_t, a_t, r_t, s_{t+1})$
20:     Sample a set of $K$ new particles $s_{t+1}^k \sim \mathcal{N}(\mu_{t,f}, \sigma_{t,f}^2)$
21:     $t \leftarrow t + 1$
22: end while
23: return $\tau_1, \tau_2, ..., \tau_t$
```

the compound error estimate, we need to propagate our state distribution through the dynamic model. Similar to [10], we use particle method to feed a distribution through the dynamic model. This involves sampling a set of particles from the input distribution and feeding the particles into the dynamic model, which yields an output distribution represented by the output particles. At the start of the rollout, the input distribution is the starting state s_0. After each rollout step, the output distribution becomes the input distribution for the next rollout step. Additionally, we re-sample a new set of particles after each rollout step and fix the dropout mask W^k for all K particles during the rollout process, similar to [10]. The next state s_{t+1} of the rollout is the mean value of the output distribution while the compound error estimate up to the current rollout step is its variance.

To solve the scale problem in issue (3), we propose a simple heuristic to normalize the error estimates of f and V by dividing each by its own estimated variance. This is equivalent to scaling the output of f and V to unit scale. Specially, we divide the error estimate of f by the variance of all the states in the replay memory R. For V, we first compute the expectation values of the states in the replay memory (i.e. $\mathbb{E}[V(s)]$) and then use the variance of these values as the estimated variance for V. Because this operation is expensive, we only compute this variance for a number of states that are randomly sampled from R and only done this computation once every few action steps. Note that this normalization scheme is only applied to the comparison between error estimates of f and V only nowhere else. Also note that this adaptive rollout algorithm can be easily attached to many existing model-based RL algorithms as long as they use neural networks as their dynamic model and state value function, which is quite common in the literature.

4 Related works

There have been several works that employ uncertainty estimate to improve model-based RL. Many of these algorithms rely on using a probabilistic model of the agent's ignorance of the world to allow it to choose actions under uncertainty. The PILCO algorithm [5] learns a model from scratch using

GP and uses it for long-term planning. During policy evaluation, uncertainty is propagated through time-steps to update the parameters of a linear controller using analytical gradients. PILCO achieved an unprecedented sample efficiency even though it learned everything from scratch. The work by [10] improves on PILCO by using neural networks with dropout as their probabilistic model instead of GP, resulting in lower planning cost than PILCO. In [16], the authors use Model Predictive Control in conjunction with reformulating the optimal control problem with learned GP models as an equivalent deterministic problem to find optimal control signals, while handling constraints in a principled way.

Another way to use the uncertainty estimate is to limit the usage of the dynamic model based on the uncertainty estimate of the rollouts. [13] combat the adverse effects of bad imagination rollout by restricting the use of imagination data to only when the variance of the rollout batch is high. Their work is arguably closest to ours. However, instead of restricting the usage of the whole rollout batch, we use the uncertainty estimate to stop rollout early before the dynamic model become too inaccurate.

[7] introduced an alternative approach to reducing the rollout length while maintaining the planning performance. They added an independent value function approximator or critic to the direct policy iteration (DPI) algorithm [15] and used a portion of the total budget of environmental interactions or model evaluations to train it. The return corresponding to the steps truncated due to the finite rollout horizon was then approximated by this learned value function instead of implicitly setting it to zero. Since the variance of the estimated return increases with the rollout length while its bias decreases, this approach helps to reduce the variance while controlling the bias through incorporation of the independent critic.

Another significant issue with longer rollouts is that the model encounters inputs that may never be produced by the environment and thus be very different from its training samples. This can cause the model to generate rollout trajectories that make no sense for the actual environment and lead to catastrophic planning failure. [22] proposed the hallucinated replays method for solving this problem by utilizing some of the model outputs themselves as training samples for the model. This was done by using the context information (state-action history) from some step in the trajectory as input to the model to produce the predicted observation. This replaces the true observation for that step to generate the hallucinated context for the next step which, along with the true observation for that step, is used to form a training sample for the model.

