Residual Policy Learning

Tom Silver*, Kelsey Allen*, Josh Tenenbaum, Leslie Kaelbling
MIT
{tslvr, krallen, jbt}@mit.edu, lpk@csail.mit.edu

Abstract—We present Residual Policy Learning (RPL): a simple method for improving nondifferentiable policies using model-free deep reinforcement learning. RPL thrives in complex robotic manipulation tasks where good but imperfect controllers are available. In these tasks, reinforcement learning from scratch remains data-inefficient or intractable, but learning a residual on top of the initial controller can yield substantial improvement.

We study RPL in five challenging MuJoCo tasks involving partial observability, sensor noise, model misspecification, and controller miscalibration. By combining learning with control algorithms, RPL can perform long-horizon, sparse-reward tasks for which reinforcement learning alone fails. Moreover, we find that RPL consistently and substantially improves on the initial controllers. We argue that RPL is a promising approach for combining the complementary strengths of deep reinforcement learning and robotic control, pushing the boundaries of what either can achieve independently.

I. INTRODUCTION

Deep reinforcement learning (RL) methods are increasingly common and increasingly successful in robotic manipulation domains like grasping and pushing [1, 2, 3, 4, 5]. But for most complex problems of interest, learning from scratch remains intractable. For example, consider the task illustrated in Figure 1a. A simulated Fetch robot must pick up and use a hook to drag an out-of-reach block to a target location. The only reward offered is a positive signal once the block reaches the target. This long-horizon, sparse-reward problem remains out of reach for current deep RL methods. In contrast, it is relatively straightforward to hand-design a policy that accomplishes this hook task perfectly in simulation (see Section V-B).

While a hand-designed policy may be robust to variations in the initial block position and target, it will likely break down with more dramatic variations in the task. For example, consider the task variation illustrated in Figure 1b. The robot must now move a more complex rigid object to the goal. The task is further complicated by static “bumps” on the table that may impede the movement of the hook and object. Moreover, the robot’s state includes no information about the bumps, which randomly regenerate at each trial, nor information about the object’s shape, which is randomly selected from a library of 100 diverse objects. The policy designed for the original task sometimes succeeds in this setup, but more often fails.

What should be done when a policy – be it a hand-designed policy, a model-predictive controller, or any other controller mapping states to actions – performs below par? One path forward is to manually tweak the policy. This option, while potentially laborious, may suffice for some problems. But for other problems like the complex hook task described above, it is unclear how to even begin improving the policy by hand.

In this work, we propose Residual Policy Learning (RPL): a simple method for improving policies using deep reinforcement learning. Our main idea is to augment arbitrary initial policies by learning residuals on top of them. Given an initial policy \( \pi : S \rightarrow A \) with states \( s \in S \) and actions \( a \in A \subseteq \mathbb{R}^d \), we learn a residual function \( f_\theta : S \rightarrow A \) so that we have a residual policy \( \pi_\theta : S \rightarrow A \) given by

\[
\pi_\theta(s) = \pi(s) + f_\theta(s)
\]

Observe that \( \nabla_\theta \pi_\theta(s) = \nabla_\theta f_\theta(s) \), i.e. the gradient of the policy does not depend on the initial policy \( \pi \). We can therefore use policy gradient methods to learn \( \pi_\theta \) even if the initial policy \( \pi \) is not differentiable.

There are two ways to see the function of the residual. If the initial policy is nearly perfect, the residual \( f_\theta \) may be viewed as a corrective term. But if the initial policy is far from ideal, we may interpret the outputs of \( \pi \) as merely “hints” that may be used (or ignored) by \( f_\theta \) as it learns to substantially improve on the original policy. In practice, these two interpretations of the residual represent ends of a spectrum. We study problems all along this spectrum in this paper.

We present experimental results on several complex manipulation tasks which include issues central to robotics and controller design: partial observability, sensor noise, model misspecification, and controller miscalibration. Our experi-

---

* Equal contribution.
ments are designed to investigate when and to what extent the following two claims hold:

1) RPL improves on initial policies; and
2) RPL is more data-efficient than learning from scratch.

We examine two common sources of initial policies: hand-designed policies and model predictive controllers (MPC). In all cases, RPL is able to substantially improve on the original policies, while requiring far less data than learning from scratch to achieve the same performance. Furthermore, in complex manipulation tasks like the hook one discussed above, RPL succeeds where learning from scratch is intractable and hand-designing perfect policies is unrealistic.

II. RELATED WORK

Residual Policy Learning can be seen as tackling two separate but related questions: how to improve imperfect controllers, and how to make deep reinforcement learning methods more data efficient and able to handle longer horizon planning.

There has been a substantial body of work on improving the data efficiency of deep reinforcement learning by combining model-free and model-based approaches. These methods often first learn a dynamics model and then use this dynamics model to simulate experience [6, 7, 8] or compute gradients for model-free updates [9, 10]. Another set of approaches uses the learned dynamics model (or inverse dynamics model) to perform trajectory optimization or model predictive control [11, 12]. Further work uses such model-based methods to guide a model-free learner in a DAGGER-style imitation strategy [13]. More recent work has shown an equivalence between model-free and model-based reinforcement learning with goal-conditioned value functions [14], and used this to improve model-free reinforcement learning data efficiency. RPL differs from all above approaches in that it is more general than using model-based strategies as the basis for a controller which can be improved upon. Indeed, RPL can be used with a model-based reinforcement learner as the initial policy, but is not restricted to this setting.

