2016 IEEE International Conference on Robotics and Automation (ICRA)
Stockholm, Sweden, May 16-21, 2016

Learning Deep Neural Network Policies with Continuous Memory States

Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, Pieter Abbeel

Abstract— **Policy learning for partially observed control tasks requires policies that can remember salient information from past observations. In this paper, we present a method for learning policies with internal memory for high-dimensional, continuous systems, such as robotic manipulators. Our approach consists of augmenting the state and action space of the system with continuous-valued memory states that the policy can read from and write to. Learning general-purpose policies with this type of memory representation directly is difficult, because the policy must automatically figure out the most salient information to memorize at each time step. We show that, by decomposing this policy search problem into a trajectory optimization phase and a supervised learning phase through a method called guided policy search, we can acquire policies with effective memorization and recall strategies. Intuitively, the trajectory optimization phase chooses the values of the memory states that will make it easier for the policy to produce the right action in future states, while the supervised learning phase encourages the policy to use memorization actions to produce those memory states. We evaluate our method on tasks involving continuous control in manipulation and navigation settings, and show that our method can learn complex policies that successfully complete a range of tasks that require memory.**

I. Introduction

Reinforcement learning (RL) and optimal control methods have the potential to allow robots to autonomously discover complex behaviors. However, robotic control problems are often continuous, high dimensional, and partially observed. The partial observability in particular presents a major challenge. Partial observability has been tackled in the context of POMDPs by using a variety of model-based approximations [1]. However, despite recent progress [2], [3], [4], [5], [6], learning the state representation, the dynamics and the observation model together remains challenging. Model-free policy search algorithms have been successfully used to sidestep the need for learning dynamics and observation models, by optimizing policies directly through system interaction [7]. However, partially observed domains, where reactive policies are insufficient, necessitate the use of internal memory. Finite state controllers have previously been applied to smaller RL tasks where value function approximation is practical [2], and policy gradient methods have been extended to recurrent neural networks (RNNs) [8]. However, effective training of complex, high-dimensional, general-purpose policies with internal memory still presents a tremendous challenge.

In this paper, we investigate a simple approach for endowing policies with memory, by augmenting the state space to include memory states that can be written to by the policy. Naïvely using such a state representation with standard policy search algorithms is quite challenging, because the algorithm must simultaneously figure out how to use the memory states to choose the action and how to store the right information into these states so that it can be recalled later. The computations needed to make such decisions about memory states require a complex, nonlinear policy structure. Such policies are difficult to efficiently learn with model-free methods, while model-based methods also require a model of the system dynamics, which can be difficult to obtain [7]. We show how the guided policy search algorithm can be adapted to the task of training policies with internal memory.

In guided policy search, the policy is optimized using supervised learning [9], [10]. The supervision is provided by using a simple trajectory-centric reinforcement learning algorithm to individually solve the task from a collection of fixed initial states. This trajectory-centric "teacher" resembles trajectory optimization. Since each teacher only needs to solve the task from a single initial state, it is faced with a much easier problem. The final policy is trained with supervised learning, which allows us to use a nonlinear, high-dimensional representation for this final policy, such as a multilayer neural network, in order to learn complex behaviors with good generalization. A key component in guided policy search is adaptation between the trajectories produced by the teacher and the final policy. This adaptation ensures that, at convergence, the teacher does not take actions that the final policy cannot reproduce. This is realized by an alternating optimization procedure, which iteratively optimizes the policy to match each teacher, while the teachers adapt to gradually match the behavior of the final policy.

To incorporate memory states into this method, we add the memory states to both the trajectory-centric teacher and the final neural network policy. Since the trajectories are adapted to the neural network policy, the teacher selects memory states that will cause the neural network to take the right action, essentially telling the network which information should be memorized to achieve good performance. Because of this, the neural network only needs to learn how to reproduce the memory states chosen by the teacher. The teacher effectively shows the neural network which information needs to be written into the memory, and the network need only figure out how to obtain this information from the observations.

Our experimental results show that our method can be used to learn a variety of continuous control tasks in manipulation and navigation settings. For example, we show how memory states can allow a simulated robotic manipulator to remember the target position for a peg insertion task, or place plates and bottles into both vertical and horizontal slots, by remembering past sensory inputs from contacts

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94709

978-1-4673-8026-3/16/$31.00 ©2016 IEEE

that allow it to determine the orientation of the opening. We also show that using memory states with guided policy search outperforms other algorithms and representations, including policies represented by LSTM neural networks and alternative policy search methods that do not use a trajectory-centric teacher to optimize the memory state values.

II. Related Work

While a complete survey of reinforcement learning methods for partially observed problems is outside the scope of the paper, we highlight several relevant research areas in this section. Discrete partially observed tasks have been tackled using a variety of reinforcement learning and dynamic programming methods [11], [1]. While such methods have been extended to small continuous spaces [12], they are difficult to scale to the kinds of large state spaces found in most robotic control tasks. In these domains, methods based on direct policy search are often preferred, due to their ability to scale gracefully with task dimensionality [7]. While most policy search methods are concerned with reactive policies, a number of methods have been proposed that augment the policy with internal state, including methods based on finite state controllers [2], [4] and explicit memory states that the policy can alter using memory storage actions [3]. However, these methods have been evaluated only in small or discrete settings. While our approach also supplies the policy with internal memory states and explicit actions that can be used to alter that state, our memory and storage actions are continuous, and our experiments show that our method can scale to high-dimensional problems that are representative of real-world robotic control tasks.

