Learning Compound Multi-Step Controllers under Unknown Dynamics

Weiqiao Han Sergey Levine Pieter Abbeel

***Abstract*— Applications of reinforcement learning for robotic manipulation often assume an episodic setting. However, controllers trained with reinforcement learning are often situated in the context of a more complex compound task, where multiple controllers might be invoked in sequence to accomplish a higher-level goal. Furthermore, training such controllers typically requires resetting the environment between episodes, which is typically handled manually. We describe an approach for training chains of controllers with reinforcement learning. This requires taking into account the state distributions induced by preceding controllers in the chain, as well as automatically training reset controllers that can reset the task between episodes. The initial state of each controller is determined by the controller that precedes it, resulting in a non-stationary learning problem. We demonstrate that a recently developed method that optimizes linear-Gaussian controllers under learned local linear models can tackle this sort of non-stationary problem, and that training controllers concurrently with a corresponding reset controller only minimally increases training time. We also demonstrate this method on a complex tool use task that consists of seven stages and requires using a toy wrench to screw in a bolt. This compound task requires grasping and handling complex contact dynamics. After training, the controllers can execute the entire task quickly and efficiently. Finally, we show that this method can be combined with guided policy search to automatically train nonlinear neural network controllers for a grasping task with considerable variation in target position.**

I. Introduction

Reinforcement learning holds the promise of enabling robots to automatically learn large repertoires of motion skills for interacting with the world. Indeed, recent results have shown that it can be an effective tool for learning a range of behaviors, from games such as ball-in-cup and table tennis [1], to manipulation [2], [3] and robotic locomotion [4]–[7]. However, controllers trained with reinforcement learning are rarely situated in a vacuum. In complex, real-world tasks, they might precede and follow other controllers, and their initial state distributions are determined by the controllers that precede them. Furthermore, practical application of episodic reinforcement learning requires manually resetting the state of the system between episodes, which is often done manually or with hand-designed controllers.

In this work, we propose a framework that can train chains of controllers for performing compound tasks, together with a set of reset controllers that can reset the system to the initial state for each stage. When multiple controllers are trained together in this way, the resulting learning problem becomes non-stationary, since the initial state distribution of each controller is determined by the controllers that might precede it. While a number of methods could be designed to handle such non-stationarity, we show that a recently developed reinforcement learning algorithm based on fitting local linear models can handle tasks with non-stationary initial state distribution without any special modification [8]. This approach combines ideas from model-free and model-based reinforcement learning: the algorithm can handle tasks with unknown dynamics, but does not attempt to fit a single global dynamics model, which can be exceedingly difficult for complex manipulation tasks with contacts. Instead, this algorithm fits time-varying local linear dynamics around samples generated from the previous controller, and uses these linear dynamics to update the control policy using a variant of iterative LQR [9]. The algorithm can be used to train simple time-varying linear-Gaussian controllers, and can be combined with the framework of guided policy search to train nonlinear neural network controllers to tackle tasks that require generalization to new situations [8].

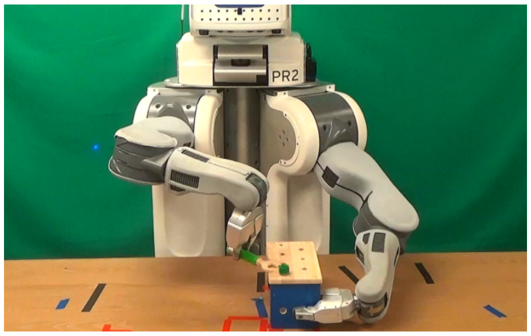


Fig. 1: PR2 robot performing the wrench task. This task consists of seven stages.

First, we experimentally demonstrate that this method can train reset controllers that automatically reset the system to the initial state following each episode, while at the same time training the forward controller that performs the task. Concurrently learning both forward and reset controllers does not appreciably increase overall training time, and makes it substantially easier to apply reinforcement learning to a wide range of robotic tasks without manual engineering.

Second, we demonstrate that this approach can be used to train a chain of controllers, with corresponding reset controllers, for tasks that consists of multiple steps. While learning the compound task, the robot executes each controller in sequence. The first controller that fails is trained further, along with its corresponding reset controller, allowing the entire compound task to be learned incrementally. We demonstrate this capability by training a sequence of

Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, CA, USA. {weiqiaohan,sergey.levine,pabbeel}@berkeley.edu

seven forward and reset controllers for a complex task that consists of picking up a wrench and using it to screw in a bolt, shown in Figure 1. We also show that our method can be used to automatically train a neural network grasping controller that can pick up a toy wrench from a variety of different positions, with concurrently learned controllers to reposition the wrench during training. This allows us to train a complex neural network policy that can generalize to new object positions with minimal human intervention.

II. Related Work

Policy search methods in robotics have been used to learn a variety of behaviors, ranging from games such as ball-in-cup [1], to manipulation [2], [3] and robotic locomotion [4]–[7]. Policy search methods based on standard reinforcement learning assumptions, including likelihood ratio methods and methods based on stochastic optimal control (SOC) [10], [11], usually assume that the task is stationary, which means that aspects of the environment such as the dynamics and the initial state distribution do not change during training. In this work, we empirically show that a recently developed reinforcement learning method for training linear-Gaussian controllers [12] does not require this assumption, which enables a number of interesting applications.

Our first application is to automatically train reset controllers. Most policy search methods in robotics, particularly for manipulation tasks, use an episodic finite-horizon formulation [13]. However, such methods assume that the environment is reset to the initial state by some existing approach, or simply by hand. By training the reset controller together with its corresponding forward controller, our method can make it much more straightforward to apply reinforcement learning to new tasks with minimal hand-engineering.

