Learning Long-term Dependencies with Deep Memory States

Vitchyr Pong VITCHYR@BERKELEY.EDU
University of California, Berkeley

Shixiang Gu SG717@CAM.AC.UK
University of Cambridge, Max Planck Institute for Intelligent Systems

Sergey Levine SVLEVINE@EECS.BERKELEY.EDU
University of California, Berkeley

Abstract

Training an agent to use past memories to adapt to new tasks and environments is important for lifelong learning algorithms. Training such an agent to use its memory efficiently is difficult as the size of its memory grows with each successive interaction. Previous work has not yet addressed this problem, as they either use backpropagation through time (BPTT), which is computationally expensive, or truncated BPTT, which cannot learn long-term dependencies, to train recurrent policies. In this paper, we propose a reinforcement learning method that addresses the limitations of truncated BPTT by using a learned critic to estimate truncated gradients and by saving and loading hidden states outputted by recurrent neural networks. We present results showing that our algorithm can learn long-term dependencies while avoiding the computational constraints of BPTT. These results suggest that our method can potentially be used to train an agent that stores and effectively learns from past memories.

1. Introduction

While feed-forward, reactive policies can perform complex skills in isolation (Williams, 1992; Lange & Riedmiller, 2010; Mnih et al., 2015; Schulman et al., 2016), the ability to store past events in an internal memory is crucial for a wide range of behaviors. For example, a robot navigating a building might need to incorporate past observations to optimally estimate its location (Gupta et al.; Mnih et al., 2016), or remember a command previously issued by a person. Perhaps more importantly, long-term memory can be utilized for lifelong learning, where an agent uses past experiences to quickly modify its behavior in response to changing environments. Such recurrent meta-learning has been demonstrated on a variety of supervised learning tasks (Chen et al., 2017; Vinyals et al., 2016), and more recently applied to a variety of reinforcement learning tasks (Heess et al., 2015; Wang et al.; Duan et al., 2016a). However, realistic applications of policies with memory may demand particularly long-term memorization. For example, a robot tasked with setting the silverware on the table would need to remember where it last stored it, potentially hours or days ago.

Accepted at Lifelong Learning: A Reinforcement Learning Approach Workshop @ICML, Sydney, Australia, 2017. Copyright 2017 by the author(s).

This kind of long-term memorization is very difficult with current reinforcement learning methods. Specialized architectures have been developed that improve the capabilities of recurrent networks to store information (Hochreiter & Urgen Schmidhuber, 1997; Chen et al., 2017; Vinyals et al., 2016), but such methods still require back-propagation through time (BPTT) for training, which typically limits how far back the error is propagated to at most the length of the trajectory. Past this size, the gradient is truncated (Elman, 1990; Williams & Peng, 1990; Hausknecht & Stone, 2015). Truncating the gradient between when the policy must perform a crucial task (such as finding the silverware) and the observation that needs to be memorized to know which action to perform (the last location of the silverware) can make it impossible to successfully perform the task.

While some tasks may be solved by loading entire episodes into memory and avoiding truncation (Wierstra et al., 2007; Heess et al., 2015), a lifelong learning agent has no notion of episodes. Instead, a lifelong learning agent lives out a single episode that continuously grows. Computational constraints, both in terms of memory and practical training times, impose a fundamental limit on the memory capacity of neural network policies.

Rather than loading full episodes or truncating gradients, one can instead augment the original MDP with memory states (Peshkin et al., 2001; Zhang et al.). In addition to regular MDP actions, a policy outputs a vector called memory states, which it receives as input at the next time step. These memory states are equivalent to hidden states in normal recurrent neural network, but by interpreting memory states as just another part of the MDP state, recurrent policies can be trained using standard reinforcement learning methods, including efficient off-policy algorithms that can handle potentially infinite episode lengths (Mnih et al., 2015; Lillicrap et al., 2015; Gu et al., 2016). However, the use of memory states forces the learning algorithm to rely on the much less efficient gradient-free RL optimization to learn memorization strategies, rather than the low-variance gradients obtained from back-propagation through time (BPTT). For this reason, even truncated BPTT is usually preferred over the memory states approach when using model-free RL algorithms.