Taken together with their internal memory states, our policies can be regarded as a type of recurrent neural network (RNN). Previous work has proposed training RNN policies using likelihood ratio methods and backpropagation through time [8]. However, this approach suffers from two challenges: the first is that model-free likelihood ratio methods, such as recurrent policy gradients, are difficult to scale to policies with more than a few hundred parameters [7], which makes it hard to apply the method to complex tasks that require flexible, high-dimensional policy representations, and the second is that optimizing RNNs with backpropagation through time is prone to vanishing and exploding gradients [13]. While specialized RNN representations such as LSTMs [14] or GRUs [15] can mitigate these issues, we show that we can obtain better results by training the policy to manipulate the memory states through explicit memory actions, without using backpropagation through time. To that end, we extend the guided policy search algorithm to train policies with memory states and memory actions.

The guided policy search algorithm used in this work is most similar to the method proposed by Levine et al. [16], [10]. This approach was proposed in the context of robotic control, and has been shown to achieve good results with complex, high-dimensional feedforward neural network policies. The central idea behind guided policy search is to decompose the policy search problem into alternating trajectory optimization and supervised learning phases, where trajectory optimization is used to find a solution to the control problem and produce training data that is then used in the supervised learning phase to train a nonlinear, high-dimensional policy. By training a single policy from multiple trajectories, guided policy search can produce complex policies that generalize effectively to a range of initial states. Previous work has only applied guided policy search to training reactive feedforward policies, since the algorithm assumes that the policy is Markovian. We show the BADMM-based guided policy search method [10] can be extended to handle continuous memory states. The memory states are added to the state of the system, and the policy is tasked both with choosing the action and modifying the memory states. Although the resulting policy can be viewed as an RNN, we do not need to perform backpropagation through time to train the recurrent connections inside the policy. Instead, the memory states are optimized by the trajectory optimization algorithm, which intuitively seeks to set the memory states to values that will allow the policy to take the appropriate action at each time step, and the policy then attempts to mimic this behavior in the supervised learning phase.

III. Background and Preliminaries

The aim of our method is to control a partially observed system in order to minimize the expectation of a cost function over the entire execution of a policy $\pi_\theta(\mathbf{u}_t|\mathbf{o}_1, \ldots, \mathbf{o}_t)$, given by $E_{\pi_\theta}[\sum_{t=1}^{T} \ell(\mathbf{x}_t, \mathbf{u}_t)]$ in the finite-horizon episodic setting. Here, $\mathbf{x}_t$ denotes the true state of the system, $\mathbf{u}_t$ denotes the action, $\mathbf{o}_t$ denotes the observation, and $\ell(\mathbf{x}_t, \mathbf{u}_t)$ is the cost function that specifies the task. For example, in the case of robotic control, $\mathbf{u}_t$ might correspond to the torques at the robot's motors, $\mathbf{x}_t$ might be the configuration of the robot and its environment, including the positions of task-relevant objects, and $\mathbf{o}_t$ might be the readings from the robot's sensors, such as joint encoders that provide the angles of the joints, or even images from a camera. The policy $\pi_\theta(\mathbf{u}_t|\mathbf{o}_1, \ldots, \mathbf{o}_t)$ specifies a distribution over actions conditioned on the current and previous observations. This policy is parameterized by θ. We are particularly concerned with tasks where the current observation $\mathbf{o}_t$ by itself is not sufficient for choosing a good action $\mathbf{u}_t$, and the policy must integrate information from the past to succeed. Such tasks require policies with internal state, which can be used to remember past observations and act accordingly. To optimize policies with memory, we build on the guided policy search algorithm presented by Levine et al. [10], which we summarize briefly in this section. This algorithm optimizes reactive policies of the form $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$. We discuss in Section IV-B how it can be adapted to train policies with memory.

A. Guided Policy Search

Guided policy search is a policy optimization algorithm that transforms the policy search task into a supervised learning problem, where supervision is provided by a set of

Algorithm 1 Partially observed guided policy search

for iteration $k = 1$ to K **do**
  Run each $p_i(\mathbf{u}_t|\mathbf{x}_t)$ to generate samples $\{\tau_j\}$
  Fit local linear dynamics $\hat{p}_i(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ around each $p_i(\mathbf{u}_t|\mathbf{x}_t)$ using $\{\tau_j\}$
  for inner iteration $l = 1$ to L **do**
    Optimize each $p_i(\mathbf{u}_t|\mathbf{x}_t)$ using fitted dynamics to minimize cost and match $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$
    Optimize $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ to match all distributions $p_i(\mathbf{u}_t|\mathbf{x}_t)$ along each sample trajectory τ_j
  end for
end for

simple trajectory-centric controllers, denoted $p_i(\mathbf{u}_t|\mathbf{x}_t)$, that are each optimized independently on separate instances of the task, typically corresponding to different initial states. There are two main benefits to this approach: the first is that, by requiring each trajectory-centric controller to solve the task from only a specific initial state, relatively simple controllers can be used that admit very efficient reinforcement learning methods. The second benefit is that, since the final policy is optimized with supervised learning methods, it can admit a complex, highly expressive representation without concern for the usual challenges associated with optimizing high-dimensional policies [7]. Intuitively, the purpose of the trajectory-centric controllers is to determine how to solve the task from specific states, while the purpose of the final policy is to generalize these controllers and succeed from a variety of initial states. The partially observed variant of guided policy search, which we build off of, takes this idea further, by also providing a different input to the trajectory-centric controllers compared to the policy. In this method, the trajectory-centric controllers are trained under full state observation, while the policy is trained to mimic these controllers using only the observations $\mathbf{o}_t$ as input. This forces the policy to handle partial observation, while keeping the task easy for the trajectory-centric controllers. This type of instrumented setup is natural for many robotic tasks, where training is done in a known laboratory setting, while the final policy must succeed under a variety of uncontrolled conditions. However, this method does not itself provide a way of handling internal memory.

