Model-based Reinforcement Learning with Non-linear Expectation Models and Stochastic Environments

Yi Wan [*1] **Muhammad Zaheer** [*1] **Martha White** [1] **Richard S. Sutton** [1]

Abstract

In model-based reinforcement learning (MBRL), the model of a stochastic environment provides, for each state and action, either 1) the complete distribution of possible next states, 2) a sample next state, or 3) the expectation of the next state's feature vector. The third case, that of an *expectation model*, is particularly appealing because the expectation is compact and deterministic; this is the case most commonly used, but often in a way that is not sound for non-linear models such as those obtained with deep learning. In this paper, we introduce the first MBRL algorithm that is sound for non-linear expectation models and stochastic environments. Key to our algorithm, based on the Dyna architecture, is that the model is never iterated to produce a trajectory, but only used to generate single expected transitions to which a Bellman backup with a *linear* approximate value function is applied. In our results, we also consider the extension of the Dyna architecture to partial observability. We show the effectiveness of our algorithm by comparing it with model-free methods on partially-observable navigation tasks.

1. Introduction

Model-free reinforcement learning methods have achieved impressive performance in a range of complex tasks (Mnih et al., 2015). These methods, however, require millions of interactions with the environment to attain a reasonable level of performance. With computation getting exponentially faster, interactions with the environment pose themselves to be the primary bottleneck in the successful deployment of reinforcement learning algorithms to real-world applications. In contrast, model-based reinforcement learning methods promise sample-efficient learning by modeling how the world works (Betts, 1998)(Deisenroth and Rasmussen, 2011)(Browne et al., 2012)(Oh et al., 2017).

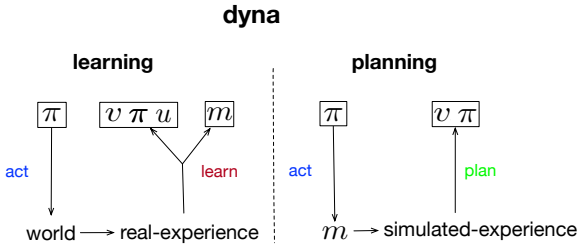


Figure 1. (a) Actor-critic with dyna-style planning uses the real-experience to not only learn the state update, value and policy functions but also the model. Concurrently, it uses the model to simulate experience and update the value and policy functions

In MBRL, it is a common practice to learn non-linear expectation models of the environment (Oh et al., 2015)(Finn et al., 2016)(Leibfried et al., 2016). When the world is stochastic, iterating a non-linear expectation model leads to an invalid trajectory. In order to deal with this limitation, we argue that the model should be used to only simulate one-step transitions and the planning process should utilize the simulated one-step transitions to perform planning updates based on Bellman backups (Bertsekas and Tsitsiklis, 1995). We also note that to obtain correct targets for Bellman backups, the value function has to be linear. To retain the expressiveness of the value function and tackle partial observability, we propose the use of *state update function*, an arbitrarily complex non-linear function of the history. In subsequent sections, we expand on each of the aforementioned issues and their solutions in detail.

The main contributions of this paper are as follows: 1) we devise the first MBRL algorithm that is sound for non-linear expectation models and stochastic environments. Instead of iterating the model or using a non-linear value function, we simulate one-step transitions and use a linear value function; 2) In order to learn a complex value function and deal with partial observability, we learn to generate a flexible agent state using the state update function, which could be any arbitrarily complex non-linear function of the observations.

[*]Equal contribution [1]Department of Computing Science, University of Alberta, Edmonton, Canada. Correspondence to: Yi Wan <wan6@ualberta.ca>, Muhammad Zaheer <mzaheer@ualberta.ca>.

Accepted at the FAIM workshop "Prediction and Generative Modeling in Reinforcement Learning", Stockholm, Sweden, 2018.
Copyright 2018 by the author(s).

The proposed algorithm learns the model of the transition dynamics in the agent state space, intermixes model-free and model-based methods and performs Bellman backups using the simulated experience in the agent state space; 3) We fuse the above two strategies with model-free Asynchronous Advantage Actor-Critic (A3C) method with Generalized Advantage Estimation (Mnih et al., 2016)(Schulman et al., 2015). We evaluate the proposed method in fully and partially observable navigation tasks. Our experiments show that the proposed method considerably improves the model-free baselines methods in terms of the number of environmental interactions.

2. Stochastic Environments and Non-Linear Models

In order to model a stochastic world, one can in principle learn a distributional model, which provides a distribution over the next state, a sample model, which produces a sample of the next state, or an expectation model, which provides the expected next agent state's feature vector (in the following, we refer to the agent state's feature vector simply as agent state). When the state space is high-dimensional, both the distribution model and the sample model are prohibitively hard to learn. In contrast, the expectation model is relatively easier to learn and it scales well with the size of the state space.