We propose a hybrid recurrent reinforcement learning algorithm that combines both memory states and BPTT. To obtain a practical algorithm that enables memorization over potentially unbounded episodes, we must use some form of memory states to manage computational constraints. However, we must also use BPTT as much as possible to make it feasible for the learner to acquire appropriate memorization strategies. Our actor-critic algorithm includes memory states and write actions, but performs analytic BPTT within each batch, loading subsequences for each training iteration. This approach allows us to use batches of reasonable size with enough subsequences to decorrelate each batch, while still benefiting from the efficiency of BPTT. Unfortunately, the use of memory states by itself is insufficient to provide for a Markovian state description, since an untrained policy may not store the right information in the memory. This makes it difficult to use memory states with a critic, which assumes Markovian state. To address this issue, we also propose a method for backpropagating Bellman error gradients, which encourages the policy to take write actions that reduce future Bellman error.

We describe the approach, and present preliminary results on simple recurrent memorization tasks that show that our method performs comparable to full BPTT. Our tasks require the learner to memorize information over long periods and learn an effective critic and actor model, suggesting that the hybrid approach may extend successfully to more challenging reinforcement learning scenarios.

2. Preliminaries

We consider a partially observed Markov Decision Problem (POMDP), which is made of a state space $\mathcal{S}$, an observation space $\mathcal{O}$, an action space $\mathcal{A}$, an initial state distribution $\mathbf{s}_1 \sim p_0(\cdot)$, a conditional observation probability $p_{\mathcal{O}|\mathcal{S}}(\mathbf{o}|\mathbf{s})$, and a transition dynamics distribution $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$. Let $\mathbf{o}_{1:t} = (\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_t)$ be a sequence of observations. A policy $\pi_\theta(\mathbf{a}|\mathbf{o}_{1:t})$ maps from a sequence of observation $\mathbf{o}_{1:t}$ to a distribution over actions $\mathbf{a}$ and is parameterized by θ. We denote a trajectory by $\tau = (\mathbf{s}_1, \mathbf{o}_1, \mathbf{a}_1, \mathbf{s}_2, \mathbf{o}_2, \mathbf{a}_2, ...)$. At each time step, the policy gets a reward of $r(\mathbf{s}_t, \mathbf{a}_t)$. The total discounted reward is $\sum_{k=t}^{\infty} \gamma^k r(\mathbf{s}_k, \mathbf{a}_k)$ where $\gamma \in (0,1)$ is a discount factor. The objective is to find the parameter θ that maximizes the policy's expected discounted return, given by

$$J(\theta) = \mathbb{E}_{\tau_\pi}\left[\sum_{k=t}^{\infty} \gamma^k r(\mathbf{s}_k, \mathbf{a}_k)\right]$$

where τ_π is the trajectory induced by following the (possibly deterministic) policy π_θ. Lastly, we define the Q function, defined as

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}_{\tau_\pi}\left[\sum_{k=1}^{\infty} \gamma^k r(\mathbf{s}_k, \mathbf{a}_k) \mid \mathbf{s}_0 = \mathbf{s}, \mathbf{a}_0 = \mathbf{a}\right]$$

which represents the expected value of taking action $\mathbf{a}$ at state $\mathbf{s}$ and following policy π thereafter. This Q function is not easy to compute and so in practice we approximate it with a function Q^ω parameterized by ω, using a neural network.

2.1. Recurrent Neural Networks

A popular approach to handle partial observability is to use recurrent neural networks (RNNs) to represent policies (Wierstra et al., 2007; Hausknecht et al., 2016; Oh et al., 2016). A RNN policy has two functions: an action function $\phi(\mathbf{o}_t, \mathbf{h}_t)$ and a hidden state function $\psi(\mathbf{o}_t, \mathbf{h}_t)$. At each time step t, a RNN policy takes as input observation $\mathbf{o}_t$ and hidden vector $\mathbf{h}_t$ and the action function outputs the next action of the policy, while the hidden state function outputs the next hidden state, $\mathbf{h}_{t+1} = \psi(\mathbf{o}_t, \mathbf{h}_t)$. The hidden vector is then received as input to the network at the next time step. Since $\mathbf{h}_t$ depends on $\mathbf{o}_{1:t}$, the next action is a function that depends on all of the previous actions. These policies may be trained with BPTT to solve tasks that require memory by loading sequence of observations and actions (Wierstra et al., 2007).