RPL can also be seen as a form of imitation learning. This set of approaches considers an expert which provides demonstrations of a task to a learner. Most approaches then attempt to copy the expert’s strategy [13, 15], or to use inverse reinforcement learning to infer goals and subgoals of the expert agent [16, 17]. Such approaches are sensitive to the trajectories of experts, and do not necessarily perform well when the expert only provides partial “hints” instead of full trajectories [15]. By contrast, RPL performs well when the expert policy only provides partial information, and is robust to data mismatch since the controller is run on the same data as the learned correction.

From robotics, many methods exist for learning different aspects of the perception, control, execution pipeline. Focusing on control specifically, Bayesian optimization approaches are popular for learning controllers based on Gaussian process models of objective functions to be optimized [18, 19, 20, 21, 22]. Learning an accurate dynamics model is another central focus for robotics (termed system identification), and has been approached using analytic gradients [23, 24], finite differences [25] or Bayesian Optimization [7]. RPL is more general than these methods as it does not presuppose which aspect of the controller needs correction. This is particularly valuable in partially observable settings, where it is unclear how to learn a good dynamics model or design a better objective function.

In the case of dynamics learning, our work is inspired by Ajay et al. [26] and Kloss et al. [27] who learn a correction to an analytical physics model in order to perform better model predictive control. RPL is more general in that it can learn to correct the model implicitly by correcting the policy, but can also provide corrections which could not be provided by dynamics corrections (such as partially observable or noisy domains).

Concurrent work by Johannink et al. [28] also proposes residual reinforcement learning, and focuses on showing the value of the approach for real robots in a task of block insertion, investigating the effects of variation in the initial state, control noise, and the transfer from sim to real. Here we aim to show the power of residual policies for a variety of different tasks that disentangle several sources of difficulty: partial observability, sensor noise, model misspecification, and controller miscalibration. We also empirically analyze the root cause of residual policy learning’s success by introducing a baseline that uses the initial policy only as an “expert” to guide exploration.

III. BACKGROUND

Residual Policy Learning (RPL) operates within a standard Markov Decision Process (MDP) framework. An MDP is a tuple $M = (S, A, R, T, \gamma)$ where $s \in S$ are states, $a \in A$ are actions, $R(s, a) \in \mathbb{R}$ is the reward for taking action $a$ in state $s$, $T(s, a, s') = \Pr(s'\mid s, a)$ is the probability of transitioning to state $s'$ following state $s$ and action $a$, and $0 \leq \gamma \leq 1$ is a temporal discount factor. We assume all trajectories or episodes sampled from the MDP have a finite number of actions (horizon) $h$. In all of the experiments described in this paper, states and actions are real-valued vectors. A policy $\pi : S \rightarrow A$ maps states to actions. Given an initial state $s_0$, the reinforcement learning problem is to find a policy $\pi$ that maximizes expected rewards discounted over time $J = \mathbb{E}_{s_t \sim M} \left[ \sum_{t=0}^{h} \gamma^t R(s_t, \pi(s_t)) \right]$.

Let $Q^\pi : S \times A \rightarrow \mathbb{R}$ be the action-value function that gives the expected future discounted rewards following policy $\pi$. Many reinforcement learning methods make use of the Bellman equation for the action-value function

$$Q^\pi(s, a) = \mathbb{E}_{s' \sim T(s, a, \cdot)} [R(s, a) + \gamma Q^\pi(s', \pi(s'))]$$

Actor-critic methods learn both a parameterized policy $\pi_\theta$ (the actor) and a parameterized action-value function $Q_\phi$ (the critic). The critic is trained with a loss function derived from the Bellman equation above and the actor is trained to produce actions that maximize the critic. This approach is typically more stable than training the actor alone.
Fig. 2: Illustration of the original ReactivePush policy and RPL on the SlipperyPush task. The original policy was designed so that the robot pushes the block to the target (red sphere) when the block has high friction. When the block has lower friction than anticipated, the block is pushed off the table (top row). RPL, our proposed method, learns to correct the faulty policy and accomplish the task after 100,000 simulator steps (bottom row). Given the same number of time steps, reinforcement learning from scratch results in a policy where the robot does not touch the block at all (not shown).

In this work, we use Deep Deterministic Policy Gradients (DDPG) [29], an actor-critic method that works well in domains with continuous states and actions. In DDPG, the actor is updated following the deterministic policy gradient
\[ \nabla_{\theta} J \approx \mathbb{E}_{s_t \sim M}(\nabla_a Q_\theta(s, a)|_{s=s_t, a=\pi_\theta(s_t)} \nabla_\theta \pi_\theta(s)|_{s=s_t}) \]

DDPG makes use of experience replay, in which transitions \((s_t, a_t, r_t, s_{t+1})\) sampled from the environment are stored in a replay buffer. During training, transitions are then randomly drawn from the replay buffer in an effort to break the correlation between consecutive transitions.