The partially observed guided policy search method is summarized in Algorithm 1. At each iteration of the algorithm, samples are generated using each of the trajectory-centric controllers $p_i(\mathbf{u}_t|\mathbf{x}_t)$.[1] While a variety of representations for these controllers are possible, linear-Gaussian controllers of the form $p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\mathbf{K}_t\mathbf{x}_t + \mathbf{k}_t, \mathbf{C}_t)$ admit a particularly efficient optimization procedure based on iterative refitting of local linear dynamics [9]. Once these dynamics are fitted, the algorithm takes L inner iterations (4 in our implementation). These iterations alternate between optimizing each trajectory-centric controller $p(\mathbf{u}_t|\mathbf{x}_t)$ using a variant of LQR under the fitted dynamics, and optimizing the policy $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ to match the actions taken by the trajectory-centric controllers at each observation $\mathbf{o}_t^i$ encountered along the sampled trajectories. The controllers are optimized to minimize their expected cost $E_p[\ell(\tau)]$, as well as minimize their deviation from the policy, measured in terms of KL-divergence. The policy is optimized to minimize the KL-divergence from the controllers. This alternating optimization ensures that the trajectory-centric controllers and the policy agree on the same actions. In general, supervised learning is not guaranteed to produce good long-term policies, since errors in fitting the action at each time step accumulate over time [17]. Formally, the issue is that the policy will not have the same state visitation frequency as the controllers it is trained on. The alternating optimization addresses this by gradually forcing the controllers and policy to agree. To ensure agreement, guided policy search uses Lagrange multipliers on the means of the policy and the controllers, which are updated every iteration. The full details of this method, including the objectives for controller and policy optimization, are derived in previous work [10].

[1] We will drop the subscript i from $p_i(\mathbf{u}_t|\mathbf{x}_t)$ in the remainder of the paper for clarity of notation, but all of the exposition extends trivially to the case of multiple trajectory-centric controllers.

B. Trajectory-Centric Reinforcement Learning

In guided policy search, the linear-Gaussian controllers $p(\mathbf{u}_t|\mathbf{x}_t)$ are optimized with respect to the cost $\ell(\mathbf{x}_t, \mathbf{u}_t)$, as well as an additional term that penalizes deviation from the policy $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$. This term consists of the KL-divergence between $p(\mathbf{u}_t|\mathbf{x}_t)$ and $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ with a weight ν_t, as well as a Lagrange multiplier $\lambda_{\mu t}$ on the mean action. Together, the cost and the penalty form the following objective:

$$L(p) = E_{p(\mathbf{x}_t, \mathbf{u}_t)}[\ell(\mathbf{x}_t, \mathbf{u}_t) - \mathbf{u}_t^{\mathrm{T}}\lambda_{\mu t} + \nu_t D_{\mathrm{KL}}(p(\mathbf{u}_t|\mathbf{x}_t)\|\pi_\theta(\mathbf{u}_t|\mathbf{x}_t))].$$

The linear-Gaussian controllers $p(\mathbf{u}_t|\mathbf{x}_t)$ can be optimized in a variety of ways, including offline trajectory optimization methods with known models [18] and trajectory-centric reinforcement learning [9]. We adopt the latter approach in this work, which we briefly summarize in this section.

When the dynamics are locally smooth, a linear-Gaussian controller of the form $p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\mathbf{K}_t\mathbf{x}_t + \mathbf{k}_t, \mathbf{C}_t)$ can be viewed as inducing a mean trajectory with some linear feedback for stabilization. Hence, we refer to the process of learning such controllers as trajectory-centric or, more simply, as trajectory optimization. An efficient way to optimize these controllers is to draw samples from the current $p(\mathbf{u}_t|\mathbf{x}_t)$, fit time-varying linear-Gaussian dynamics to these samples of the form $\hat{p}(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(f_{\mathbf{x}t}\mathbf{x}_t + f_{\mathbf{u}t}\mathbf{u}_t + f_{ct}, \mathbf{F}_t)$, compute a local second-order Taylor expansion of the cost $\ell(\mathbf{x}_t, \mathbf{u}_t)$, and then optimize the controller $p(\mathbf{u}_t|\mathbf{x}_t)$ using the LQR dynamic programming algorithm. As described in previous work, this approach can achieve sample-efficient learning for a variety of robotic manipulation skills [9], [16], but it requires an additional constraint to ensure that the optimized controller remains in the region where the estimated dynamics $\hat{p}(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ are valid. This can be done by constraining the KL-divergence between the new controller $p(\mathbf{u}_t|\mathbf{x}_t)$ and the previous controller $\bar{p}(\mathbf{u}_t|\mathbf{x}_t)$, which

generated the samples that were used to fit $\hat{p}(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$. The corresponding optimization problem is given by

$$\min_p L(p) \text{ s.t. } D_{\text{KL}}(p(\tau)\|\bar{p}(\tau)) \leq \epsilon,$$

where $p(\tau)$ is the trajectory distribution induced by $p(\mathbf{u}_t|\mathbf{x}_t)$ and dynamics $\hat{p}(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$. Using KL-divergence constraints for controller optimization has been proposed in a number of previous works [19], [20], [21], but in the case of linear-Gaussian controllers, we can use a modified LQR algorithm to solve this problem. We refer the reader to previous work for details [9].