2.2. Memory States

An alternative formulation to train policies with memory capabilities is to interpret the hidden vectors outputted by the RNN as *memory states* and considering these memory states as another component of a new MDP (Peshkin et al., 2001; Zhang et al.). Specifically, at each time step t, the memory state m_t is appended to the observations seen in the original POMDP. The POMDP action space is also augmented by adding a memory write action, $\mathbf{w}_t$. This modification gives us new states and actions:

$$\widehat{\mathbf{s}}_t = \begin{bmatrix} \mathbf{o}_t \\ \mathbf{m}_t \end{bmatrix}, \quad \widehat{\mathbf{a}}_t = \begin{bmatrix} \mathbf{a}_t \\ \mathbf{w}_t \end{bmatrix}, \qquad \widehat{\mathbf{s}}_{t+1} = \begin{bmatrix} \mathbf{o}_{t+1} \\ \mathbf{m}_{t+1} = \mathbf{w}_t \end{bmatrix} \tag{1}$$

At each time step, the previous memory write action is copied directly to the next memory state, so that $\mathbf{m}_{t+1} = \mathbf{w}_t$. This augmented state space, action space, and transition gives us a new MDP. Intuitively, one hopes that the memory augmented state $\widehat{\mathbf{s}}$ captures all the information in the true state, $\mathbf{s}$. In this augmented MDP, Q is now a function of $\widehat{\mathbf{s}}$ and $\widehat{\mathbf{a}}$, and so we write $Q(\widehat{\mathbf{s}}, \widehat{\mathbf{a}})$. Similarly the policy will be written as $\pi(\widehat{\mathbf{s}})$.

While RNN policies and policies with memory states are computationally equivalent, they are conceptually quite different. A RNN policy is a function of a history of observations and must be trained with histories. With memory states, the policy is a function of the current observation and the current memory. When training RNN policies, truncated BPTT cannot learn long-term dependencies, but truncated BPTT allows us to exploit the known dynamics of the hidden states. In contrast, the memory states formulation instead assumes that the memory augmented MDP is completely Markovian and can therefore capture long-term dependencies. Unfortunately, this assumption is unlikely to be true at the start of training, since the memory state outputted by the policy depends on the initialization. We would like to combine the best of both methods: leverage BPTT and use a framework that can learn long-term dependencies.

3. Method

BPTT is an efficient algorithm for optimizing RNNs, but it requires knowing the gradient of the loss function with respect to each output of a RNN. In reinforcement learning, we only have access to samples to the loss function. Methods that train recurrent policies with BPTT (Wierstra & Alexander; Wierstra et al., 2007; Duan et al., 2016a; Heess et al., 2015) estimate this gradient by using the REINFORCE trick, or by training a critic and using its derivative with respect to the action, as shown in Figure 1a.

When episodes become too long to load full episodes for BPTT, truncated BPTT loads a sub-trajectory from time t to $t+k$: $(\mathbf{o}_t, \mathbf{m}_t, \mathbf{a}_t, \mathbf{w}_t, r_t, \ldots, \mathbf{o}_{t+k}, \mathbf{m}_{t+k}, \mathbf{a}_{t+k}, \mathbf{w}_{t+k}, r_{t+k})$ and initializes the RNN hidden state to all zeros, as shown in Figure 1b. However the future returns can substantially change depending on past observations, including observations that are not loaded into the current subtrajectory. Truncation makes it impossible to learn dependencies on past observations that were never loaded.

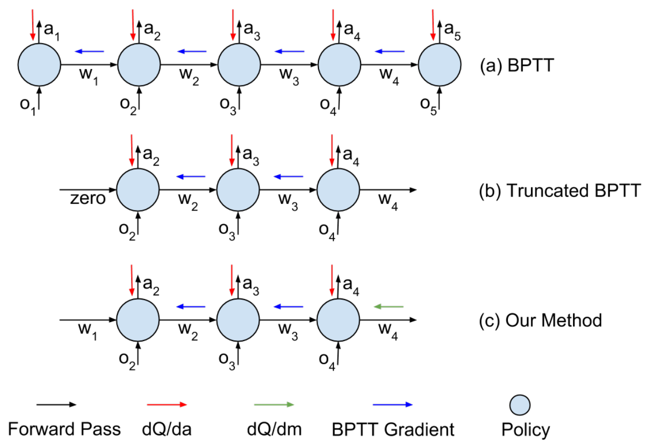


Figure 1. Different techniques for training RNN policies. At each time step, the policy receives an observation o_i and outputs both an action a_i and the next memory state m_{i+1}. An approximation of $\frac{dQ}{da}$ is used at each time step to supervise the policy action outputs. (a) BPTT requires loading full episodes. (b) Truncated BPTT approximate BPTT, but does not capture past information (the all zeros vector is loaded in) nor does it receive supervision for its final memory write action. (c) Our method captures past dependencies by loading memory states and approximates $\frac{dQ_\omega}{d\mathbf{w}}$ to supervise the final memory write action.