If a stochastic world is modeled using an expectation model, the expected next state might not correspond to any real-state. If such expected next-state is fed into the model again as input, it would be something that the model has never encountered during the training, possibly leading to an arbitrary output. Thus, expectation models can not be used to simulate valid trajectories by iterating themselves. A potential solution to side-step this inherent limitation of the expectation model is to generate only single-step transitions and use them to perform Bellman backups. In this work, we focus on simulating multiple single-step transitions and learning from them as if they were real transitions. An added benefit of using single-step transitions for planning is that they are relatively robust to model errors. When an imperfect model is used to simulate a trajectory, model errors compound over the length of the rollout leading to unlikely trajectories with hypothetical states (Talvitie, 2017).

In order to perform Bellman backups with single-step transitions, the target consists of the expected value of the next state. We note that if the value is a non-linear function of the state, the aforementioned target cannot be obtained from the expected next state, which is what is given by the expectation model (Paduraru, 2007). Concretely, the value of the expected next state is not equal to the expected value of next state when the parametrization $v_{\mathbf{w}}$ is not a linear function $v_{\mathbf{w}}(\mathbb{E}[S_{t+1}|S_t = s, A_t = a]) \neq \mathbb{E}[v_{\mathbf{w}}(S_{t+1}|S_t = s, A_t = a)])$. In order to use the expectation model to obtain the correct Bellman backup target, we propose to use a linear value function.

The linearity seems to be a severe limitation on the expressiveness of the value function. To ensure that we are still able to express complex value functions, we introduce the notion of *state update function* which could be any arbitrarily complex non-linear function of history that is optimized while solving the task.

3. State Update Function and Partial Observability

While the state-update function allows us to use a linear value function, it also presents itself as a potential solution to partial observability. A common problem for RL agents operating in complex environments is that they do not have access to the states of the underlying MDP. Instead, an agent receives observations which convey partial information of the states. Given these observations, it is desirable to construct a compact summary of the agent's experience which could be used to make decisions effectively. We refer to this compact summary as *agent state* and treat this agent state the same way we treat the state in traditional RL methods. Once the notion of the agent state has been defined, the traditional RL definitions and methods can be used. For instance, the agent state can be used as the input to the policy and value functions.

At a given time-step t, the agent state can be estimated using the state update function u which recursively updates the agent state to obtain S_{t+1} using the previous agent state S_t, the action A_t, the next observation O_{t+1} and the reward R_{t+1} i.e. $S_{t+1} = u(S_t, A_t, O_{t+1}, R_{t+1})$. The state update function can be either given by the designer or could be learned by the agent.

4. A Concrete Algorithm Based on Dyna Architecture

In this section, we describe a concrete algorithm for our proposed approach. We extend the Dyna (Sutton, 1991) (Sutton et al., 2012) (Pan et al., 2018) architecture to non-linear models and partial observability and integrate it with an actor-critic method. We begin by describing the core components of our formulation. Subsequently, we discuss the algorithmic details which highlight how we interleave learning from the real-experience and planning with the simulated experience.

4.1. Architectural Components

In this section, we outline the architectural components: state update function u, expectation model m, policy func-

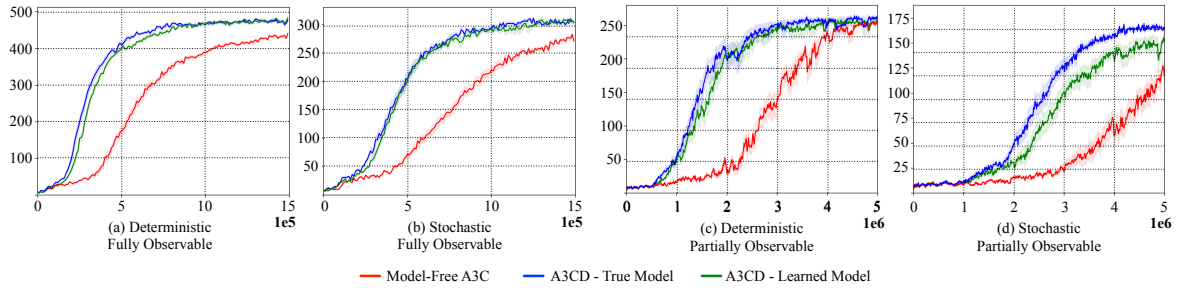


Figure 2. Learning curves on the deterministic and stochastic variants of the fully observable and the partially observable navigation tasks. x-axis represents the number of interactions with the environment, y-axis corresponds to the number of episodes completed in the last 10,000 environment interactions. Learning curves are averaged from 20 runs with different random seeds.

tion π and value function v_π parameterized by $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}$ and $\mathbf{w}$ respectively.