Hindsight Experience Replay (HER) [5] extends experience replay to dramatically improve data efficiency in domains with sparse binary rewards (goals) like those we consider in our experiments. In HER, the reward function and policy are additionally parameterized by a goal \(g\) so that they become \(R(s, a, g)\) and \(\pi(s, g)\) respectively. For our purposes, the goal \(g\) is a subvector of the final state of an episode. During training, each transition added to the replay buffer includes a goal \(g_t\) that was achieved “in hindsight,” i.e. the goal that was actually reached at the end of the training episode. Given a sampled transition \((s_t, a_t, r_t, s_{t+1}, g_t)\), the policy is then updated according to the reward \(R(s_t, a_t, g_t)\). This trick is especially useful early in training when the chance of achieving nonzero rewards is low. We combine HER and DDPG for all of the experiments presented in this work.

IV. RESIDUAL POLICY LEARNING (RPL)

In Residual Policy Learning (RPL), we begin with an MDP \(M = (S, A, R, T, \gamma)\) and an initial policy \(\pi : S \to A\). Our goal is to learn a residual \(f_\theta\) to create an improved final policy \(\pi_\theta(s) = \pi(s) + f_\theta(s)\). We create a residual MDP \(M^{(\pi)} = (S, A, R^{(\pi)}, T^{(\pi)}, \gamma)\) where
\[ T^{(\pi)}(s, a, s') = T(s, \pi(s) + a, s') \]

If we view \(M^{(\pi)}\) as an MDP like any other, we see that the residual that we wish to learn, \(f_\theta\), is a policy in this MDP. We can thus apply standard reinforcement learning techniques to learn the residual.

Residual Policy Learning is as simple as that: given an MDP and an initial policy, create a residual MDP and proceed with deep RL. We now describe a few minor extensions that can improve performance and data efficiency in practice.

A. Initializing the Residual

A desirable property of RPL is that it should never make a good initial policy worse. In the extreme case, if an initial policy is perfect, then we would like the residual policy to have no influence. We therefore endeavor to initialize the residual function so that \(f_\theta(s) = 0\) for all \(s \in S\). We do this by initializing the last layer weights of the network to be zero.

B. RPL with Actor-Critic Methods

RPL learns a residual on the output of an initial policy. Actor-critic methods like DDPG involve not only a policy but also a learned action-value function. Ideally we would initialize the critic so that it accurately captures the expected future rewards obtained by following the initial policy. However, finding the action-value function for a given policy is
generally as hard as reinforcement learning and cannot be done without knowledge of the MDP.

If we begin with a perfect initial policy and an uninformed critic, the policy performance may degrade, since it is trained with reference to the critic. We therefore propose to train the critic alone for a “burn in” period while leaving the policy fixed. We can determine an appropriate burn in length automatically by monitoring the critic loss function and waiting for it to dip below a threshold $\beta$, which becomes a hyperparameter of our method.

C. Recurrent RPL for POMDPs

RPL can also be extended to handle Partially Observable Markov Decision Processes (POMDPs). Generally, this is done in deep reinforcement learning by making $\pi_\theta(s)$ recurrent. In practice, this is challenging for DDPG, and so we present an approximation by simply considering a “history” of previous states. This is equivalent to writing $s = \{s_{t-n} \mid n \in 0, ..., N\}$ with $t$ being the current time-step, and $N$ the history length. While the history length can take on any value, we found that a history length of just 1 (meaning the policy considers the current state and previous state) to be effective. We take advantage of this extension in our NoisyHook experiment in which observation noise obscures the input to the policy.

V. Experiments

Here we investigate when and to what extent the following two claims hold:

1) RPL improves on initial policies; and
2) RPL is more data-efficient than learning from scratch.

We are also interested in an extreme version of the latter claim: that RPL can learn good policies in tasks where learning from scratch fails to solve the task.

A. Tasks

We study five simulated manipulation tasks. All environments are implemented in MuJoCo [30]. To provide direct comparison with previous work, we begin with a Push task and a PickAndPlace task, both taken from [31]. We then present three more difficult tasks that have not been previously considered. All tasks involve a Fetch robot positioned in front of a table top. Following previous work, we parameterize the action space in terms of changes to position in world coordinates [31]. A fourth action coordinate modulates the gripper’s two fingers symmetrically. (In the push tasks, the gripper is locked, and the fourth action coordinate has no effect.) All actions are normalized so that the resulting action space is $A = [-1, 1]^4$. The state spaces and rewards vary per task; we describe these and more environment details below.

1) Push: This task is taken directly from [31]. The objective is to move an object (a cube) to a target location on the table surface. The grippers are locked, forcing the robot to push, rather than pick and place, the object. At the beginning of each episode, the initial position of the object and the target location are randomized. The state space includes:

- Gripper $xyz$ position and velocity (6 dims)
- Object $xyz$ position, $ypr$ rotation, velocity, and angular velocity (12 dims)
- Object position relative to the gripper (3 dims)
- Gripper finger joint states and velocities (4 dims)

for a total dimensionality of 25. To use Hindsight Experience Replay, we must also specify achieved and desired goals. Here the achieved goal is the three-dimensional final position of the object and the desired goal is the target location. Rewards are sparse and binary: a reward of 1 is given when the object is within a small radius around the target location and 0 otherwise. The episode is counted as a success if the last reward is 1, i.e. the goal is achieved. Episode lengths are 50 and do not terminate early.