This approach also suffers from another issue: because the last write action of the loaded sub-trajectory has no supervision signal (i.e. $\frac{dL}{d\mathbf{w}_{t+k}} = 0$), the recurrent weight is not rewarded for minimizing future losses.

We show how to overcome both of these issues via a memory state formulation. First we reiterate that a RNN policy is computationally equivalent to a feed-forward policy that outputs actions and memory states that are given as input to the policy at the next time step. Rather than initializing the RNN hidden state to all zeros, we load the memory state that the policy outputted during the forward pass. If we assume that the loaded memory state summarizes the past (see Section 3.1 for how we enforce this), then this mechanism will allow the RNN to learn dependencies on that past information without needing to load all previous observations.

Furthermore, since the last write action $\mathbf{w}_{t+k}$ outputted by the network is simply part of the augmented MDP action space, our critic Q_ω is already a function of that memory state. Therefore, we use $\frac{dQ_\omega}{d\mathbf{w}}$ to estimate the final gradient $\frac{dL}{d\mathbf{w}_{t+k}}$. This is mathematically equivalent to training the policy to maximize the estimated Q value in our augmented state space. We do so by minimize the following loss function

$$\mathcal{L}_\pi(\pi_\theta) = -Q_\omega(\widehat{\mathbf{s}}_t, \pi_\theta(\widehat{\mathbf{s}}_t)) \tag{2}$$

with gradient descent.

We train the critic by minimizing the following squared Bellman error loss function

$$\mathcal{L}_Q(Q_\omega) = \frac{1}{T}\sum_t \|Q_\omega(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t) - \mathbf{y}_t\|_2^2 \tag{3}$$

where

$$\mathbf{y}_t = r(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t) + \gamma \max_{\widehat{\mathbf{a}}' \in \mathcal{A}} Q_\omega(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}') \tag{4}$$

$$= r(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t) + \gamma Q_\omega(\widehat{\mathbf{s}}_t, \arg\max_{\widehat{\mathbf{a}}' \in \mathcal{A}} Q_\omega(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}')) \tag{5}$$

$$\approx r(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t) + \gamma Q_\omega(\widehat{\mathbf{s}}_t, \pi_\theta(\widehat{\mathbf{s}}_t)) \tag{6}$$

Equations (3, 4, 5) define the typical Bellman-error that is minimized in Q-learning methods in our augmented MDP. Because we deal with continuous action spaces, the maximization in (5) is computationally expensive. So, we approximate the maximum with the current policy in Equation 6, as in neural fitted Q-iteration for continuous action (Hafner & Riedmiller, 2011).

3.1. Learning good memory states

We explain how we enforce the memory augmented states to be Markovian, which is important because Q-learning methods assume that the state is Markovian. Given a random initialization of a network, it is unlikely that the memory states will be Markovian and so the critic will have difficulty minimizing the Bellman error in Equation (3). Our insight is that we can encourage the policy to write Markovian memory state that make it easy for the critic to minimize Equation (3). We do so by making the policy also minimize the Bellman error of the critic. The policy minimizes the following loss function

$$\mathcal{L}_\pi(\pi_\theta) = -Q_\omega(\widehat{\mathbf{s}}_t, \pi_\theta(\widehat{\mathbf{s}}_t)) + \lambda \|Q_\omega(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t) - \mathbf{y}_t\|_2^2 \tag{7}$$

where $\mathbf{y}_t$ (defined in Equation (6)) depends on the policy. Since minimizing the Bellman error requires the memory augmented state space to be Markovian, the policy must learn to make the memory states Markovian, which is precisely what we want. The hyperparameter λ trades off how much the policy tries to maximize its immediate reward and how much it tries to make the memory states more useful for the critic.