Our second application consists of learning compound tasks that require chaining multiple controllers together, where the initial state of the next controller depends on the terminal state of the previous one. Several prior methods have proposed chaining controllers based on dynamic movement primitive (DMP) representations [14]–[16], which lend themselves to efficient learning algorithms, but are primarily kinematic. The linear-Gaussian controllers trained by our approach explicitly encode a distribution over actions (torques) in terms of the robot's state at each time step, allowing them to carry out fast, fluent motions with direct torque control. Previous work has also sought to develop ways of choosing which controller to invoke based on sensory input [17] and constructing subgoals for each stage from demonstrations [18] and system interaction [19]. Our work is concerned primarily with training the controllers in the sequence, and is therefore complementary to previous work that deals with automatically acquiring subgoals and choosing which controller to invoke. In fact, a complete and generally applicable learning system might combine our method with automatic subgoal segmentation, in order to train complex behaviors with minimal manual intervention.

While time-varying linear-Gaussian controllers can represent a wide range of different behaviors, they are essentially trajectory-centric and limited in their ability to generalize to new situations. To address this limitation, previous work has proposed to use guided policy search to train a nonlinear neural network policy from multiple linear-Gaussian controllers trained for different initial states [8]. This nonlinear policy can then succeed from the initial states of each linear-Gaussian controller, and can generalize to new states. We illustrate that our approach makes it even easier to train such neural network controllers, by automatically training reset controllers that can reset the environment during training.

III. Learning Controllers under Unknown Dynamics with Trajectory Optimization

In this section, we review a recent method for training linear-Gaussian controllers for robotic manipulation with locally linear models, and briefly summarize how this approach can be extended using guided policy search to train complex nonlinear policies, such as large neural networks. Our implementation follows prior work [8], [12], [20]. As we will discuss in Sections IV and V, this approach can extend to the case of non-stationary initial state distributions, which makes it possible to apply it to train reset controllers and compound motion skills that require chaining together multiple motion primitives.

A. Training Linear-Gaussian Controllers

The aim of the algorithm in this section is to control a dynamical system with states $\mathbf{x}_t$ and actions $\mathbf{u}_t$ in order to minimize the total expected cost over a finite-length episode, given by $E_p[\sum_{t=1}^{T} \ell(\mathbf{x}_t, \mathbf{u}_t)]$, which we abbreviate as $E_p[\ell(\tau)]$ for convenience, using $\tau = \{\mathbf{x}_1, \mathbf{u}_1, \ldots, \mathbf{x}_T, \mathbf{u}_T\}$ to denote a trajectory. In order to make this learning problem tractable, the class of controllers is first constrained to time-varying linear-Gaussian controllers of the form $p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\hat{\mathbf{u}}_t + \mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t), \mathbf{C}_t)$. Such controllers can be viewed as stabilizing the system around some nominal trajectory $\hat{\tau} = \{\hat{\mathbf{x}}_1, \hat{\mathbf{u}}_1, \hat{\mathbf{x}}_2, \hat{\mathbf{u}}_2, \ldots, \hat{\mathbf{x}}_T, \hat{\mathbf{u}}_T\}$, using the feedbacks $\mathbf{K}_t$ for stabilization. We therefore call them trajectory-centric controllers. These types of controllers allow for a very efficient learning algorithm that combines ideas from model-free and model-based reinforcement learning. This method does not require a model of the system dynamics to be known in advance, and does not attempt to fit a global model of the dynamics, as is standard in model-based reinforcement learning. Instead, a new time-varying linear-Gaussian model of the form $p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(f_{\mathbf{x}t}\mathbf{x}_t + f_{\mathbf{u}t}\mathbf{u}_t, \mathbf{F}_t)$ is estimated at each iteration around the samples drawn from the controller at the previous iteration. Because this model is time-varying, it provides considerable flexibility, and because it is local, it does not need to capture the potentially complex global dynamics of the task. Once the dynamics are estimated, a time-varying linear-Gaussian controller can be optimized under the estimated dynamics using a variant of linear-quadratic-Gaussian (LQG) algorithm, which we review below.

In the LQG setting, we use a quadratic expansion of the cost function $\ell(\mathbf{x}_t, \mathbf{u}_t)$ and a linearization of the dynamics to locally improve a trajectory. The linearization is typically obtained from a known model of the dynamics, but since the dynamics in our tasks are not known, we use the fitted dynamics, defined by $f_{\mathbf{x}t}$ and $f_{\mathbf{u}t}$. We will use $f_{\mathbf{xu}t} = [f_{\mathbf{x}t}\ f_{\mathbf{u}t}]$ to denote the concatenation of these matrices, and $\ell_{\mathbf{xu}t}$ and $\ell_{\mathbf{xu},\mathbf{xu}t}$ to denote the first and second derivatives of the cost around the current nominal trajectory $\hat{\tau} = \{\hat{\mathbf{x}}_1, \hat{\mathbf{u}}_1, \hat{\mathbf{x}}_2, \hat{\mathbf{u}}_2, \ldots, \hat{\mathbf{x}}_T, \hat{\mathbf{u}}_T\}$. In this setting, the Q-function and value function are quadratic, and are given by

$$V(\mathbf{x}_t) = \frac{1}{2}\mathbf{x}_t^{\mathrm{T}} V_{\mathbf{x},\mathbf{x}t}\mathbf{x}_t + \mathbf{x}_t^{\mathrm{T}} V_{\mathbf{x}t} + \text{const}$$

$$Q(\mathbf{x}_t, \mathbf{u}_t) = \frac{1}{2}[\mathbf{x}_t\, \mathbf{u}_t] Q_{\mathbf{xu},\mathbf{xu}t} [\mathbf{x}_t\, \mathbf{u}_t]^{\mathrm{T}} + [\mathbf{x}_t\, \mathbf{u}_t] Q_{\mathbf{xu}t} + \text{const}$$