2) SlipperyPush: Here we present a slight modification to the original Push environment. In the original environment, the object has a sliding friction coefficient of 1.0. In this SlipperyPush environment, the same coefficient is set to 0.18. The initial state randomization, state space, goals, rewards, and horizon are otherwise identical to Push.

3) PickAndPlace: This task is taken directly from [31]. As in the previous tasks, the objective is to move an object (a cube) to a target location. However, the target location may now be either on the table top or in the air above the table. At the beginning of each episode, the $xy$ position for the target location is randomly sampled as before. Then with 0.5 probability, the $z$ location is set to be on the table surface; otherwise, the $z$ location is randomly sampled to be above the table surface. As mentioned above, the gripper is now unlocked so that the fingers open and close following the fourth action dimension. All other environment details are unchanged with respect to Push.

4) NoisyHook: In this task, the table top is extended away from the robot and the initial block positions are such that the robot cannot reach the block with its gripper. A new hook object is introduced and positioned to the right of the robot (see Figure [1]). The objective is to move the cube to a target location, but now the robot must use the hook to manipulate the cube. The target location is randomly initialized so that it lies between the cube and the robot. Thus the robot must reach the hook around the cube and pull back in order to accomplish the task.

In addition to the 25 state dimensions included in the previous tasks, the state space now includes information about the hook:

- Hook $xyz$ position, $rpy$ rotation, velocity, and angular velocity (12 dims)
- Hook position relative to the gripper (3 dims)

for a total of $15 + 25 = 40$ dimensions. Rewards and goals are the same as in previous tasks; we provide no additional shaping rewards.
This NoisyHook task is further complicated with the addition of observation noise. We suppose that the robot has precise proprioception but has significant uncertainty about the positions of the hook and cube. At each time step, we add IID diagonal Gaussian noise (µ = 0.0, σ² = 0.025) to the xy position of the block and the xyz position of the hook, as well as the rotation of both objects. Since the achieved goals are derived from the state, they too are affected by this observation noise.

Since this task requires the robot to first pick up the hook, bring it to the object, and pull the object to the target location, we double the episode length from the previous tasks for a total of 100 frames.

5) ComplexHook: This final task again features a hook and an object that must be moved to a target location. There is no longer Gaussian noise added to the state. There is, however, significant uncertainty of two different, structured kinds. We first replace the simple cube from previous tasks with complex objects that vary substantially in mass, friction, and shape. We use 100 objects taken from previous work by Finn et al. [1]. The object meshes were originally downloaded from thingiverse.com and include bowls, teddy bears, and small chairs among many other shapes. No information about the object shape or physical parameters are included in the state. To accomplish this task robustly, a policy must work across all possible objects.

To introduce a second source of structured uncertainty, we simulate large “bumps” on the table. A bump is a rigid box that is fixed to the table top. The width, length, height, position, and count of the bumps are randomly selected. See Figures 1 and 5 for two examples. Note crucially that no information about the bumps are included in the state space. Thus the complete state space and other task parameters remain unchanged from NoisyHook.

B. Initial Policies

In RPL, we begin with an environment and an initial policy π : S \rightarrow A and we learn to improve on that initial policy. Here we describe the initial policies that were used in our experiments. Each initial policy corresponds to one of the environments in Section V-A except for the last policy, which is used for both hook environments.

1) DiscreteMPCPush: Suppose we have a learned or known transition model \( Pr(s'|s,a) \) that can be queried to predict the state trajectories and rewards that may result from a sequence of actions taken from an initial state. In Model Predictive Control (MPC), we use this transition model to select each action taken by the policy \( \pi \). More specifically, given the current state \( s_t \), an MPC policy will internally consider multiple sequences of actions \( a_t, a_{t+1}, \ldots, a_h \) and compute the expected rewards accrued for each sequence. The first action in the best sequence is then the output of \( \pi(s_t) \). To design an MPC policy, we must therefore specify the model and a procedure for selecting possible action sequences.

In high-dimensional tasks with long horizons, sparse rewards, and continuous states and actions, MPC is intractable without an efficient mechanism for selecting action sequences. Here we opt to discretize the action space as a means to simplify the search. In particular, rather than consider the infinite number of possible gripper movements, we consider only six, one per cardinal direction. We can then use a discrete graph search to explore possible action sequences.

We develop a discrete MPC policy for the Push task. The model is a perfect copy of the environment (i.e. a separate instance of MuJoCo). We further improve the policy by introducing an informative heuristic to guide the discrete search. The heuristic is a tuple \( (d_1, d_2) \) where \( d_1 \) is the distance between the object and the target location and \( d_2 \) is the distance between the gripper and the “push location.” The push location is meant to be the desired position of the gripper for pushing the block to the target; it is approximated by extending the vector difference between the object and target location by a small amount corresponding to the radius of a sphere circumscribed around the object. The heuristic is such that the second entry \( d_2 \) is only used to break ties when the first entry \( d_1 \) matches, i.e. it is lexicographic. We use this heuristic to perform a best-first search with 10 node expansions per environment step. If no action sequence is found that improves on the heuristic of the current state, a no-op action is taken. Otherwise the first action in the sequence with the best found heuristic is taken.

2) ReactivePush: Our second policy is designed for pushing an object to a target location. While this policy works nearly perfectly in the original Push task, its performance drops dramatically when the sliding friction on the block is reduced as in the SlipperyPush task. Given an input state, the policy checks the following conditional statements in order until one holds and proceeds accordingly.