3.2. Generating On-Policy Memory States

Our algorithm is derived to work in off-policy settings, but we suspect that it performs better if we use more on-policy data. Collecting more on-policy data normally requires sampling more data from the environment. However, since we know the dynamics of the memory write actions and memory states, we can simulate more data. When we train our policy on a subtrajectory minibatch, we unroll the policy for the length of that subtrajectory and get its outputted environment action and new memory write action at each time step. Subtrajectories are sampled from all valid subtrajectory of a full subtrajectory, and not, e.g., just from the beginning or end of the trajectory. We save these newly computed memory states back into the replay buffer. The final algorithm is summarized in 1.

Algorithm 1 Our Method

```
Initialize critic network $Q^\omega(\mathbf{a}_t, \mathbf{h}_t)$ and actor $\pi^\theta(\mathbf{h}_t)$
with params $\omega$ and $\theta$.
Initialize target networks $Q^{\omega'}$ and $\pi^{\theta'}$ with weights
$\omega' \leftarrow \omega$, $\theta' \leftarrow \theta$.
Initialize replay buffer R.
for episode $= 1, M$ do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  Initialize memory state to all zeros $\mathbf{m}_1 = 0$
  for $t = 1, T$ do
    Select augmented action $\mathbf{a}_t, \mathbf{w}_t = \pi_\theta(\widehat{\mathbf{s}}_t) + \mathcal{N}_t$
    $\mathbf{a}_t, r_t, \mathbf{o}_{t+1} = \text{step}(\mathbf{a}_t)$
    Construct $\widehat{\mathbf{a}}_t$, $\widehat{\mathbf{s}}_t$, and $\widehat{\mathbf{s}}_{t+1}$ using Equation 1
    Store $(\widehat{\mathbf{a}}_t, \widehat{\mathbf{s}}_t, \widehat{\mathbf{s}}_{t+1}, r_t)$ in $R$
    Sample $(\widehat{\mathbf{s}}^i_{t:t+K+1}, \widehat{\mathbf{a}}^i_{t:t+K}, r^i_{t:t+K})^N_{i=1}$ from $R$.
    Compute target values for each sample

        $\mathbf{y}^i_t = r^i_t + \gamma Q^{\omega'}(\widehat{\mathbf{s}}_t, \widehat{\mathbf{a}}_t)$.

    Perform gradient update for critic loss (3) and pol-
    icy loss (7), averaging across $i$ samples.
    Save computed write action $\mathbf{w}^i_t$ back into $R$.
    Update target networks with $\omega' \leftarrow \alpha\omega + (1-\alpha)\omega'$
    and $\theta' \leftarrow \alpha\theta + (1-\alpha)\theta'$
  end for
end for
```

4. Experiments

On a toy task, we demonstrate that our method can train policies with short subsequences to solve tasks that require long-term memory. In this problem, the observation space is $\mathcal{O} = \{-1, 0, 1\}$ and the action space is $[0, 1]$. At the first time step, the policy either receives $+1$ or -1. For the remaining time steps $t = 2, \ldots, H = 25$, the policy receives an observation of zero. At the last time step, the policy outputs a number between -1 and 1. The reward for the last time step is the policy's output times the initial observation. All other rewards are zero. To maximize its returns, the policy must learn to memorize the initial input and output a value of the same sign in the last time step. While supervised learning is able to solve this task easily, we explore how reinforcement learning algorithms perform on this problem.

In addition to this toy task, we tested our algorithm in a

Markov decision problem called "2D Target," where supervised learning could not be used. In this problem, the goal is to navigate a 2-dimensional point to a target circle. The action is a two-dimensional force applied to a point, and the observations are the x- and y-coordinate of the point, as well as whether or not the point is on the target. The field is a 10x10 arena (with walls stopping the point mass from existing the arena), the target is a circle with radius 2 whose location is randomly initialized every episode, and the forces and velocities are clipped to 1, where all distances are normalized. The agent receives a reward of 1 when it is on the target circle and otherwise receives a reward of zero. Only at the first time step, the agent receives the x- and y-coordinate of the center of the target. Without memory, the optimal strategy is to wander the entire arena until the agent stumbles upon the target. If the agent can learn to use its memory well, then it can learn to go directly to the target by remembering its first observation.