C. Recurrent Neural Networks

In order to avoid task-specific manual engineering of the policy class, guided policy search is often used with general-purpose function approximators such as large neural networks. One way to integrate memory into such policies is to use recurrent neural networks (RNNs). Unlike feedforward networks, RNNs can maintain a memory of past observations through their hidden states, which are propagated forward in time according to the hidden state dynamics.

We can define an RNN with inputs $\mathbf{o}_t$, outputs $\mathbf{u}_t$, and internal state $\mathbf{h}_t$ with two functions: an output function $\phi(\mathbf{o}_t, \mathbf{h}_t) = \mathbf{u}_t$ and a dynamics function $\psi(\mathbf{o}_t, \mathbf{h}_t) = \mathbf{h}_{t+1}$. In practice, ϕ and ψ might share some parameters, but viewing them as separate functions will make it convenient for us to compare standard RNNs with our memory states, which we describe in the next section.

RNNs are typically trained by viewing them as one large neural network and computing the gradient of the parameters with respect to the loss by using backpropagation through time. However, learning long-term temporal dynamics is still very difficult for RNNs, since backpropagation through time can lead to vanishing and exploding gradients. Many solutions have been proposed for these issues. One popular solution consists of altering the architecture of the network to make optimization easier, with the LSTM architecture being particularly popular. We therefore evaluate such an architecture as the baseline in our experiments in Section V.

Guided policy search provides us with an easier and more effective method for training such policies, by including the hidden states (referred to as memory states for clarity) directly into the state of the dynamical system. This avoids the need for using backpropagation through time, and instead uses trajectory optimization to optimize the memory state values. This approach, which we describe in detail below, achieves significantly better results in our experiments, and has a number of appealing computational benefits.

IV. Memory States

Instead of directly optimizing RNNs with backpropagation through time, we consider a different method for integrating memory into the policy. In our approach, the memory states $\mathbf{h}_t$ are directly concatenated to the physical state of the system $\mathbf{x}_t$ and the observation $\mathbf{o}_t$, to produce an augmented state $\tilde{\mathbf{x}}_t$ and augmented observation $\tilde{\mathbf{o}}_t$. We also concatenate

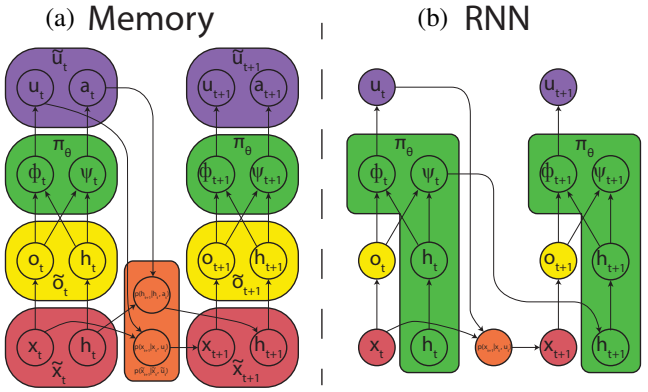

Fig. 1. Diagram comparing memory states (a) and RNNs (b). In the case of memory states, both ϕ and ψ are incorporated into the policy, rather than treating ψ as a hidden state dynamics function. This makes the apparent dynamics of the memory states independent of the network parameters, making it easy to apply guided policy search. Note that the computational units ϕ and ψ are identical in both cases, the only difference is in whether $\mathbf{h}_t$ is considered to be part of the policy or part of the state.

memory writing actions $\mathbf{a}_t$ to the action $\mathbf{u}_t$ to produce an augmented action $\tilde{\mathbf{u}}_t$. The entire system is as follows:

$$\tilde{\mathbf{x}}_t = \begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_t\end{bmatrix} \qquad \tilde{\mathbf{o}}_t = \begin{bmatrix}\mathbf{o}_t \\ \mathbf{h}_t\end{bmatrix} \qquad \tilde{\mathbf{u}}_t = \begin{bmatrix}\mathbf{u}_t \\ \mathbf{a}_t\end{bmatrix}.$$

The dynamics factorize according to

$$p(\tilde{\mathbf{x}}_{t+1}|\tilde{\mathbf{x}}_t, \tilde{\mathbf{u}}_t) = p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t),$$

because the action $\mathbf{u}_t$ only affects the physical state $\mathbf{x}_t$, and the memory writing actions $\mathbf{a}_t$ only modify the memory states $\mathbf{h}_t$. There are various ways to choose $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$ depending on the semantics of $\mathbf{a}_t$. In this work, we define $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t) = \mathcal{N}(\mathbf{h}_t + \mathbf{a}_t, \sigma^2\mathbf{I})$, where σ^2 is chosen to be a small constant (10^{-6} in our implementation) to ensure that the trajectory distributions remains well-conditioned.

Training a policy $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{o}}_t)$ on this augmented dynamical system produces a policy that, in principle, can use the memory actions $\mathbf{a}_t$ to write to the memory states $\mathbf{h}_t$. In practice, the particular choice of policy optimization algorithm makes a significant difference in how well the policy can utilize the memory states, since there is no guidance on how they should be used, aside from overall task performance. In the next section, we describe the relationship between these policies and RNNs, and in Section IV-B, we will describe how the guided policy search algorithm can train policies that effectively utilize memory states. In Section V we will show that this approach can produce effective policies that succeed on a range of simulated manipulation and navigation tasks that require memory.

A. Comparison of Memory States and RNNs

Policies of the form $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$, that use states and actions augmented with memory, are equivalent to RNNs when combined with the memory state dynamics $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$. In fact, these RNNs are in general stochastic, though we use linear-Gaussian memory dynamics $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$ with a small variance, as described in the previous section, which makes them effectively deterministic in our implementation. Note, however, that $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ by itself is not recurrent. This

distinction is illustrated in Figure 1, which compares the structure of an RNN with output function $\phi(\mathbf{o}_t, \mathbf{h}_t)$ and dynamics function $\psi(\mathbf{o}_t, \mathbf{h}_t)$ to the policy $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ with memory state dynamics $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$.