Theoretical analysis of this method was presented in [24] along with a novel error type called hallucinated one step error formulated as an alternative to the standard one step prediction error. This error was shown to be more closely related to the actual model performance within the setting of a deterministic environment and blind rollout policy. This result was used to extend the DAgger-MC method of [23] with unrolled models and hallucinated replays. The resultant algorithm, called H-DAgger-MC, was also proven to be theoretically capable of learning the perfect model if one existed in the model class even with hallucinated samples in the training set. H-DAgger-MC was further extended by [25] to apply hallucinated replays while learning the reward function too so that the latter can be specialized to flawed model dynamics.

5 Experiments

5.1 A simple V learning algorithm

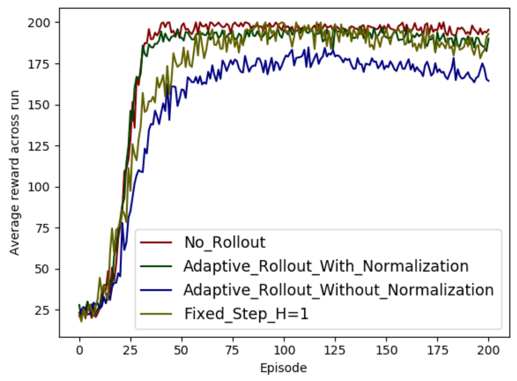
We test the effects of using adaptive rollout instead of fixed step rollout in a simple Dyna-Q-like algorithm called V learning (Algorithm 3). Here, we learn the state value function V, the dynamic model f and the reward function r instead of the state-action value function Q. Additionally, we simulate long trajectories from some starting transitions sampled randomly from the replay memory R. As we need to model the uncertainty estimate, V and f are modeled using neural networks with dropout that approximate GPs. Although we do not learn Q, we can reconstruct it via V, f and r using $Q(s,a) = r(s,a,s') + \gamma \mathbb{E}[V(s')]$ where $s' = \mathbb{E}[f(s,a)]$ is the expected next state computed by averaging the outputs obtained by sampling f using Monte Carlo dropout a number of times. Similarly, we can compute the expected next state value $\mathbb{E}[V(s')]$ by averaging the outputs obtained by sampling V. The policy used in this algorithm is the ϵ-greedy policy with ϵ decaying over time.

Similar to [17], we use a replay buffer R to store all real transitions as training data. During an episode, the algorithm executes an action a_t using the ϵ-greedy policy on the current state s_t, observes a reward r_t and the next state s_{t+1} and stores the transition (s_t, a_t, r_t, s_{t+1}) into R. Next, a mini-batch of N real transitions is sampled from the replay buffer R and used for training f and r with L_2 losses. To train V, we set $y_i = r_i + \gamma\mathbb{E}[V(s_{i+1})]$ for every transition in the sampled mini-batch and do an update on parameters of V by minimizing the loss $L = \frac{1}{N}\sum_i (y_i - V(s_i))^2$. Note that this update is similar to and inspired by the TD update but has been modified to be suitable for training neural networks with dropout.

After training f, r and V on real transitions, we use f and r to generate simulated transitions that are used as additional training data to train V. These start from a real transition and can be generated using either fixed number of steps rollout or adaptive rollout. For fair comparison between all rollout strategies, we randomly select only a fixed number $N_{rollout}$ of simulated transitions for training. Note that this algorithm only works for **discrete action space problems**.

Algorithm 3 V learning with dynamic model rollouts

```
 1: Randomly initialize parameters of the state value function $V$, the dynamic model $f$, the reward
    function $r$
 2: Initialize replay buffer $R \leftarrow \emptyset$
 3: while not solved do
 4:     Receive an initial state $s_1$
 5:     for $t = 1, T$ do
 6:         Select action $a_t$ according to the $\epsilon$-greedy policy.
 7:         Execute $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
 8:         Store $(s_t, a_t, r_t, s_{t+1})$ into replay buffer $R$
 9:         Sample a random mini-batch of $N$ transitions from $R$
10:         Train $f$ and $r$ with data in the mini-batch using $L_2$ losses.
11:         Set $y_i = r_i + \gamma\mathbb{E}[V(s_{i+1})]$ for each transition in the mini-batch
12:         Update $V$ by minimizing the loss $L = \frac{1}{N}\sum_i (y_i - V(s_i))^2$
13:         Set simulated rollouts buffer $R_f \leftarrow \emptyset$
14:         for each transition $(s_i, a_i, r_i, s_{i+1})$ in the mini-batch of N real transitions do
15:             Use $f$ and $r$ to generate simulated transitions from $(s_i, a_i, r_i, s_{i+1})$. Can use
                    fixed steps rollout or adaptive rollout.