These functions can be computed by means of the following recurrence:

$$\begin{aligned} Q_{\mathbf{xu},\mathbf{xu}t} &= \ell_{\mathbf{xu},\mathbf{xu}t} + f_{\mathbf{xu}t}^{\mathrm{T}} V_{\mathbf{x},\mathbf{x}t+1} f_{\mathbf{xu}t} \\ Q_{\mathbf{xu}t} &= \ell_{\mathbf{xu}t} + f_{\mathbf{xu}t}^{\mathrm{T}} V_{\mathbf{x}t+1} \\ V_{\mathbf{x},\mathbf{x}t} &= Q_{\mathbf{x},\mathbf{x}t} - Q_{\mathbf{u},\mathbf{x}t}^{\mathrm{T}} Q_{\mathbf{u},\mathbf{u}t}^{-1} Q_{\mathbf{u},\mathbf{x}t} \\ V_{\mathbf{x}t} &= Q_{\mathbf{x}t} - Q_{\mathbf{u},\mathbf{x}t}^{\mathrm{T}} Q_{\mathbf{u},\mathbf{u}t}^{-1} Q_{\mathbf{u}t}, \end{aligned}$$

and the new deterministic optimal controller is given by $g(\mathbf{x}_t) = \hat{\mathbf{u}}_t + \mathbf{k}_t + \mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t)$, where $\mathbf{K}_t = -Q_{\mathbf{u},\mathbf{u}t}^{-1} Q_{\mathbf{u},\mathbf{x}t}$ and $\mathbf{k}_t = -Q_{\mathbf{u},\mathbf{u}t}^{-1} Q_{\mathbf{u}t}$. A derivation of this control law can be found in prior work [9].

However, as discussed in prior work [8], simply solving for the optimal controller under the currently estimated dynamics and running this controller on the robot generally does not produce good results, since the estimated linear-Gaussian dynamics are only accurate in the vicinity of the trajectory distribution induced by the previous controller. In practice, such an algorithm rarely makes good progress. Instead, we need to keep the new controller close to the old one, such that the resulting distribution over trajectories $p(\tau)$ does not change too drastically. A natural choice for bounding the amount of change in this distribution is the KL-divergence $D_{\mathrm{KL}}(p(\tau)\|\hat{p}(\tau))$ between the new distribution $p(\tau)$ and the old one $\hat{p}(\tau)$, resulting in the following constrained optimization problem:

$$\min_{p(\tau)\in\mathcal{N}(\tau)} E_p[\ell(\tau)] \text{ s.t. } D_{\mathrm{KL}}(p(\tau)\|\hat{p}(\tau)) \leq \epsilon, \qquad (1)$$

where the trajectory distributions are formed by multiplying the linear-Gaussian dynamics and the action distributions: $p(\tau) = p(\mathbf{x}_1)\prod_{t=1}^T p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)p(\mathbf{u}_t|\mathbf{x}_t)$. Since the dynamics of the new and old trajectory distributions are assumed to be the same, the KL-divergence is given by

$$D_{\mathrm{KL}}(p(\tau)\|\hat{p}(\tau)) = \sum_{t=1}^T E_{p(\mathbf{x}_t,\mathbf{u}_t)}[\log \hat{p}(\mathbf{u}_t|\mathbf{x}_t)] - \mathcal{H}(p),$$

and the Lagrangian of the constrained problem in Equation (1) is given by

$$\mathcal{L}_{\mathrm{traj}}(p,\eta) = E_p[\ell(\tau)] + \eta[D_{\mathrm{KL}}(p(\tau)\|\hat{p}(\tau)) - \epsilon] = \left[\sum_t E_{p(\mathbf{x}_t,\mathbf{u}_t)}[\ell(\mathbf{x}_t,\mathbf{u}_t) - \eta\log\hat{p}(\mathbf{u}_t|\mathbf{x}_t)]\right] - \eta\mathcal{H}(p(\tau)) - \eta\epsilon.$$

The constrained problem in Equation (1) can then be optimized with dual gradient descent [21], where we alternate between minimizing the Lagrangian with respect to the primal variables, which are the parameters of p, and taking a subgradient step on the Lagrange multiplier η. The optimization with respect to p can be performed very efficiently using a variant of the LQG algorithm, by observing that the Lagrangian is simply the expectation of a quantity that does not depend on p and an entropy term. As shown in prior work [22], the optimal linear-Gaussian controller for an objective that requires minimizing cost and maximizing the entropy of the controller is simply given by $p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\hat{\mathbf{u}}_t + \mathbf{k}_t + \mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t), Q_{\mathbf{u},\mathbf{u}t}^{-1})$. It therefore only remains to transform the Langragian into an objective of the form $\sum_{t=1}^T E_{p(\mathbf{x}_t,\mathbf{u}_t)}[\tilde{\ell}(\mathbf{x}_t,\mathbf{u}_t)] - \mathcal{H}(p)$, and then we can use the standard LQG algorithm to obtain the solution under the modified cost $\tilde{\ell}(\mathbf{x}_t,\mathbf{u}_t)$. We can derive this modified cost by dividing the Lagrangian by η, which does not alter the optimum with respect to the primal variables. We then obtain $\tilde{\ell}(\mathbf{x}_t,\mathbf{u}_t) = \frac{1}{\eta}\ell(\mathbf{x}_t,\mathbf{u}_t) - \log\hat{p}(\mathbf{u}_t|\mathbf{x}_t)$.