1) If the object is already at the target location, do nothing.
2) If the block is between the gripper and the target location, move the gripper towards the target location.
3) If the gripper is above the push location (see definition in DiscreteMPCPush), move the gripper down to prepare to push.
4) Move the gripper to above the push location.

To determine whether the object or gripper is “at” a location, we measure the distance and check if it is below a global threshold. The other key hyperparameter is a gain that determines how far the gripper moves at each time step. We manually tuned this gain to achieve near optimal performance on the original Push task.

3) ReactivePickAndPlace: Our third policy is designed to pick up a cube and bring it to a target location on or above the table. Given an input state, the policy checks the following conditional statements in order until one holds and proceeds accordingly.

1) If the object is already at the target location, do nothing.
2) If the gripper is grasping the object, move towards the target location.
3) If the object is between the gripper fingers (but not grasped), close the gripper.
4) If the gripper is above the object
5) ... and the gripper is closed, open the gripper.
6) ... and the gripper is open, move the gripper down.
7) Move the gripper towards the location above the object.

To determine whether the gripper is grasping the object, we check that the object location is between the two fingers and that the fingers are not more than the block width apart. We again use the distance threshold and gain hyperparameters described above.

4) ReactiveHook: Our fourth and final policy is designed to pick up a hook, move it behind and to the right of an object, and push and pull the object towards a target location. The policy works nearly perfectly when the object is a cube, the table is clear of obstacles, and the observations are noise-free (see Figure 1a). However, the policy performance drops substantially when transferred to the NoisyHook and ComplexHook tasks. Given an input state, the policy checks the following conditional statements in order until one holds and proceeds accordingly.

1) If the object is already at the target location, do nothing.
2) If the hook is not grasped and lifted above the table, grasp and lift the hook using ReactivePickAndPlace as a subroutine.
3) If the hook is not beyond and to the right of the object, move forward or rightward accordingly.
4) Move the gripper following the vector difference between the object and the target location.

The grasp position is fixed so that the robot always attempts to pick up the same part of the hook (near the bottom). In addition to the global threshold and gain hyperparameters, we use knowledge of the length and width of the hook to determine gripper movements as a function of desired hook movements.

C. Architectures and Training Details

RPL is a simple and general approach, and is indifferent to the deep RL method applied or architecture used, and could even be applied with alternative mechanisms for learning such as Bayesian Optimization. However, for consistency, in our experiments we use the same actor-critic architecture, with Deep Deterministic Policy Gradients [29] and Hindsight Experience Replay [5]. The network consists of 3 fully connected layers of 256 units each, with ReLU non-linearities. We use mostly the same hyperparameters as in [31], given in the appendix. We slightly tuned some of these values for RPL. Our only substantial modification is to initialize the last layer of the network to zeros, so that the policy starts with the base controller (as described in section IV-A). Thresholds (described in section IV-B) are also given in the appendix.

When training in noisy environments, we use a history of 1 (see Section IV-C). We considered two variants. In the first variant, the states are concatenated and fed to the network: \( f_0(s_1, s_2) \). In the second variant, we consider the average of the features obtained for the states: \( 0.5 (f_0(s_1) + f_0(s_2)) \). In practice, we found the second variant to work better, and so use it for all noisy environments.

D. Baselines

We consider three baselines for all experiments. First, in all experiments, we show the result of running the initial policy without learning. Second, we show the result of learning from scratch for all environments.

One hypothesis for why residual learning might be helpful is that the initial policy provides a smart means for exploration. We introduce a baseline, “Expert Explore”, which builds on the classic \( \epsilon \)-greedy exploration method. The baseline requires one further parameter, \( \alpha \), denoting the proportion of actions taken which follow the initial “expert” policy. The selected action then follows:

\[
\begin{align*}
  z &= \text{rand}(0, 1) \\
  \alpha &= \begin{cases} 
  \pi(s) & \text{if } z < \epsilon \\
  \text{rand()} & \text{if } z < \epsilon (1 - \alpha) \\
  f_0(s) & \text{if } z > \epsilon 
\end{cases}
\end{align*}
\]

The agent acts \((1 - \epsilon)\% \) of the time according to the learned policy, \( \epsilon \times \alpha \% \) according to the expert, and the rest of the time takes random actions. This is not unlike policy-reuse methods, which treat the previously learned policy as the expert, and design an \( \epsilon \)-greedy strategy for sampling new policies [32]. \( \epsilon \) and \( \alpha \) were determined for our experiments by performing a small grid search to optimize performance for the hook environment. These values are given in the appendix, and used for all experiments.

E. Results

Here we present empirical and qualitative results for RPL across the five complex manipulation tasks described in Section V-A. For each task, we show RPL’s superior data efficiency and performance compared to the three baselines described in Section V-D. All empirical results are presented with mean and standard deviation across five random seeds.

1) DiscreteMPCPush in Push: In this experiment, we examine whether RPL can overcome the limitations of an MPC controller that makes coarse approximations in an effort to trade performance for speed. In particular, we use the DiscreteMPCPush as our initial policy for the Push task.