Non-linear State update function
$u_{\boldsymbol{\alpha}}$: $S_{t-1}, A_{t-1}, O_t, R_t \rightarrow S_t$, which estimates the current agent state s_t given the last agent state S_{t-1}, last action A_{t-1}, the current observation O_t and the current reward R_t. The state update function can be modeled using an LSTM (Hochreiter and Schmidhuber, 1997).

Non-linear Expectation model
$m_{\boldsymbol{\beta}}$: $S, A \rightarrow \hat{S}', \hat{R}$, which takes agent state S and action A as input and outputs the next expected agent state and the next expected reward. The non-linear expectation can be modeled using a feedforward neural network.

Non-linear policy function
$\pi_{\boldsymbol{\theta}}(A|S)$, which takes agent state S and action A as input and outputs the probability of taking action A in agent state S. The policy can be modeled using a feedforward neural network followed by a softmax function.

Linear value function
$v_{\mathbf{w}}$: $S \rightarrow \hat{v}_{\pi_\theta}$ which takes agent state S as input and outputs the value $v_{\mathbf{w}}(S)$. We approximate $v_{\mathbf{w}}$ with a linear function, i.e., $v_{\mathbf{w}}(S) = \mathbf{w} \cdot S$.

4.2. Algorithmic Description

Dyna architecture involves three distinct phases: direct learning, model learning and planning. The three phases can proceed serially or concurrently. Figure 1 illustrates the actor-critic architecture with dyna-style planning and contrasts it with the model-free actor-critic architecture. In this section, we further describe each of the three phases in the context of actor-critic algorithm.

Direct Learning: In the direct learning phase, we learn the state update function, the value function and the policy with real experience using the actor-critic method. Concretely, we first learn the value function $v_{\mathbf{w}}$ and the state update function $u_{\boldsymbol{\alpha}}$ by updating parameters with the semi-gradient of the n-step TD-error. We learn the policy $\pi_{\boldsymbol{\theta}}$ and the state update function $u_{\boldsymbol{\alpha}}$ by taking the gradient ascent on the policy gradient objective.

Model Learning: During direct learning, we collect the transitions $(S_t, A_t, R_{t+1}, S_{t+1})$ and store them in a recency buffer. During the model learning phase, we sample a mini-batch of random transition (S_i, A_i, R_i, S_i') and learn the model $\hat{S}_i', \hat{R}_i \Leftarrow m_{\boldsymbol{\beta}}(S_i, A_i)$ by minimizing the mean-squared error between S_i', R_i and $\hat{S}_i', \hat{R}_i$. It should be noted that the learned model is task specific since the state representation is influenced by the reward function.

Planning: In the planning phase, a mini-batch of agent states S_i is randomly sampled from the set of recently visited states so that $p(S_i) \approx \mu_{\pi_{\boldsymbol{\theta}}}(S_i)$, where $\mu_\pi(S_i)$ is the agent state distribution under policy π. Once the agent states have been sampled, we then choose action a_i for each S_i according to the current policy $\pi_{\boldsymbol{\theta}}$. Next, given S_i and a_i, we predict the next agent state $\hat{S}_i'$ and the next reward $\hat{R}_i$ using the model $m_{\boldsymbol{\beta}}$. We fix the state update function $u_{\boldsymbol{\alpha}}$ and update $v_{\mathbf{w}}$ and $\pi_{\boldsymbol{\theta}}$.

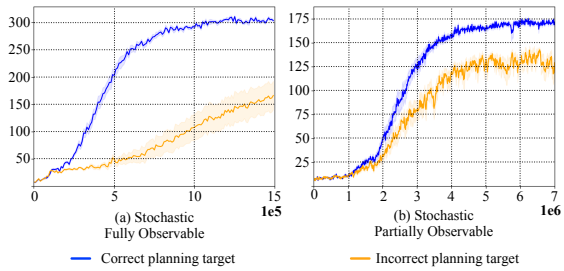
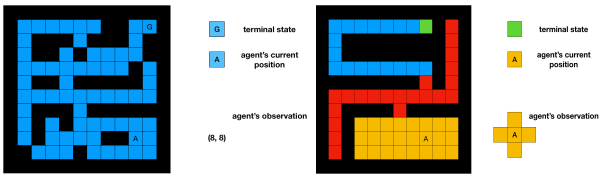


Figure 3. Planning with correct planning targets vs planning with incorrect planning targets: Learning curves on the stochastic variant of the fully and partially observable navigation tasks. x-axis represents the number of interactions with the environment, y-axis corresponds to the number of episodes completed in the last 10,000 environment interactions. Learning curves are averaged from 20 runs with random seeds