In summary, each iteration of this algorithm consists of generating samples by running the controller $\hat{p}(\mathbf{u}_t|\mathbf{x}_t)$, fitting time-varying linear-Gaussian dynamics to these samples, and solving the constrained optimization in Equation (1) under these dynamics. This constrained optimization in turn is performed with dual gradient descent, which alternates between solving the LQG problem with a modified cost $\tilde{\ell}(\mathbf{x}_t,\mathbf{u}_t) = \frac{1}{\eta}\ell(\mathbf{x}_t,\mathbf{u}_t) - \log\hat{p}(\mathbf{u}_t|\mathbf{x}_t)$ and incrementing the Lagrange multiplier η by its subgradient $D_{\mathrm{KL}}(p(\tau)\|\hat{p}(\tau)) - \epsilon$.

Our implementation also includes a number of improvements detailed in previous work [8], including the use of a Gaussian mixture model (GMM) prior during the dynamics fitting and an adaptive step size adjustment rule to control ϵ. The GMM prior serves to reduce the sample complexity of the linear regression procedure for fitting the dynamics, by exploiting the fact that the dynamics have strong temporal correlations. A single GMM is fitted to all samples $(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})$, and the clusters that best explain the samples at a particular time step act as the prior for fitting the dynamics at that step. This allows us to use many fewer samples per iteration than there are dimensions in the system. The step size ϵ is adjusted based on the degree to which the cost predicted under the old dynamics agrees with the estimated cost under the new dynamics, under the assumption that the step size should be reduced when this disagreement is high. Both the GMM prior and the step size adjustment method are described in detail in prior work [8].

B. Training Nonlinear Policies

While the linear-Gaussian controllers described in the preceding section can execute simple tasks with modest amounts of variation, many tasks require policies that can handle a variety of initial states. For example, a policy for picking up an object might be required to move the arm into a range of different positions for a successful grasp, depending on the object's pose. To handle such variation, we can train multiple linear-Gaussian controllers for different

Algorithm 1 Guided policy search with BADMM [20]

```
1: for iteration $k = 1$ to $K$ do
2:     Generate samples $\{\tau_i^j\}$ from each linear-Gaussian
       controller $p_i(\mathbf{u}_t|\mathbf{x}_t)$ by running it on the robot
3:     Train nonlinear neural network policy $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ to
       match the sampled trajectories $\{\tau_i^j\}$
4:     Estimate dynamics $p_i(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ for each linear-
       Gaussian controller $p_i(\mathbf{u}_t|\mathbf{x}_t)$
5:     Modify cost $\bar{\ell}(\mathbf{x}_t, \mathbf{u}_t) = \ell(\mathbf{u}_t, \mathbf{x}_t) + \phi(\mathbf{x}_t, \mathbf{u}_t, \theta)$ to
       penalize deviation from $\pi_\theta$
6:     Update $p_i(\mathbf{u}_t|\mathbf{x}_t)$ using the LQG-like method with
       augmented cost $\bar{\ell}(\mathbf{x}_t, \mathbf{u}_t)$
7: end for
8: return optimized policy parameters $\theta$
```

instances of the task, such as different target object positions, and combine them into a single nonlinear neural network policy that can generalize to new states. This is done using the framework of guided policy search [8], [20]. In particular, we use the BADMM-based guided policy search algorithm discussed by Levine et al. [20].

A summary of this method is provided in Algorithm 1, although a full derivation is outside the scope of the paper. The algorithm has three principle differences from the linear-Gaussian training procedure in the previous section. First, it requires samples from multiple linear-Gaussian controllers at each iteration. Second, the samples are used to not only estimate the dynamics of each linear-Gaussian controller, but are also used as training data to train the nonlinear neural network policy with supervised learning. Third, the cost function for the linear Gaussian controllers is modified to include a term that penalizes deviation from the behavior of the latest neural network policy, which causes the linear-Gaussian controllers to converge to the same behavior as the neural network policy, ensuring that the network exhibits good long-horizon performance. For details on each of these components, we refer the reader to previous work [20].

Prior applications of guided policy search considered each linear-Gaussian controller in isolation, usually trained for a different instance of the task (e.g. different positions of a target object for manipulation). In this work, we instead chain these controllers into a continuous execution stream, and combine a subset of the controllers into a neural network that can generalize effectively. For example, in Section VI-C, we train a neural network for grasping in multiple positions, together with reset controllers that retract the arm. When training compound controllers, we may not be able to collect new samples from each controller at each iteration until the backward controllers have learned to reset successfully. In this case, we only update the neural network once samples from all controllers are available. Aside from this, our implementation of guided policy search follows prior work.

IV. Reset Controllers

In the previous section, we reviewed an algorithm for optimizing controllers for episodic tasks. These controllers are trained using samples generated by running the current controller on the robot. Each such episode requires resetting the robot into the task's initial state. For some behaviors, this reset is trivial, and simply requires moving the arm into a particular configuration, which can be accomplished with simple PD control. However, for many manipulation tasks, resetting the state of the system can be more complex, and additional manual engineering is often required to perform the reset automatically. In order to make the learning algorithm in the preceding section general and easy to apply to new tasks, we can learn a separate controller to perform the reset procedure. However, the behavior of this reset controller depends closely on the corresponding forward controller, since the terminal state distribution of the forward controller serves as the initial state distribution for the reset controller, and vice versa.

Algorithm 2 Training with reset controllers

```
1: for iteration $k = 1$ to $K$ do
2:     for samples $i = 1$ to $N$ do
3:         Run $p_f(\mathbf{u}_t|\mathbf{x}_t)$ on the robot to generate sample $\tau_f^i$
4:         Run $p_r(\mathbf{u}_t|\mathbf{x}_t)$ on the robot to generate sample $\tau_r^i$
5:     end for
6:     Use $\{\tau_f^i\}$ to update $p_f$ with cost $\ell(\mathbf{x}_t, \mathbf{u}_t)$
7:     Use $\{\tau_r^i\}$ to update $p_r$ with reset cost $\ell_r(\mathbf{x}_t, \mathbf{u}_t)$
8: end for
9: return forward controller $p_f$ and reset controller $p_r$
```

Learning the forward and reset controllers together constitutes a non-stationary problem, since the initial state distribution of each of the controllers changes during learning. As we experimentally demonstrate in Section VI, the algorithm in the preceding section can learn such reset controllers with minimal loss in performance compared to a forward controller trained with manually designed reset behavior. This is because it does not assume that the problem is stationary. Instead, a new controller is optimized at each iteration, using an efficient LQG-based algorithm. The only modification that is required compared to the standard approach is to estimate a new Gaussian initial state distribution in order to accurately evaluate the expected cost, which is done simply by fitting a Gaussian to the initial states in the samples.