16:             Add all simulated transition to $R_f$
17:         end for
18:         Randomly select $N_{rollout}$ transitions from $R_f$
19:         Update $V$ using selected simulated transitions from previous step in a similar manner to
                updating $V$ using real transitions.
20:     end for
21: end while
```



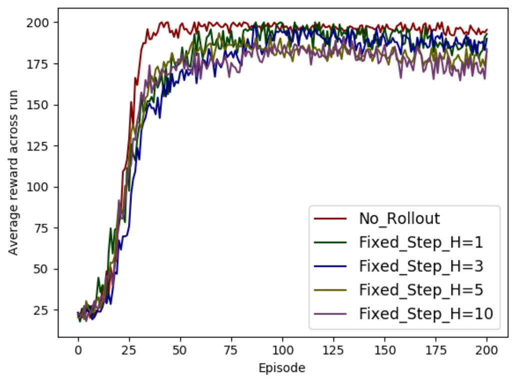


Figure 1: **Average rewards across 20 runs for different rollout strategies.** Left: No rollout vs Adaptive rollout vs Fixed 1 step rollout. Right: No rollout vs Fixed 1, 3, 5 and 10 steps rollout.

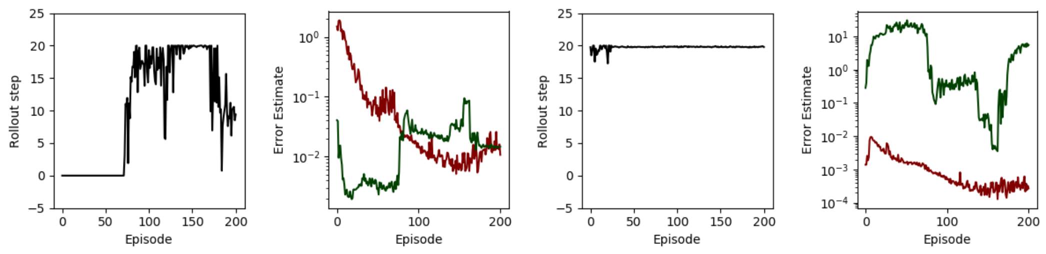


Figure 2: **Number of rollout steps and estimated error for both adaptive rollout algorithms**. The two figures on the left are for the algorithm with normalization and the two on the right are for the one without normalization. Black, red and green lines respectively represent number of rollout steps, dynamic model error and state value error. Number of rollout steps is capped at 20.

5.2 Results

We have tested adaptive rollout against no rollout and fixed steps rollout with number of steps $H \in \{1, 2, 3, 5, 10\}$ on the CartPole problem. No rollout means that no simulated transitions are used for additionally training V. Note that the no rollout algorithm actually does a one-step lookup as our Q function decomposition requires looking at the expected next states $\mathbb{E}[V(s')]$. We have used 3-layer neural networks for f, r and V, with hidden layer dimensionality of 8 for f and r and 256 for V. f and V both use a dropout rate of 0.05. All neural networks use $tanh$ as their hidden activation and no output activation. Adam optimizer [14] is used for training with a learning rate of 0.001.

Figure 1, left, shows the average episode rewards for 200 episodes averaged over 20 runs. Overall, adaptive rollout algorithm does quite well with normalization but fails without it. In fact, it achieves the second best performance, better than 1-step Rollout and just behind the optimal strategy of no rollout for the CartPole problem. It improves as fast as no rollout at the beginning but its performance decreases a bit toward the end. Meanwhile, adaptive rollout without normalization improves slowly and has the worst performance of all the rollout strategies.

Figure 1, right, compares the performance of fixed steps rollout with no rollout. The latter can be seen to be the best strategy and performance decreases with increasing steps. This is because the CartPole task does not require looking far ahead into the future to find a good policy; one can be obtained by just looking at the next state and choosing the action that makes the pole as vertical as possible.