(a) Fully observable task (b) Partially observable task

Figure 4. In the fully observable task, the agent receives an observation which represents the x, y coordinates of its position in the maze. In the partially observable task, the agent receives an observation which includes the color information of the agent's current cell and its four neighbours. The black cells mark the obstacles while the colored cells mark the accessible regions. We formulate both tasks as episodic with a discount factor $\gamma = 0.99$. At the start of each episode, the agent is randomly placed in the maze. The reward is 0 at each time-step except when the agent moves to the terminate state, at which point the agent receives a reward of 1 and the episode terminates. We consider the deterministic and stochastic transition dynamics for both tasks. In stochastic case, the intended action is executed with probability 0.7 while a random action is executed with probability 0.3

We fuse the aforementioned algorithm in A3C with Generalized Advantage Estimation. which we call Asynchronous Advantage Actor Critic with Dyna-style planning (A3CD).

5. Experiments

We designed our experiments to answer the following questions: a) does our proposed method improve sample-efficiency when compared with the model-free methods? b) how robust is the learned model to the changing state update function? c) how does the proposed method compare with a method which uses a non-linear expectation model with a non-linear value function?

We evaluate our framework in fully and partially observable 2D navigation tasks, illustrated in 4.

5.1. Baselines

- Model-Free A3C with Generalized Advantage Estimation. It is important to note that our architecture reduces to this baseline if it does not perform the model learning and planning steps.

- Dyna-style planning with linear value function and the true model. This baseline demonstrates the improvement in sample efficiency if an accurate model is available.

- Dyna-style planning with non-linear value function and the true model. This baseline is used to empirically demonstrate that non-linear expectation model provides wrong planning targets if a non-linear value function is used.

5.2. Results

Model-free, true expectation model and learned expectation model. Results are summarized in Figure 2. We used the number of episodes completed in the last 10,000 steps as the evaluation metric. We first notice that in the stochastic and deterministic variants of both tasks, A3CD with true expectation model achieves the optimal performance considerably faster than the model-free A3C. This indicates that planning by only updating the value and policy functions, but not the state update function, can still obtain significant improvements over the model-free method.

We also notice that in both fully and partially observable tasks, the learned model performs almost as well as the true model, which suggests that the model can be fairly robust to the problem of the changing state update function. One exception is the performance of the learned model in the stochastic variant of the partially observable task. While the learned model does not reach the performance level of the true model, it still significantly outperforms the model-free baseline. This is expected, as the model learning is considerably more challenging if the task is both partially observable as well as stochastic.

Incorrect planning targets with non-linear value function. To empirically show that in a stochastic world, non-linear value function with non-linear expectation model leads to wrong planning targets which could consequently hurt the performance, we compare the performance of the linear and the non-linear value functions for planning with the true expectation model. While the former's planning process only updates parameters of the policy and the linear value functions and not the parameterized state update function, the latter updates all the parameters of the architecture, but with a wrong target. We compare the two methods in the stochastic variant of the fully and partially observable tasks. As illustrated in Figure 3, when the expectation model is used, planning updates with the non-linear value function performs considerably worse than the planning updates which involve the linear value function.

6. Conclusion

In this paper, we devised the first MBRL algorithm that is sound for non-linear expectation models and stochastic environments. The proposed approach learns the model online and uses it to simulate experience in the agent state space to update the value function and the policy. We also extend dyna to partially observable environments. Experiments on partially observable navigation tasks show the benefit of the proposed algorithm for sample efficient learning.

References

Bertsekas, D. P. and Tsitsiklis, J. N. (1995). Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE.

Betts, J. T. (1998). Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, 21(2):193–207.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

Deisenroth, M. and Rasmussen, C. E. (2011). 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.

Finn, C., Goodfellow, I., and Levine, S. (2016). Unsupervised learning for physical interaction through video prediction. In *Advances in neural information processing systems*, pages 64–72.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Leibfried, F., Kushman, N., and Hofmann, K. (2016). A deep learning approach for joint video frame and reward prediction in atari games. *arXiv preprint arXiv:1611.07078*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937.

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

Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871.

Oh, J., Singh, S., and Lee, H. (2017). Value prediction network. In *Advances in Neural Information Processing Systems*, pages 6120–6130.

Paduraru, C. (2007). Planning with approximate and learned models of markov decision processes. *These de maıtre, University of Alberta*.

Pan, Y., Zaheer, M., White, A., Patterson, A., and White, M. (2018). Organizing experience: a deeper look at replay mechanisms for sample-based planning in continuous state domains. In *IJCAI*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Sutton, R. (1991). Integrated modeling and control based on reinforcement learning and dynamic programming. In *Advances in Neural Information Processing Systems*.

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. P. (2012). Dyna-style planning with linear function approximation and prioritized sweeping. *arXiv preprint arXiv:1206.3285*.

Talvitie, E. (2017). Self-correcting models for model-based reinforcement learning. In *AAAI*, pages 2597–2603.