First, we tested our algorithm on both problems while varying the subtrajectory length. For comparison, we tested two reinforcement learning algorithms on this task: deep deterministic policy gradient (DDPG) Lillicrap et al. (2015), an off-policy actor-critic algorithm, and trust region policy optimization (TRPO) Schulman et al. (2015), a policy gradient method. We also trained these algorithms when we augmented the environment with memory write action and memory states and (for the 2D target task) when using their recurrent version, recurrent deterministic policy gradient Heess et al. (2015) and a recurrent TRPO implementation from (Duan et al., 2016b).

An important advantage of our algorithm over BPTT is that our algorithm does not need to load full trajectories into batches. This ability is especially important when the maximum size of a batch is fixed due to computational limits. To simulate this scenario, in all experiments, we trained our algorithm with a fixed batch size of 100 time steps while varying the length of the loaded subtrajectories. For example, if each loaded subtrajectory has length 5, then we can load 20 subtrajectories, whereas if we load full trajectories (length 25), we can only load 4 subtrajectories.

Our method differs from truncated BPTT in two regards: (1) we load the saved memory state rather than initializing the memory sate to zero and (2) the policy uses the gradient $\frac{dQ_\omega}{d\mathbf{w}}$ to supervise its memory write actions. We conducted an ablative analysis to understand the contribution of each modification.

4.1. Results

While both DDPG and TRPO have been shown to solve challenging, fully observed tasks, we see in Figures 2 and 3 that these methods cannot solve the partially observed toy task or 2D Target task. Furthermore, DDPG and TRPO

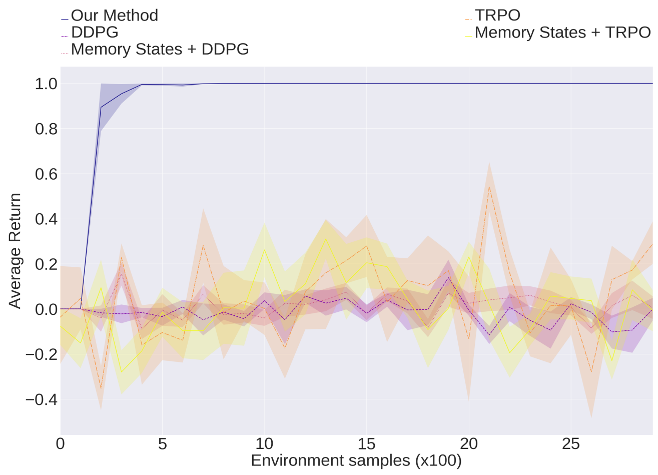


Figure 2. Average return on toy task vs. number of sampled time steps. Error bars show standard deviation across 5 different random seeds. Performance of our algorithm with full BPTT (Our Method), deep deterministic policy gradient (DDPG), DDPG with memory states (Memory States + DDPG), trust region policy optimization (TRPO), and trust region policy optimization with memory states (Memory States + TRPO). Other algorithms fail to solve this partially observed task, even when memory states are added to the problem.