Aside from the stochastic aspects of $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ and $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$, which become neglible as the variance of both functions goes to zero, the relationship between this structure and the RNN is that $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ contains both $\phi(\mathbf{o}_t, \mathbf{h}_t)$ and $\psi(\mathbf{o}_t, \mathbf{h}_t)$. For example, when $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ is Gaussian, with a mean that depends on $\tilde{\mathbf{x}}_t$ and a constant covariance, and $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$ has the form in the previous section, we have

$$E_{\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)}[\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t] = \begin{bmatrix} \phi(\mathbf{o}_t, \mathbf{h}_t) \\ \psi(\mathbf{o}_t, \mathbf{h}_t) - \mathbf{h}_t \end{bmatrix},$$

so that the policy outputs the action $\phi(\mathbf{o}_t, \mathbf{h}_t)$ and the next hidden state is $\psi(\mathbf{o}_t, \mathbf{h}_t)$. Thus, we see that any RNN can be encoded as a non-recurrent policy with memory states. Furthermore, when $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ and $p(\mathbf{h}_{t+1}|\mathbf{h}_t, \mathbf{a}_t)$ have non-negligible stochasticity, memory states can be used to encode stochastic recurrent networks. While the variance of the policies in our experiments is independent of $\tilde{\mathbf{x}}_t$ and $\tilde{\mathbf{u}}_t$, it would be straightforward to extend our method with more complex stochastic policies.

B. Guided Policy Search with Memory States

Memory states can in principle be combined with any policy search algorithm. In fact, prior work has proposed using discrete memory with storage actions [3]. However, for high-dimensional, continuous tasks, the logic required to choose which information to store and recall and when can become quite complex. This necessitates the use of powerful, expressive function approximators with hundreds or even thousands of parameters to represent $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$, which are generally very difficult to train with standard policy search techniques [7]. Guided policy search has previously been shown to be effective at learning these types of policies, and in this section we describe how guided policy search can be adapted to handle memory states.

Since the memory states and memory writing actions are simply appended to the observation and action vectors, the supervised learning procedure for the policy remains identical, and the policy is automatically trained to use the memory actions to mimic the pattern of memory activations optimized by the trajectory-centric "teacher" algorithms. The trajectory-centric teacher optimizes linear-Gaussian controllers $p(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ that control both the physical and memory states, essentially choosing the memory that the policy needs to have in order to take the right action. This happens automatically, because guided policy search adds a term to the cost function that penalizes deviation from the policy $\pi_\theta(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ in terms of KL-divergence. As described in Section III-B, this penalty takes the form of a KL-divergence and a linear Lagrange multiplier term.

We make a small modification to the trajectory-centric teacher algorithm to account for the particularly simple structure of the memory states. This modification also helps to make the algorithm scalable to larger memory state dimensionalities. Optimizing the linear-Gaussian controllers $p(\tilde{\mathbf{u}}_t|\tilde{\mathbf{x}}_t)$ requires estimating the dynamics $\hat{p}(\tilde{\mathbf{x}}_{t+1}|\tilde{\mathbf{x}}_t, \tilde{\mathbf{u}}_t) = \mathcal{N}(\tilde{f}_{\tilde{\mathbf{x}}t}\tilde{\mathbf{x}}_t + \tilde{f}_{\tilde{\mathbf{u}}t}\tilde{\mathbf{u}}_t + \tilde{f}_c, \mathbf{F}_t)$, which we do by using linear regression with a Gaussian mixture model prior, as described in previous work [9]. This approach is highly sample efficient, but we can make it even more efficient in the case of memory states by exploiting our knowledge of their dynamics. To that end, the dynamics are fitted according to

$$\tilde{f}_{\tilde{\mathbf{x}}t} = \begin{bmatrix} f_{\mathbf{x}t} & 0 \\ 0 & \mathbf{I} \end{bmatrix} \qquad \tilde{f}_{\tilde{\mathbf{u}}t} = \begin{bmatrix} f_{\mathbf{u}t} \\ \mathbf{I} \end{bmatrix} \qquad \tilde{f}_c = \begin{bmatrix} f_{ct} \\ 0 \end{bmatrix},$$

where $f_{\mathbf{x}t}$, $f_{\mathbf{u}t}$, and f_{ct} are estimates of the dynamics of the physical system, computed from the samples in the same way as in prior work [9]. Aside from this modification, the guided policy search algorithm we employ follows Algorithm 1.

V. Experimental Results

We evaluated our approach on a simple 2D navigation task, as well as two high-dimensional simulated tasks involving robotic manipulation, and compared it to alternative policy architectures and training methods. The aim of these experiments was to answer the following questions:

1) Can guided policy search with memory states solve complex tasks that require memory?
2) How do memory states compare with more standard RNN policies?
3) Does guided policy search make it easier to train policies with memory states, compared to alternative policy search methods?

The purpose of the 2D navigation task is to provide a platform for comparing the various methods and representations that is physically simple, but requires memory to succeed. The purpose of the more complex manipulation tasks is to evaluate the methods on a task that requires handling complex dynamics and high-dimensional state, and therefore requires a policy that both has memory and can learn complex control functions.

A. Representations and Methods

For the manipulation tasks, we used a neural network policy with two hidden layers of 40 rectified linear units (ReLU) of the form $z = \max(a, 0)$, while the navigation task used a single hidden layer with 10 units. The manipulation policies used 7-dimensional memory states, while the navigation policies used 4-dimensional memory states.