We graph the success rates of RPL and the baselines in Figure 3a. The success rate of DiscreteMPCPush starts around 0.5. We noticed three common sources of suboptimality for this initial policy. First, the limited node expansions per MPC call, which is necessitated by the speed bottleneck of querying the MPC’s model, means that a good action sequence is not always found. Second, the discreteness of the actions sometimes leads to circuitous executions in which the episode ends before the object reaches the target. Third, the heuristic...
used to guide the MPC’s search, while very informative, can also be misleading in rare cases where it is preferable to move the gripper rather than moving the block. These failure modes are especially common when the gripper must move from one side of the cube to the other, since the cube acts as an obstacle in this context.

We confirm the results reported in previous work [31] that learning from scratch with DDPG and HER works well in this domain, converging to a success rate of nearly 1.0 after roughly 2 million simulator steps. The performance of RPL before convergence greatly surpasses both the initial policy and learning from scratch, while still converging to a perfect success rate. For example, RPL takes an order of magnitude fewer training samples to reach an average success rate of 0.9 versus the learning from scratch baseline.

To analyze the source of RPL’s superior data efficiency, we turn to the performance of the Expert Explore baseline. We find that this baseline also improves on learning from scratch, but does not achieve the efficiency of RPL. For example, the baseline takes roughly 5x as many samples to reach an average success rate of 0.9. This suggests that RPL’s advantage in this Push task derives partly from more efficient exploration, but also from good initialization.

2) ReactivePush in SlipperyPush: Our second experiment is designed to capture the common phenomenon of model misspecification. We tuned the ReactivePush policy to achieve near perfect performance in the original Push task. We now transfer this policy to the SlipperyPush task in which the sliding friction coefficient of the cube is 5x smaller.

The success rates of RPL and the baselines on the SlipperyPush task are shown in Figure 3b. As expected, the ReactivePush policy is not perfect, achieving a success rate of around 0.45. The most common failure mode of this initial policy is when the gripper pushes the slippery cube too hard and the cube slides off the table. In other cases, the cube does not fall off, but is pushed back and forth across the goal without converging. A representative trial is illustrated in Figure 2 (top row).

As in the first experiment, we find that RPL is far better before convergence, and converges to the same perfect success rate as model-free learning from scratch. In this case, we find that the Expert Explore baseline quickly matches and keeps pace with RPL, indicating that here RPL’s advantage may be attributed to improved exploration early in training.

3) ReactivePickAndPlace in PickAndPlace: In this experiment, we consider an example of a poorly calibrated initial policy that leads to detrimental oscillatory behavior. Such oscillations are a common issue in stateless robotic control when gains are improperly tuned. To create a representative scenario, we start with the ReactivePickAndPlace policy and artificially increase the gains. Oscillations quickly arise, e.g. when the gripper overshoots the waypoints implicit in the design of the policy. These oscillations cause the success rate of the ReactivePickAndPlace to drop to roughly 0.5, as seen in Figure 3c.

As reported in previous work [31], learning from scratch with DDPG and HER requires far more data to reach a success rate of 1.0 in PickAndPlace versus Push. Here we find the data efficiency of RPL to be substantially better. RPL converges to a success rate of 1.0 after roughly 100,000 simulator steps, which represents a nearly 10x improvement over learning from scratch. Comparing with the Expert Explore baseline, we find that not all of the advantage can be explained by improved exploration; the good initialization of the policy is also to credit.

We make a few additional remarks. First, it was not a priori obvious that the initial policy would aid RPL here as much as it apparently does. By design, we know that the policy is close in “gain space” to a near optimal one, but that does not guarantee that the policy is similarly close in “residual weight space.” Fortunately, it seems the two notions coincide here. A second interesting observation is that the performance of RPL drops starkly early in training before quickly recovering and surpassing the baselines. This is a manifestation of
the issue discussed in Section IV-B whereby the critic is initialized poorly with respect to the actor. We found that decreasing the burn-in parameter $\beta$ mitigated the drop here but did not significantly affect the time to convergence. We thus left the results as they are for the benefit of discussion.

4) ReactiveHook in NoisyHook: Now we turn to another prevalent problem in robotic control – sensor noise – and investigate whether RPL can improve the robustness of a sensitive initial policy. As discussed in Section V-A, the NoisyHook task features Gaussian noise applied to the positions and rotations of the block and hook. While the ReactiveHook policy is nearly perfect in a noiseless version of the same task, the policy proves to be quite sensitive to the sensor noise. We observe diverse failures modes throughout the course of execution: the gripper often moves to a wrong position, sometimes fails to pick up the hook, and other times drops the hook (thinking that it already has). As shown in Figure 4a, the success rate of the initial policy is roughly 0.15, far lower than in our previous experiments.

In this experiment, we make use of the two frame policy architecture described in Section IV-C to cope with sensor noise. We use the same architecture for all three learning methods for comparison.

Learning from scratch with DDPG and HER fails in this task, never achieving a nontrivial success rate. This failure is not surprising given the long horizon and sparse rewards in the task. Interestingly, the Expert Explore baseline fails here as well. We speculate that this failure is due to the fact that the hook is so often dropped by the initial policy.

Whereas this NoisyHook task proves intractable for all baselines, we see that RPL quickly converges to a success rate of roughly 0.8. This represents the first instance of RPL obtaining strong performance in a task that is both out of reach for current deep reinforcement learning methods and nontrivial for robotic control alone. Moreover, the results suggest that RPL is a promising method for overcoming the common challenge of sensor noise.