The resulting algorithm has a simple structure presented in Algorithm 2. For each sample from the forward controller p_f, a corresponding sample is generated from the reset controller p_r. If the reset controller is successful, this also has the desirable property of placing the robot into the correct initial state. However, even if success is not attained, the forward controller is executed from whichever state is actually reached by the reset controller. Both controllers are updated (separately) once the required number of samples has been gathered. The forward controller is updated to improve its performance with respect to the task cost $\ell(\mathbf{x}_t, \mathbf{u}_t)$, while the reset controller is updated with respect to a reset cost $\ell_r(\mathbf{x}_t, \mathbf{u}_t)$, which consists of a quadratic penalty for the distance between the state $\mathbf{x}_t$ and the desired reset state, as well as a quadratic actuation penalty on $\mathbf{u}_t$.

The reset controller makes it straightforward to train linear-Gaussian controllers for a variety of tasks without manually designing a reset procedure. However, we can take the idea of training multiple controllers further and train groups of forward and reset controllers for complex tasks with multiple steps, as described in the following section.

V. Compound Controllers

Linear-Gaussian controllers are well suited for representing individual motion primitives [8], [23]. However, many real-world tasks consist of multiple steps, which must be executed sequentially. Learning a single linear-Gaussian controller for such a task would be difficult and highly prone to local optima. However, if we are provided with a set of intermediate subgoals, we can optimize separate linear-Gaussian controllers for reaching each subgoal, and even combine some of them into a nonlinear neural network policy that can generalize to new situations. The challenge with this approach is that the behavior of the preceding controller will influence the initial state of the next one, again resulting in a non-stationary learning problem.

We will denote the forward controllers for each stage in such a compound task as $p_{f_1}, p_{f_2}, \dots, p_{f_M}$. Each forward controller is associated with a corresponding reset controller $p_{r_1}, p_{r_2}, \dots, p_{r_M}$, which aims to reach the target of the preceding forward controller or, in the case of p_{r_1}, the initial state. The cost for each forward controller is assumed to be provided. Often this cost will require the robot to position its end-effector (or an object held in the end-effector) at a specific position. This does not fully determine the configuration of the robot, and the target is not always reached, so the task remains non-stationary. Some of the controllers might also be used to train a neural network policy π_{θ_j}, as discussed later in this section.

The full set of forward and reset controllers is trained incrementally, as shown in Algorithm 3. We generate samples using a simple schedule: the forward controllers are executed in sequence until one of them fails, at which point the corresponding reset controller is executed, and the forward controller attempts to run again. When all forward controllers succeed, the process is reversed and the reset controllers are executed in sequence. Success and failure are defined manually. In our experiments, success always depends on the position of the end-effector, which must be within a threshold distance of the target. Reset controllers are assumed to always succeed during the reset pass, since we found that they were usually successful by the time the corresponding forward controller learned the task. However, it would also be straightforward to add a success test for the reset controller based on a distance threshold, and invoke the corresponding forward controller in the case of failure.

The pseudocode in Algorithm 3 illustrates this procedure with a loop that repeats until each controller has been trained for at least K iterations, where each iteration requires N samples. For each invocation of the loop, we execute either the current forward controller, or the current backward controller, depending on whether we are going forward or backward. If the new sample raises the sample count for this controller to N, we take one training iteration. In either case, if the sample succeeds, we move on to the next controller in the sequence. Otherwise we execute the backward controller so that the forward controller can try this step again.

Algorithm 3 Training compound controllers

```
 1: $K$ is the desired number of iterations per controller
 2: $N$ is the desired number of samples per iteration
 3: Set: $k_j = 0$, $i_{fj} = 0$, $i_{rj} = 0$, $s = 1$, forward = true
 4: **while** $\min_j k_j < K$ **do**
 5:     **if** forward = true **then**
 6:         Run $p_{f_s}(\mathbf{u}_t|\mathbf{x}_t)$ to get sample $\tau_{f_s}^{i_{fs}}$, increment $i_{fs}$
 7:         **if** $i_{fs} = N$ **then**
 8:             Update $p_{f_s}$ using $\{\tau_{f_s}^{i_{fs}}\}$ with cost $\ell_s(\mathbf{x}_t, \mathbf{u}_t)$
 9:             **if** $p_{f_s}$ is used for neural network $\pi_{\theta_j}$ and samples
                from all controllers for $\pi_{\theta_j}$ available **then**
10:                 Update $\pi_{\theta_j}$ with latest samples (Alg 1 line 3)
11:                 Update $\ell_s(\mathbf{x}_t, \mathbf{u}_t)$ policy term (Alg 1 line 5)
12:             **end if**
13:             Reset $i_{fs}$ to 0, increment iteration $k_s$ by one
14:         **end if**
15:         **if** the last sample succeeded **then**
16:             **if** $s = M$, set forward $\leftarrow$ false **else** increment $s$
17:         **else**
18:             Run $p_{r_s}(\mathbf{u}_t|\mathbf{x}_t)$ to get sample $\tau_{r_s}^{i_{rs}}$, increment $i_{rs}$
19:             **if** $i_{rs} = N$ **then**
20:                 Update $p_{r_s}$ using $\{\tau_{r_s}^{i_{rs}}\}$ with cost $\ell_{rs}(\mathbf{x}_t, \mathbf{u}_t)$
21:                 Reset $i_{rs}$ to 0
22:             **end if**
23:         **end if**
24:     **else**
25:         Run $p_{r_s}(\mathbf{u}_t|\mathbf{x}_t)$ to get sample $\tau_{r_s}^{i_{rs}}$, increment $i_{rs}$
26:         **if** $i_{rs} = N$ **then**
27:             Update $p_{r_s}$ using $\{\tau_{r_s}^{i_{rs}}\}$ with cost $\ell_{rs}(\mathbf{x}_t, \mathbf{u}_t)$
28:             Reset $i_{rs}$ to 0
29:         **end if**
30:         **if** $s = 1$, set forward $\leftarrow$ true **else** decrement $s$
31:     **end if**
32: **end while**
33: **return** Forward controllers $p_{fj}$, reset controllers $p_{rj}$,
    neural network policies $\pi_{\theta_j}$
```