In addition to guided policy search with memory states, we evaluated three alternative policy representations, all trained with guided policy search, as well as an alternative optimization algorithm. The first of these representations was a feedforward neural network, without memory, but with the same number of units as the memory states policy. This type of policy was used with guided policy search in previous work [9], but it cannot learn tasks that require preserving information from previous time steps. Thus, it served as a baseline for demonstrating that memory is crucial for completing the tasks that we chose. The second

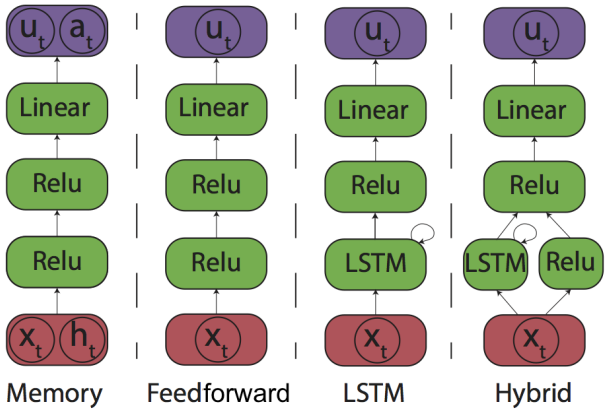

Fig. 2. Illustrations of the architecture used by our policy, as well as the alternative architectures.

representation was a recurrent network with LSTM units, which have previously been shown to achieve good results for long-term memorization tasks [14] and have recently become the architecture of choice for recurrent networks [22]. We chose this comparison to demonstrate the difference between memory states and a more standard RNN policy. The last representation was a hybrid network that consisted of both a feedforward branch and a recurrent LSTM branch at the first layer. We constructed this hybrid representation after observing that the standard LSTM policies often performed worse than the purely feedforward network, and in practice we noted that this architecture performed better than the feedforward and LSTM networks. Illustrations of each of the architectures are shown in Figure 2.

Besides guided policy search, we evaluated the memory states approach with the reward-weighted regression (RWR) algorithm [23], [24]. In previous work, we observed that this method performed well on tasks with high-dimensional policy representations [9], making it a good baseline method for our tasks. In general, we found in previous work that many prior methods had difficulty succeeding at the types of complex, high-dimensional tasks that we experimented with. We found that on all tasks, RWR achieved better results with a linear policy than with a neural network, so all reported RWR results use a linear parameterization.

B. Tasks

All of our tasks involve some partially observed component, where the observation $\mathbf{o}_t$ received by the policy does not contain all of the information necessary to accomplish the task, in contrast to the full state $\mathbf{x}_t$ that is provided to the linear-Gaussian controllers during training. The simple navigation task required the agent to travel to a designated position and, after the first half of the episode, return to the starting location. The starting location was varied, requiring a successful policy to remember its point of origin in order to return there in the second half of the episode. This was intentionally constructed to be sufficiently simple that the main challenge stemmed from the partial observability, rather than the physical difficulty of the behavior. The state space had two dimensions, and the two dimensional actions directly set the agent's velocity in the plane.

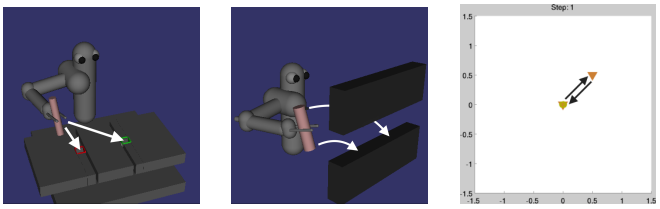

Fig. 3. Illustrations of the three simulated tasks. In peg sorting (left), the robot must insert the peg, shown in pink, into one of the two holes, denoted by the arrows, based on the specified condition. In the bottle and plate task (middle), the robot must insert the bottle into the horizontal cubby. The curved arrows denote the desired trajectory in this case, and do not represent different conditions. In different conditions for this task, the bottle changed into a plate, which is flat, wide, and horizontally oriented, and the cubby changed to be vertically oriented. In the 2D navigation task, the 2D pointmass, shown as the yellow dot, must travel out to the target location, shown as the orange triangle, and then return to the initial location. The arrows here denote the desired trajectory for this condition, and the different conditions correspond to different initial locations for the pointmass.

In the more complex manipulation tasks, the policies needed to control a 7 degree of freedom robot arm directly with joint torques in a full physics simulator. The controls $\mathbf{u}_t$ had 7 dimensions, and the configuration of the robot was provided in terms of joint angles and two 3D points on the object being manipulated, as well as their time derivatives, for a total dimensionality of 26. In the first manipulation task, shown in Figure 3, the robot was required to sort a peg into one of two holes. The hole position was provided to the policy on the first time step, and the policy was required to remember this position and move to the correct target. To prevent the policy from applying a large force in the direction of the target immediately when the target was presented, the robot was not allowed to physically move until the second time step. This task therefore could not be completed without memory, and provides a good comparison between our method and the alternative architectures.

In the second manipulation task, also shown in Figure 3, the robot was required to insert plates and bottles through a horizontal or vertical slot ("cubby"), and position them in the desired pose. The robot was required to determine both which object it is holding, and which way the cubby is oriented in order to angle the object correctly. This task is in fact possible to solve without memory, by using an appropriate reactive strategy that responds to collisions, but becomes significantly easier when memory is available.