5) ReactiveHook in ComplexHook: In this final experiment, we studied structured uncertainty inspired by the common mismatch between physics simulators and real robotics tasks. As described in Section V-A, the ComplexHook task contains two challenging innovations over the noiseless hook task: bumps are randomly scattered across the table surface; and the object takes on a variety of shapes, masses, and coefficients of friction. We observed that each of these two innovations independently cause the ReactiveHook policy performance to drop by roughly 20%. With both changes present, the initial policy success rate drops to 0.55, as shown in Figure 4b.

It is worth noting that a random or null policy is occasionally successful in this task. The scene randomization is such that the objects will sometimes initially collide with a bump and land at the target location due only to the force of gravity. With this in mind, we see that learning from scratch with DDPG and HER does not obtain any nontrivial success rate, as in the previous experiment. We again find that the policy never causes the gripper to touch the hook, let alone move it to reach the object.

Interestingly, the Expert Explore baseline does achieve a nontrivial success rate, eventually slightly surpassing the success rate of the initial policy. This task is easier than NoisyHook from the perspective of the expert baseline if only because the initial success rate is much higher. We speculate that the lack of sensor noise also plays a significant role.

Finally, RPL learns a robust policy with strong data efficiency, converging at a success rate just below 0.8. This result is the first where we find both RPL and the expert baseline to succeed and converge, but to different values. The fact that RPL is able to achieve this success rate is fairly remarkable given the diversity in the objects and obstacles, and the fact that the state contains no information about this diversity. RPL has apparently learned a “compliant” policy that works for most objects and obstacles without discretion. We show one intriguing example of RPL succeeding where the initial policy fails in Figure 5.

VI. DISCUSSION AND CONCLUSION

We have described Residual Policy Learning (RPL), a simple method that combines the strengths of deep reinforcement learning and robotic control. Our experimental results suggest that RPL is a powerful approach to deal with pervasive issues in complex manipulation tasks such as sensor noise, model misspecification, controller miscalibration, and partial observability. We find that RPL consistently improves on initial policies and achieves better data efficiency than learning from scratch. Furthermore, RPL can improve on initial policies for long-horizon, sparse-reward problems where learning from scratch fails.

We postulate two main causes for the success of RPL. First, as described in Section IV we take care to initialize the residual policy so that its output at first matches the initial policy. When the initial policy is strong, this initialization gives RPL a clear boost. The second cause of RPL's success is improved exploration early on during training. In learning from scratch with sparse rewards and long horizons, the first successful trajectory must be discovered by chance. Hindsight Experience Replay is designed to face this challenge, but RPL offers a more direct solution. RPL can discover successful trajectories immediately if the initial policy produces them with nontrivial frequency. To disentangle these two causes, we introduced the Expert Explore baseline described in Section V-D which has the benefit of the second cause but not the first. Empirically we find this baseline performance to lie midway between RPL and learning from scratch, suggesting that both causes contribute substantially to RPL's success. Further, RPL is able to improve upon hard-coded policies in part because of its ability to “compress” state representations for action. We believe this is particularly helpful for handling noisy sensor data.
Fig. 4: RPL and baseline results for the NoisyPush and ComplexPush tasks. For both tasks, the “Initial” policy is ReactiveHook. In the first task, RPL quickly and substantially improves on the initial policy while the other two learning methods fail. In the second task, RPL again improves on the initial policy. See also Figures 1b and 5 for illustrations of this task.

Fig. 5: Illustration of the original policy and RPL on the Complex Hook task. The original policy was designed to work when the object is a simple cube and the table has no obstacles (see Figure 1b). The same policy pushes a larger complex object off the table rather than to the target (red sphere) as required (top row). RPL, our proposed method, learns to improve the policy that pulls the object to the target (bottom row). The learned policy exhibits interesting behavior that qualitatively resembles lifting the hook to avoid obstacles and reaching around at a wider angle than originally programmed.

Though the five case studies we have presented all involve robotic manipulation with DDPG and HER, RPL is far more general than any specific task domain or deep reinforcement learning method. The method we have described can be immediately applied in any domain with continuous action parameterizations and with any gradient-based learning method. However, RPL is especially well suited for complex manipulation because of the availability of good but imperfect initial policies and the long-horizon, sparse-reward tasks that naturally arise.

In recent years, complex manipulation problems have been at the forefront of research in robotics and deep reinforcement learning. Both fields have made significant strides in often complementary directions. RPL should be viewed as one piece of a larger effort to combine the strengths of both approaches. We conjecture that solving the hardest open problems in manipulation will require such a synthesis.

ACKNOWLEDGMENTS

We gratefully acknowledge support from NSF grants 1523767 and 1723381; from ONR grant N00014-13-1-0333; from AFOSR grant FA9550-17-1-0165; from ONR grant N00014-18-1-2847; from Honda Research; and from the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF-1231216. KA acknowledges support from NSERC. Any opinions, findings, and conclusions or recom-
mandations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. In Conference on Robot Learning, pages 357–368, 2017.

[2] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In Conference on Robot Learning, pages 651–673, 2018.

[3] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In Robotics and Automation (ICRA), 2017 IEEE International Conference on, pages 3389–3396. IEEE, 2017.