If a particular forward controller is also included in the training set for a neural network policy π_{θ_j}, we update this neural network policy at the same time as its corresponding controllers, using the latest samples from each controller that contributes to the neural network. After this, the cost of each of these controllers is updated to include the policy deviation term described on line 5 of Algorithm 1 for the latest policy π_{θ_j}. In the next section, we describe an example task where such a neural network policy is trained in the context of other controllers to provide robustness to variation in the positions of objects in a manipulation task.

VI. Experimental Results

In this section, we discuss the experiments we performed to evaluate our method. We first aim to determine whether learning the reset controller simultaneously with the corresponding forward controller decreases performance or increases the required number of samples beyond what would be needed with a manually specified reset procedure. We then proceed to train a chain of controllers for a complex seven-stage task, in order to determine whether such elaborate behaviors can effectively be tackled with a sequence of linear-Gaussian controllers that are trained incrementally. Finally, we train a nonlinear neural network policy for a grasping task by using five controllers trained to grasp at different positions, along with three other supporting controllers to automate the learning process.

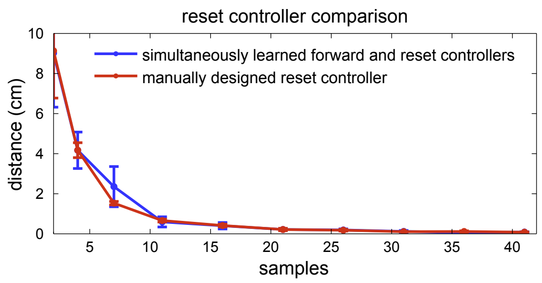
All experiments were conducted on the PR2 robot shown in Figure 1. All controllers were intialized by using LQR to stabilize around the initial state of the task, using an initial double-integrator guess of the dynamics, with a substantial amount of Gaussian noise added for exploration. All controllers were therefore initially identical, starting out with random motions that were roughly centered on the initial pose. Note that this initial pose was the same for all controllers in the compound controller task. All of the tasks and subtasks required placing the end-effector in a particular pose and minimizing torque, with the end-effector pose fully defined by the position of three points in the frame of the wrist. We therefore follow previous work [8] and use a cost function of the following form:

$$\ell(\mathbf{x}_t, \mathbf{u}_t) = wd(\mathbf{x}_t)^2 + v\log(d(\mathbf{x}_t)^2 + \alpha) + w_{\mathbf{u}}\|\mathbf{u}_t\|_G^2,$$

where w, v, and $w_{\mathbf{u}}$ are constant weights, G is a diagonal matrix of gains that penalized torques more strongly at smaller joints such as the wrist, compared to larger joints such as the shoulder, and $d(\mathbf{x}_t)$ gives the average distance between the three points in the frame of the wrist and their target positions. The second $\log$ term encourages the controller to match the target position with high precision. More details about the motivation for this cost function are provided in previous work [8]. At each iteration of our LQR-based learning algorithm, we used between 3 and 5 sample trajectories, with an adaptive sample size adjustment rule described in prior work used to adaptively increase or decrease the sample count based on the quality of the dynamics estimate [8]. Video recordings of each of the experiments in this section can be found on the project website: `http://rll.berkeley.edu/reset_controller/`.

A. Reset Controller Experiment

In order to evaluate our algorithm with automatically learned reset controllers, we set up a Lego block task that required the robot to stack a large Lego block onto another block at a specific location. Previous work showed that this task can be performed successfully with a manually designed reset controller [8]. The results in Figure 2 show that learning the reset controller simultaneously with the forward controller does not noticeably affect the learning



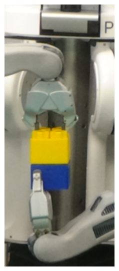


Fig. 2: Learning curves for the Lego block task with a manually designed reset controller, as well as with an automatically learned reset controller (left), along with an illustration of the Lego block stacking task (right). Learning the reset controller together with the forward controller does not adversely affect the rate at which the task is learned.

speed for this task. However, as shown in the supplementary video, the behavior of the learning process appears very different: since the initial reset controller is simply random, the robot begins by continuously moving its arm through random motions, until a distinct forward and reset phase emerge. The final learned reset controller is fast and torque-efficient, pulling the lego block off of the block beneath it and quickly setting the arm into the initial state.