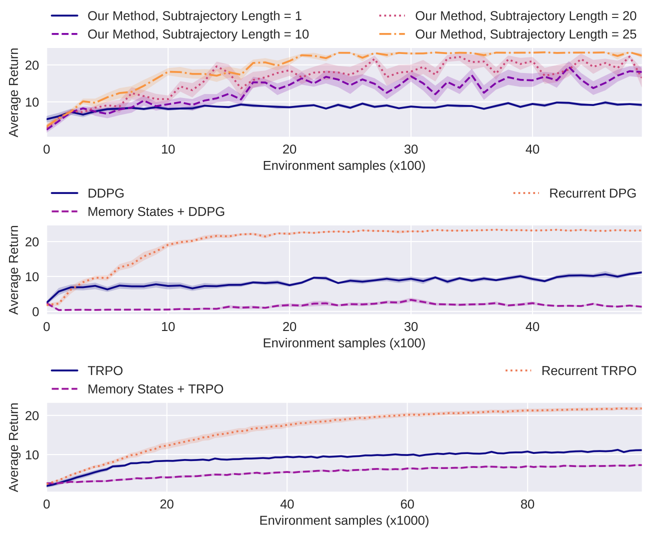
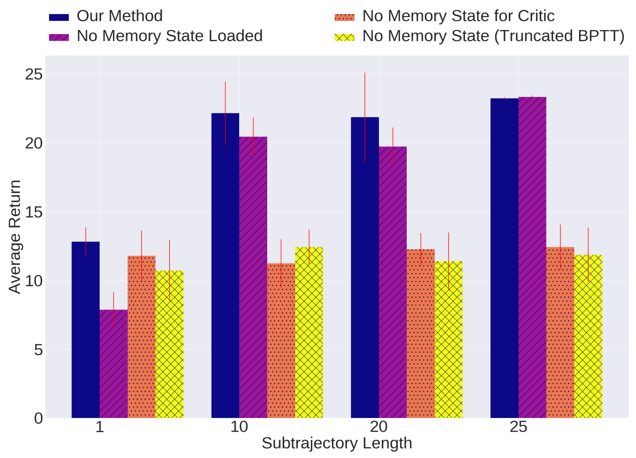
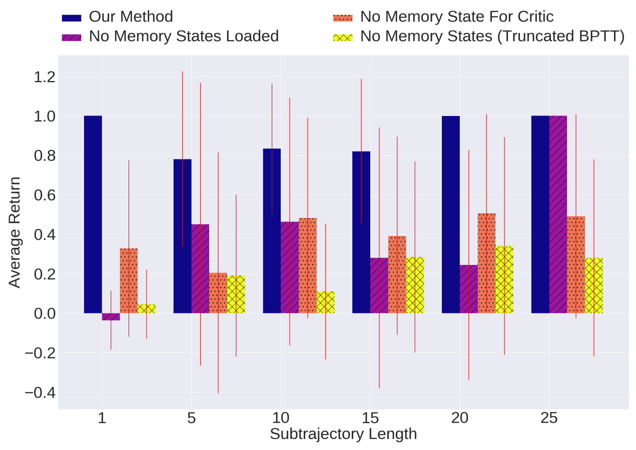


Figure 3. Average return on 2D Target vs number of sampled time steps for variants of our algorithm (top row), deterministic policy gradient (middle row), and trust region policy optimization (bottom row). Error bars show standard deviation across 5 different random seeds. Note that TRPO uses 20 time more samples to reach a similar performance.



(a) 2D Target Task



(b) Toy Task

Figure 4. Average return vs. subtrajectory length for different variations of our algorithm after (a) 100,000 samples for the 2D Target task and (b) 3000 samples for the toy task. Error bars show standard deviation across 5 different random seeds. Our method performs the best across varying subtrajectory lengths. Both loading saved memory states and making the critic a function of the memory state is important.

were not able to solve the task even with the augmented memory state space, suggesting that simply augmenting the state space is insufficient to make these algorithms solve partially observed tasks. In contrast, our method is able to consistently solve the tasks when using full subtrajectories.

When the subtrajectory length is less than the full sequence, we see that our method is still able to solve the task as shown in Figure 4. While we expected the performance to monotonically decrease as we used shorter and shorter subsequence lengths, we were surprised to see that our algorithm performed very well when the subtrajectory length was one for the toy task. We hypothesize that this result is because using many, small subtrajectories allows the samples to be maximally decorrelated, but we note that this trend was not observed in the more difficult 2D target task.

Figure 4 also shows that our differences from truncated BPTT are important. Without either modification, the performance of our method quickly deteriorates as the subtrajectory length decrease for the toy task. Though less drastic, this trend is also observed in the 2D Target task. We also note that while no variations do well when no BPTT is used (the subtrajectory length is 1), our method still performs well even when the length of the subtrajectory is less than half of the entire episode length.

5. Conclusion

In this paper, we reformulate a partially observed MDP as a fully observed MDP with an augmented state and action space. However, we show that it is difficult to rely solely on high-variance reinforcement learning algorithms to optimize a policy in this new MDP. We show how to combine Q-learning for continuous actions with BPTT to effectively train recurrent policies. Our results show that it is possible to train recurrent policies to learn long-term dependencies without ever needing to load full trajectories, even if the dependencies span the entire duration of the trajectory. We show that our method is applicable in both supervised learning and reinforcement learning tasks as a means of learning long-term dependencies without needing to load full trajectories.

References

Chen, Steven W, Atanasov, Nikolay, Khan, Arbaaz, Karydis, Konstantinos, Lee, Daniel D., and Kumar, Vijay. Neural Network Memory Architectures for Autonomous Robot Navigation. 2017.

Duan, Yan, Chen, Xi, Schulman, John, and Abbeel, Pieter. Benchmarking Deep Reinforcement Learning for Continuous Control. *arXiv*, 48:14, 2016a.