[4] Andy Zeng, Shuran Song, Stefan Velker, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Learning synergies between pushing and grasping with self-supervised deep reinforcement learning. arXiv preprint arXiv:1803.09956, 2018.

[5] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In Advances in Neural Information Processing Systems, pages 5048–5058, 2017.

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

[7] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. IEEE transactions on pattern analysis and machine intelligence, 37(2):408–423, 2015.

[8] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In International Conference on Machine Learning, pages 2829–2838, 2016.

[9] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In Advances in Neural Information Processing Systems, pages 2944–2952, 2015.

[10] Derrick H Nguyen and Bernard Widrow. Neural networks for self-learning control systems. IEEE Control systems magazine, 10(3):18–23, 1990.

[11] Igor Mordatch, Nikhil Mishra, Clemens Eppner, and Pieter Abbeel. Combining model-based policy search with online model learning for control of physical humanoids. In Robotics and Automation (ICRA), 2016 IEEE International Conference on, pages 242–248. IEEE, 2016.

[12] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. arXiv preprint arXiv:1805.12114, 2018.

[13] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 7559–7566. IEEE, 2018.

[14] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control. International Conference on Learning Representations, 2018.

[15] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In Proceedings of the fourteenth international conference on artificial intelligence and statistics, pages 627–635, 2011.

[16] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In Proceedings of the twenty-first international conference on Machine learning, page 1. ACM, 2004.

[17] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In AAAI, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[18] Zi Wang, Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Active model learning and diverse action sampling for task and motion planning. CoRR, abs/1803.00967, 2018.

[19] Alonso Marco, Felix Berkenkamp, Philipp Hennig, Angela P Schoellig, Andreas Krause, Stefan Schaal, and Sebastian Trimpe. Virtual vs. real: Trading off simulations and physical experiments in reinforcement learning with bayesian optimization. In Robotics and Automation (ICRA), 2017 IEEE International Conference on, pages 1557–1563. IEEE, 2017.

[20] Daniel J Lizotte, Tao Wang, Michael H Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression.

[21] Matthew Tesch, Jeff Schneider, and Howie Choset. Using response surfaces and expected improvement to optimize snake robot gait parameters. In Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, pages 1069–1074. IEEE, 2011.

[22] Alonso Marco, Philipp Hennig, Jeannette Bohg, Stefan Schaal, and Sebastian Trimpe. Automatic lqr tuning based on gaussian process global optimization. In Robotics and Automation (ICRA), 2016 IEEE International Conference on, pages 270–277. IEEE, 2016.

[23] Eric A Wan and Rudolph Van Der Merwe. The unscented kalman filter for nonlinear estimation. In Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000, pages 153–158. Ieee, 2000.
[24] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In Advances in Neural Information Processing Systems, pages 7176–7187, 2018.

[25] Svetoslav Kolev and Emanuel Todorov. Physically consistent state estimation and system identification for contacts. In Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on, pages 1036–1043. IEEE, 2015.

[26] Anurag Ajay, Jiajun Wu, Nima Fazeli, Maria Bauza, Leslie P. Kaelbling, Joshua B. Tenenbaum, and Alberto Rodriguez. Augmenting physical simulators with stochastic neural networks: Case study of planar pushing and bouncing, 2018.

[27] Alina Kloss, Stefan Schaal, and Jeannette Bohg. Combining learned and analytical models for predicting action effects. arXiv preprint arXiv:1710.04102, 2017.

[28] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control. arXiv preprint arXiv:1812.03201, 2018.

[29] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. 2016.

[30] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 5026–5033. IEEE, 2012.

[31] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. arXiv preprint arXiv:1802.09464, 2018.

[32] Fernando Fernández and Manuela Veloso. Probabilistic policy reuse in a reinforcement learning agent. In Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pages 720–727. ACM, 2006.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

VII. APPENDIX

All experiments in this paper use the following hyperparameters, which are mostly taken from [31].

- Actor and critic networks: 3 layers with 256 units each and ReLU non-linearities
- Adam optimizer [33] with $1 \cdot 10^{-3}$ for training both actor and critic
- Buffer size: $10^6$ transitions
- Polyak-averaging coefficient: 0.95
- Action L2 norm coefficient: 1.0
- Observation clipping: $[-200, 200]$,
- Batch size: 256
- Cycles per epoch: 50
- Batches per cycle: 40
- Test rollouts per epoch: 10
- Probability of random actions: 0.3
- Probability of HER experience replay: 0.8
- Normalized clipping: $[-5, 5]$.

Environment specific hyperparameters:

- Rollouts per MPI worker:
  - hook: 4
  - push (scratch), pick and place: 2
  - push (expert-explore, residuals): 4
- Number of MPI workers:
  - hook: 1
  - push, pick and place: 19
- Scale of additive Gaussian noise:
  - hook: 0.1
  - push (scratch [31]): 0.2
  - push (residuals+expert-explore): 0.1
  - pick and place (expert-explore): 0.1
  - pick and place (residuals): 0.001
  - pick and place (scratch [31]): 0.2

"Expert Explore" baseline hyperparameters (optimized with small grid search)

- $\epsilon$ 0.6 for hook environment
- $\epsilon$ 0.5 for push and pick and place environments
- $\alpha$ 0.8 for all environments

Thresholds:

- threshold 0.5 for push and pick and place environments
- threshold 0.7 for hook environment