B. Compound Controller Experiment

To evaluate our method on a compound multi-step task, we trained a sequence of forward and reset controllers for a task that requires using a toy wrench to screw in a bolt. This task consists of 7 steps, which include grasping the wrench and making one full turn. Illustrations of each step are shown in Figure 3. During training, the entire sequence of forward controllers was able to succeed after 98 iterations (with 5 episode per iteration), and was able to succeed consistently after 147 iterations. A more detailed breakdown of the number iterations for the forward and backward controllers at each stage is provided in Table I. Note that we refer to each execution of a controller (forward or reset) as an "episode." Unlike in the standard reinforcement learning setting, one episode is not one execution of the task: since the task consists of 7 steps, it requires 7 episodes to execute the task end-to-end, in addition to the reset controllers. Since the training is incremental, we do not need to execute the entire sequence of controllers prior to each learning iteration.

	stage	(a)	(b)	(c)	(d)	(e)	(f)	(g)
until first success	forward	10	5	6	6	9	6	7
	backward	10	5	6	6	9	6	7
until consistently successful	forward	16	13	9	10	13	9	9
	backward	16	13	7	8	11	6	7

TABLE I: Number of iterations required for each stage in the wrench task. Each iteration required 5 episodes.

After the full sequence of controllers is trained, the robot can screw in the bolt for any number of turns by repeating

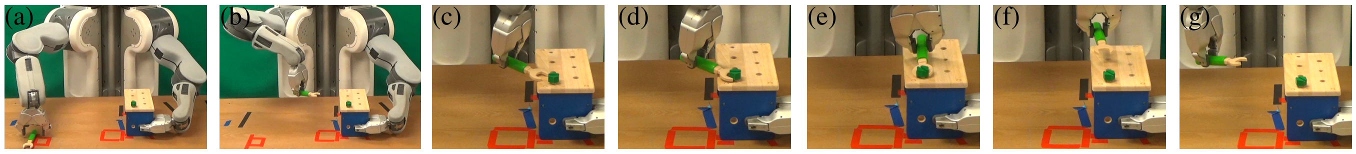


Fig. 3: Illustration of the compound wrench task. The wrench is grasped (a), then placed in a rest position (b), then positioned near the bolt (c) and then on the bolt (d), and then turned to rotate the bolt about 60 degrees (e). The wrench is then withdrawn (f) and placed back in the rest position (g), at which point it is ready for another turn. By repeating steps (c) through (f), the robot can turn the bolt repeatedly, screwing it into the bench. The additional approach step (c) was added to ensure that the wrench is positioned carefully, as it was prone to rotate in the gripper if it struck the bench or the bolt too quickly.

steps (c) through (g) in a loop, and can execute the reset controllers to put the wrench back on the table. Videos of this task are included on the project website.[1].

The trained controller sequence was tested on 10 trials, each of which required grasping the wrench, tightening the bolt for six turns, and placing the wrench back on the table. The controllers were completely successful on 8 of the 10 trials, indicating a high degree of reliability. On one trial, the bolt was turned three times, but the fourth turn failed when the wrench struck the bolt and rotated in the robot's gripper. One trial failed during the first turn, again because the wrench rotated in the robot's gripper. Since the method does not currently measure the orientation of the wrench after it has been grasped, the controllers had no way to detect this failure. However, if the problem was detected, it could have been corrected by running the reset controllers to place the wrench back on the table and then picking it back up again. Detecting such failures is a promising direction for future work, and would improve the robustness of our method.

C. Compound Controller with Neural Network Policy

In the last experiment, we evaluated the ability of our method to automate the training of nonlinear neural network policies that can generalize over variation in the task. The goal was to grasp a toy wrench from a variety of different positions. The grasping was performed by executing two forward controllers: first a single forward controller moved the arm over the table, and then one of five second-stage forward controllers was executed to move the gripper onto the wrench. Each of these five second-stage controllers was trained to place the gripper at a different position. The wrench was localized by using a Kinect-based object detector, and the controller trained for the nearest position was chosen for the second stage. The position of the target was included in the state of the controller, so each of the five controllers could tolerate small deviations in the position of the wrench. However, no single controller could grasp the wrench at all positions. The five second-stage forward controllers shared a single reset controller. Together with the reset for the first stage, the task consisted of eight controllers (six forward and two reset). The five grasping controllers were used to train a single neural network policy with two hidden layers of 40 units each, using soft rectifying nonlinearities as described in previous work [8]. After training, this policy was able to grasp the wrench at any position in the training region, and could also extrapolate to a substantially larger region around the training area.

[1]See `http://rll.berkeley.edu/reset_controller/`

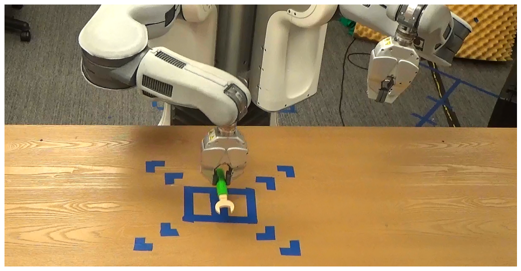


Fig. 4: Illustration of the testing setup for the wrench grasping task. The inner blue rectangle indicates the region in which the neural network policy was trained, and the other rectangles are larger by 5 cm and 10 cm on each side, respectively. Note that not all points in the outermost ring are within the workspace of the robot.

Figure 4 shows the test setup for the trained neural network policy. The inner rectangle indicates the region in which the network was trained, the middle rectangle is larger by about 5 cm on each side, and the largest one is larger by about 10 cm on each side. Table II shows the success rate of the neural network for grasping the wrench at various distances. Although the network was trained in a relatively small area, it was able to generalize to a substantially larger region.

distance	training region	+5 cm	+10 cm
success rate: first attempt	20/20	20/20	16/20
success rate: five attempts	20/20	20/20	17/20

TABLE II: Success rate of the neural network grasping policy at various distances from the center of the training region. The first row shows the success rate on the first attempt, while the second shows the success rate when the robot was allowed to attempt to grasp five times for each test position, using the reset controllers to reset in the case of failure.

This experiment illustrates one of the advantages of reset controllers for more complex tasks. During training, the combination of forward and reset controllers was able to practice grasping the wrench, withdraw the arm, and place the wrench back on the table at new positions, allowing the set of forward and reset controllers to train the neural network completely autonomously. The resulting neural network policy was then able to succeed at the task even when the position of the wrench was altered substantially, indicating robustness to task variability.