C. Results

The results for each method on each of the tasks are presented in Figure 4. Each of the graphs shows the distance to the target for each task in terms of the number of samples used for training. For the manipulation tasks, the distance is measured between the object and its desired position. For the navigation task, we measure the minimum distance to the target in the first half of the episode and the minimum distance to the initial state in the second half, and the reported distance is the larger of the two. In this task, we defined a success to be achieving a reported distance of less than 0.5, which roughly corresponds to learning both to navigate

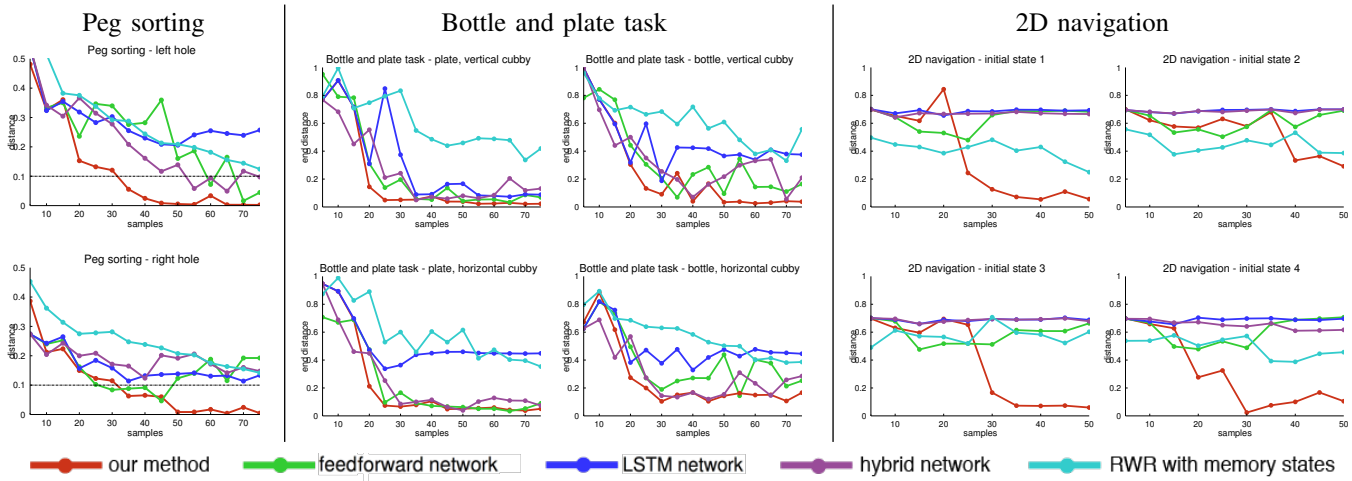

Fig. 4. Plots of the distance to the target in terms of the number of samples. For each method, the different lines show the distance for a different target position under the same policy. For the peg sorting and bottle and plate tasks, we plot the distance from the peg, while the retrieval task shows the distance between the retrieved object and the agent's starting position. In the sorting task, distances greater than the black dotted line correspond to failed trajectories. Note that a successful policy must succeed on *all* of the conditions for the task. Our policy with memory states is able to successfully solve each of the tasks, while the alternative architectures and methods fail on at least one condition for each of the tasks.

to the target and navigate back to the starting position. For each task, we show separate plots for each condition. For the peg sorting task, there are two conditions corresponding to the two targets, and the dotted line shows the depth of the hole. Policies with minimum distances above this line fail to insert the peg into the hole. For the cubby task, the conditions correspond to the orientation of the cubby and whether the robot is holding a plate or bottle. For the navigation task, the conditions correspond to different starting states. Each method was provided with 5 samples per target per iteration, which corresponds to 25 seconds of experience time. For example, our method took roughly 35 samples to succeed on both conditions of the peg sorting task, which corresponds to 175 seconds, or just under 3 minutes, of experience time.

Good performance on each task requires the policy to succeed for all of the conditions. For the peg sorting task, our method is able to insert the peg into the hole for both targets, while the feedforward policy simply picks the same target each time, succeeding on one condition but failing on the other. The standard LSTM also did not learn to remember the target, and instead found a "middle ground" strategy where it moved to the center rather than choosing a hole. Despite the fact that in theory this network could complete this task, in practice we found the LSTM network to be more difficult to train than our method, which required substantially less tuning. The hybrid network that consisted of feedforward and LSTM layers outperformed both the feedforward and pure LSTM policy, but still did not achieve the same performance as our memory states method.

On the bottle and plate task, the feedforward policy was able to succeed on three of the four conditions, but was unable to rotate the bottle to insert it into the horizontal cubby. Both the LSTM and hybrid policies were able to successfully insert the object into the cubby, but the resulting policies were substantially less stable, and were unable to position the object accurately at the desired position. This again reflects the difficulty of optimizing recurrent policies with backpropagation through time. In contrast, our policy with memory states was able to both insert the object into the cubby in each condition, and position it accurately at the target location. This task neatly illustrates one of the motivating factors for our method: even without explicit memory states, feedforward policies can adopt strategies that "offload" memory onto the physical state of the system, by utilizing subtly different joint angles and velocities depending on their past experience. However, with internal memory, this type of physical "offloading" is unnecessary.

For both manipulation tasks, RWR was unable to discover an effective policy, either with a neural network parameterization or with the linear parameterization shown in the plots, though the linear variant achieved slightly lower cost. This agrees with results reported in prior work [9], which showed that, for tasks of this type, guided policy search typically outperformed direct policy search methods, including RWR.