To inspect the behavior of adaptive rollout in more detail, we plot the number of rollout steps and error estimates of f and V in Figure 2. Note that we limit the maximum number of rollout steps to 20. Because the difference in scale between the outputs of f and V, adaptive rollout without normalization fails and always returns the maximum number of rollout steps. Adaptive rollout with normalization does well in the first half of the episodes. However, it starts to rollout after a while. Upon comparing the error estimates of f and V, we see that the latter rises abruptly in the middle of the graph. This corresponds to significant changes in the agent's policy. Our hypothesis is that, when the agent changes its policy, it sees new states that it has not been trained on and our models have the tendency to output constant value for such states [9]. This phenomenon messes up the variance estimates of f and V, causing this strange behavior. This problem will be addressed in a future work.

6 Discussion

This paper proposed a simple adaptive rollout algorithm based on uncertainty estimate to address several issues with fixed number of steps rollout. This approach can provide improved performance on both discrete and continuous action space tasks due to the increased quality of the rollout transitions. Existing approaches that use uncertainty estimate to improve model-based RL usually focus on reducing sample-complexity [5, 10] or use uncertainty estimate to limit model usage [13]. Our algorithm provides an alternative way to use uncertainty estimate in model-based RL and can be easily plugged into many existing model-based approaches where a dynamic model is available. Though experiments on more complicated tasks are needed to verify its usefulness, we believe that automatic rollout steps selection provides a promising direction for future exploration.

References

[1] S. Bansal, R. Calandra, S. Levine, and C. Tomlin. MBMF: Model-Based Priors for Model-Free Reinforcement Learning. *arXiv preprint arXiv:1709.03153*, 2017.

[2] Y. Bengio, O. Delalleau, and N. L. Roux. The curse of highly variable functions for local kernel machines. In *Advances in neural information processing systems*, pages 107–114, 2006.

[3] A. Damianou and N. Lawrence. Deep Gaussian processes. In C. Carvalho and P. Ravikumar, editors, *Proceedings of the Sixteenth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, AISTATS '13, pages 207–215. JMLR W&CP 31, 2013.

[4] A. Damianou and N. Lawrence. Deep gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215, 2013.

[5] M. Deisenroth and C. 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.

[6] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine. Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning. *ArXiv e-prints*, Feb. 2018.

[7] V. Gabillon, A. Lazaric, M. Ghavamzadeh, and B. Scherrer. Classification-based Policy Iteration with a Critic. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 1049–1056, USA, 2011. Omnipress.

[8] Y. Gal and Z. Ghahramani. Dropout as a Bayesian Approximation: Appendix. *ArXiv e-prints*, May 2016. arXiv:1506.02157v5.

[9] Y. Gal and Z. Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[10] Y. Gal, R. McAllister, and C. E. Rasmussen. Improving PILCO with Bayesian Neural Network Dynamics Models. In *Data-Efficient Machine Learning workshop, ICML*, Apr. 2016.

[11] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[12] A. F. Hernandez and M. A. Grover. Error estimation properties of gaussian process models in stochastic simulations. *European Journal of Operational Research*, 228(1):131–140, 2013.

[13] G. Kalweit and J. Boedecker. Uncertainty-driven imagination for continuous deep reinforcement learning. In *Conference on Robot Learning*, pages 195–206, 2017.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] A. Lazaric, M. Ghavamzadeh, and R. Munos. Analysis of a Classification-based Policy Iteration Algorithm. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 607–614, USA, 2010. Omnipress.

[16] F. Meier, D. Kappler, and S. Schaal. Online learning of a memory for learning rates. *arXiv preprint arXiv:1709.06709*, 2017.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[18] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *arXiv preprint arXiv:1708.02596*, 2017.

[19] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[21] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, pages 216–224. Elsevier, 1990.

[22] E. Talvitie. Model Regularization for Stable Sample Rollouts. In *UAI*, 2014.

[23] E. Talvitie. Agnostic System Identification for Monte Carlo Planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 2986–2992, 2015.

[24] E. Talvitie. Self-Correcting Models for Model-Based Reinforcement Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 2597–2603, 2017.

[25] E. Talvitie. Learning the Reward Function for a Misspecified Model. *CoRR*, abs/1801.09624, 2018.