VII. Discussion and Future Work

In this paper, we proposed an approach for training chains of controllers for performing compound tasks with per-stage subgoals, together with reset controllers that can reset the system to the initial state of each controller during training. Learning is accomplished using a recently proposed algorithm for training linear-Gaussian controllers with local time-varying linear dynamics, which we show can handle non-stationary initial state distributions. By training individual controllers in the context of a compound task, each controller can adapt to the controllers around it. Furthermore, automatic training of reset controllers makes it possible to easily deploy this method for a variety of tasks without manually crafting a reset procedure for each one, increasing the applicability of reinforcement learning to robotic control problems. In the case of compound tasks, the ability to also train reset controllers for each stage in the task makes the process largely automatic, and provides a repertoire of controllers that can be recombined at test time in a different order, and even combined into a single nonlinear neural network policy that can generalize to variation in the task.

Although our experimental results demonstrate that our method can handle a number of challenging robotic manipulation tasks, it has a number of limitations. First, the reset controllers can only be trained for systems that are practical to reset automatically using the state-space provided to the algorithm. For example, the state-space for the wrench and bolt task does not include the positions of objects in the environment. Therefore, if the robot were to drop the wrench in this task, the reset controller would not learn to retrieve it. In the neural network grasping experiment, we explicitly provide the wrench position to the policy by using a Kinect-based object detector, but this requires us to explicitly enumerate the relevant objects in the scene. Augmenting the state by using vision, perhaps by drawing on ideas presenting on recent work training deep visuomotor policies [20], could allow our method to extend to more complex tasks, but in general there are many tasks that may simply be too difficult for a robot to reset automatically.

Another limitation is that, although the forward controllers are aware of the initial state induced by the preceding forward controller, they do not attempt to optimize for the performance of the forward controller that follows them. This type of information could be fed to the forward controllers by including the value function from the first time step of the next controller into the terminal cost at the last step of the previous one, since the value function is already computed as part of the LQG backward pass. A further extension of this method might involve tasks with more complex branching structure. Our grasping task already involves a discrete decision point in deciding which forward grasping controller to invoke based on the position of the wrench, and this idea can be generalized to produce a tree of controllers for complex tasks with many decision points.

Acknowledgements This research was funded in part by DARPA through a Young Faculty Award and under Award #N66001-15-2-4047, as well as by the Army Research Office through the MAST program.

References

[1] J. Kober, E. Oztop, and J. Peters, "Reinforcement learning to adjust robot movements to new situations," in *Robotics: Science and Systems (RSS)*, 2010.

[2] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal, "Learning and generalization of motor skills by learning from demonstration," in *International Conference on Robotics and Automation (ICRA)*, 2009.

[3] M. Deisenroth, C. Rasmussen, and D. Fox, "Learning to control a low-cost manipulator using data-efficient reinforcement learning," in *Robotics: Science and Systems (RSS)*, 2011.

[4] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *International Conference on Robotics and Automation (IROS)*, 2004.

[5] R. Tedrake, T. Zhang, and H. Seung, "Stochastic policy gradient reinforcement learning on a simple 3d biped," in *International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[6] T. Geng, B. Porr, and F. Wörgötter, "Fast biped walking with a reflexive controller and realtime policy searching," in *Advances in Neural Information Processing Systems (NIPS)*, 2006.

[7] G. Endo, J. Morimoto, T. Matsubara, J. Nakanishi, and G. Cheng, "Learning CPG-based biped locomotion with a policy gradient method: Application to a humanoid robot," *International Journal of Robotic Research*, vol. 27, no. 2, pp. 213–228, 2008.

[8] 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.

[9] W. Li and E. Todorov, "Iterative linear quadratic regulator design for nonlinear biological movement systems," in *ICINCO (1)*, 2004, pp. 222–229.

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

[11] E. Theodorou, J. Buchli, and S. Schaal, "Reinforcement learning of motor skills in high dimensions," in *International Conference on Robotics and Automation (ICRA)*, 2010.

[12] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[13] 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.

[14] F. Stulp, E. Theodorou, and S. Schaal, "Reinforcement learning with sequences of motion primitives for robust manipulation," *Robotics, IEEE Transactions on*, vol. 28, no. 6, pp. 1360–1370, Dec 2012.

[15] C. Daniel, G. Neumann, O. Kroemer, and J. Peters, "Learning sequential motor tasks," in *International Conference on Robotics and Automation (ICRA)*, 2013.

[16] R. Lioutikov, O. Kroemer, J. Peters, and G. Maeda, "Learning manipulation by sequencing motor primitives with a two-armed robot," in *International Conference on Intelligent Autonomous Systems (IAS)*, 2014.

[17] S. Manschitz, J. Kober, M. Gienger, and J. Peters, "Learning to sequence movement primitives from demonstrations," in *Intelligent Robots and Systems (IROS)*, 2014.

[18] G. Konidaris, S. Kuindersma, R. Grupen, and A. Barto, "Robot learning from demonstration by constructing skill trees," *International Journal of Robotics Research*, vol. 31, no. 3, pp. 360–375, 2012.

[19] G. Konidaris and A. S. Barreto, "Skill discovery in continuous reinforcement learning domains using skill chaining," in *Advances in Neural Information Processing Systems (NIPS)*, 2009.

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

[21] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[22] S. Levine and V. Koltun, "Guided policy search," in *International Conference on Machine Learning (ICML)*, 2013.

[23] R. Lioutikov, A. Paraschos, G. Neumann, and J. Peters, "Sample-based information-theoretic stochastic optimal control," in *International Conference on Robotics and Automation (ICRA)*, 2014.