For the 2D navigation and retrieval task, our method was able to succeed from each of the starting positions. The feedforward network could not return the object back to the initial state due to lack of memory, while both the standard LSTM and hybrid policies could not be optimized successfully and did not produce a coherent behavior. Due to the substantially lower dimensionality of this task, RWR was in fact able to discover a policy that succeeded on one of the four conditions, but could not learn to effectively utilize the memory states to succeed from all four initial states.

We found that the LSTM policy was difficult to train for all of the tasks. We tested a variety of hyperparameters for this baseline and chose the best-performing policy to report. Testing many different combinations of learning parameters leads us to believe that it would be possible to successfully optimize an LSTM policy for these tasks, as some combinations led to better-performing policies. However, getting LSTMs to succeed generally requires a large search over the space of parameters, and our method was able to succeed with substantially less parameter tuning.

The project website contains supplementary videos that illustrate the behavior of these policies.[2]

VI. Discussion and Future Work

We presented a method for training policies for continuous control tasks that require memory. We augment the state space of the system with memory states, which the policy can choose to read from and write to as needed to accomplish the task. In order to make it tractable for the policy to learn effective memorization and recall strategies, we use guided policy search, which employs a simple trajectory-centric reinforcement learning algorithm to optimize over the memory state activations. This trajectory optimization procedure effectively tells the policy which information needs to be stored in the memory states, and the policy only needs to figure out how to reproduce the memory state activations. This tremendously simplifies the problem of searching over memorization strategies in comparison to model-free policy search methods and, unlike standard model-based methods for recurrent policies, it also avoids the need to backpropagate the gradient through time. However, when viewed together with the memory states, the policy is endowed with memory, and can be regarded as a recurrent neural network. Our experimental results show that our method can be used to learn policies for a variety of simulated robotic tasks that require maintaining internal memory to succeed.

Part of the motivation for our approach came from the observation that even fully feedforward neural network policies could often complete tricky tasks that seemed to require memory by using the physical state of the robot to "store" information, similarly to how a person might "remember" a number while counting by using their fingers. In our approach, we exploit this capability of reactive feedforward policies by providing extra state variables that do not have a physical analog, and exist only for the sake of memory.

While we presented experiments in simulation, guided policy search has been applied extensively on real robotic platforms [16], [10]. One of the reasons that guided policy search has had success in real-world robotic domains is because it can often succeed with many orders of magnitude fewer samples compared to other methods, and we retain this benefit in our work. Experiments on a real-world robotic platform would be valuable for evaluating the degree to which memory states can help general-purpose policies deal with partial observability stemming from real-world sensors, such as occlusions in camera images. We are beginning to experiment with these types of real-world tasks, and exploring this further is an exciting direction for future work.

VII. Acknowledgment

This research was funded in part by the Army Research Office through the MAST program, and by Darpa under Award # N66001-15-2-4047. C. F. was also funded in part by a SanDisk Fellowship.

[2] http://rll.berkeley.edu/gpsrnn/

References

[1] G. Shani, J. Pineau, and R. Kaplow, "A survey of point-based pomdp solvers," *Autonomous Agents and Multi-Agent Systems*, vol. 27, no. 1, pp. 1–51, 2013.

[2] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling, "Learning finite-state controllers for partially observable environments," in *Proceedings of the Fifteenth conference on Uncertainty in Artificial Intelligence (UAI)*, 1999, pp. 427–436.

[3] L. Peshkin, N. Meuleau, and L. Kaelbling, "Learning policies with external memory," in *Proceedings of the Sixteenth International Conference on Machine Learning (ICML)*, 2001, pp. 307–314.

[4] M. Toussaint, L. Charlin, and P. Poupart, "Hierarchical pomdp controller optimization by likelihood maximization." in *Proceedings of the 24th conference on Uncertainty in Artificial Intelligence (UAI)*, 2008, pp. 562–570.

[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)*, 2011, pp. 465–472.

[6] M. Watter, J. T. Springenberg, J. Boedecker, and M. Riedmiller, "Embed to control: A locally linear latent dynamics model for control from raw images," in *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[7] M. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Foundations and Trends in Robotics*, vol. 2, no. 1-2, pp. 1–142, 2013.

[8] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber, "Solving deep memory pomdps with recurrent policy gradients," in *Artificial Neural Networks–ICANN 2007*. Springer, 2007, pp. 697–706.

[9] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," 2014.

[10] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *arXiv preprint arXiv:1504.00702*, 2015.

[11] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa, "Online planning algorithms for pomdps," *Journal of Artificial Intelligence Research*, pp. 663–704, 2008.

[12] E. Brunskill, L. P. Kaelbling, T. Lozano-Perez, and N. Roy, "Continuous-state pomdps with hybrid dynamics." in *ISAIM*, 2008.

[13] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *arXiv preprint arXiv:1211.5063*, 2012.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[16] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," in *International Conference on Robotics and Automation (ICRA)*, 2015.

[17] S. Ross, G. Gordon, and A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," *Journal of Machine Learning Research*, vol. 15, pp. 627–635, 2011.

[18] S. Levine and V. Koltun, "Learning complex neural network policies with trajectory optimization," in *International Conference on Machine Learning (ICML)*, 2014.

[19] J. A. Bagnell and J. Schneider, "Covariant policy search," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[20] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

[21] J. Peters, K. Mülling, and Y. Altün, "Relative entropy policy search," in *AAAI Conference on Artificial Intelligence*, 2010.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 3104–3112.

[23] J. Peters and S. Schaal, "Applying the episodic natural actor-critic architecture to motor primitive learning," in *European Symposium on Artificial Neural Networks (ESANN)*, 2007.

[24] J. Kober and J. Peters, "Learning motor primitives for robotics," in *International Conference on Robotics and Automation (ICRA)*, 2009.