Duan, Yan, Schulman, John, Chen, Xi, Bartlett, Peter, Sutskever, Ilya, and Abbeel, Pieter. RL^2: Fast Reinforcement Learning Via Slow Reinforcement Learning. *arXiv*, pp. 1–14, 2016b. ISSN 0004-6361. doi: 10.1051/0004-6361/201527329.

Elman, Jeffrey L. Finding structure in time. *Cognitive Science*, 14(1 990):179–211, 1990. ISSN 03640213.

Gu, Shixiang, Lillicrap, Timothy, Sutskever, Ilya, Levine, Sergey, and Com, Slevine@google. Continuous Deep Q-Learning with Model-based Acceleration. *Icml*, 2016.

Gupta, Saurabh, Davidson, James, Levine, Sergey, Sukthankar, Rahul, and Malik, Jitendra. Cognitive Mapping and Planning for Visual Navigation.

Hafner, Roland and Riedmiller, Martin. Reinforcement learning in feedback control : Challenges and benchmarks from technical process control. *Machine Learning*, 84(1-2):137–169, 2011. ISSN 08856125.

Hausknecht, Matthew and Stone, Peter. Deep Recurrent Q-Learning for Partially Observable MDPs. 2015.

Hausknecht, Matthew, Stone, Peter, and Mc, Off-policy. On-Policy vs. Off-Policy Updates for Deep Reinforcement Learning. *Ijcai*, 2016.

Heess, Nicolas, Hunt, Jonathan J, Lillicrap, Timothy P, and Silver, David. Memory-based control with recurrent neural networks. *arXiv*, pp. 1–11, 2015.

Hochreiter, Sepp and Urgen Schmidhuber, J. LONG SHORT-TERM MEMORY. *Neural Computation*, 9(8): 1735–1780, 1997.

Lange, Sascha and Riedmiller, Martin. Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, jul 2010. ISBN 978-1-4244-6916-1.

Lillicrap, Timothy P., Hunt, Jonathan J., Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, pp. 1–14, 2015. ISSN 1935-8237.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei a, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 0028-0836.

Mnih, Volodymyr, Badia, Adrià Puigdomènech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy P, Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous Methods for Deep Reinforcement Learning. *arXiv*, 48: 1–28, 2016.

Oh, Junhyuk, Chockalingam, Valliappa, Singh, Satinder, and Lee, Honglak. Control of Memory, Active Perception, and Action in Minecraft. *arXiv*, 2016.

Peshkin, Leonid, Meuleau, Nicolas, and Kaelbling, Leslie. Learning Policies with External Memory. *Sixteenth International Conference on Machine Learning*, (March): 8, 2001. ISSN 1098-6596.

Schulman, John, Levine, Sergey, Jordan, Michael, and Abbeel, Pieter. Trust Region Policy Optimization. *Icml-2015*, pp. 16, 2015. ISSN 2158-3226.

Schulman, John, Moritz, Philipp, Levine, Sergey, Jordan, Michael, and Abbeel, Pieter. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv*, pp. 1–9, 2016.

Vinyals, Oriol, Blundell, Charles, Lillicrap, Timothy, Kavukcuoglu, Koray, and Wierstra, Daan. Matching Networks for One Shot Learning. *arXiv*, 2016.

Wang, Jx, Kurth-Nelson, Z, Tirumala, D, Soyer, H, Leibo, Jz, Munos, R, Blundell, C, Kumaran, D, and Botvinick, M. LEARNING TO REINFORCEMENT LEARN.

Wierstra, Daan and Alexander, F. Recurrent Policy Gradients. (May 2009).

Wierstra, Daan, Foerster, Alex, Peters, Jan, and Schmidthuber, Juergen. Solving Deep Memory POMDPs with Recurrent Policy Gradients. *Icann2007*, 1(1):697–706, 2007. ISSN 03029743.

Williams, Ronald J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. 1992.

Williams, Ronald J and Peng, Jing. An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Appears in Neural Computation*, (2): 490–501, 1990. ISSN 0899-7667.

Zhang, Marvin, Mccarthy, Zoe, Finn, Chelsea, Levine, Sergey, Abbeel, Pieter, and Sep, L G. Learning Deep Neural Network Policies with Continuous Memory States.