Residual Policy Learning*

Antonia Bronars †
bronars@mit.edu

Rebecca Jiang †
rhjiang@mit.edu

Siddharth Nayak †
sidnayak@mit.edu

Abstract

This paper presents an investigation of residual policy learning (RPL) for simple manipulation tasks. We focus on planar sliding and pushing, pick-and-place, nut assembly, and peg-in-hole insertion. In such tasks, simple hand-designed controllers can make progress toward completing tasks, but often fail to precisely satisfy goals. While reinforcement learning can achieve consistent task completion, learning policies from scratch can be computationally expensive. We hand-design simple controllers with imperfect success rates for each of our tasks, and implement deep deterministic policy gradient (DDPG) with hindsight experience replay (HER) in the sparse reward setting case and Soft-Actor Critic (SAC) in the dense reward setting case to modify our simple controllers and achieve success rates closer to 1. We also learn a policy from scratch using DDPG and HER for each task in a sparse reward setting and compare results across methods. We find that learning an effective policy from residuals is faster than from scratch. We achieve high success rates for pushing and pick-and-place. For sliding, our only impulsive task, we never achieve consistent success from any policy. The code for this work can be found at https://github.com/nsidn98/Residual-Policy-Learning

1 Introduction

Solutions for simple manipulation tasks frequently have intuitive elements that can be incorporated into hand-designed controllers. A pick-and-place controller may locate a gripper around the object, close the gripper jaws, and move toward the goal location. A pushing controller may position a manipulator behind the object and move it in the direction of the goal. However, straightforward implementations of these intuitive solutions may be imperfect due to the various conditions that can arise. For instance, if an object being pushed deviates to the side of the manipulator, the manipulator should take corrective action to stabilize the push (a behavior that may not be encoded in a simple hand-designed controller).

Various works have approached robotic manipulation tasks [1, 2, 3, 4] and humanoid-based tasks [5, 6, 7, 8, 9] using reinforcement learning (RL). The main problem with learning policies using RL is the bad sample efficiency of the algorithms to learn good policies quickly and the deep neural networks used to parametrize the agents being data-hungry. In general reinforcement learning problems, during the exploration phase the agent takes random actions to explore the policy-space. In a sparse reward setting, reaching the desired goal with random exploration is quite rare and hence the number of iterations required to learn to complete the task is quite large. In some complex tasks, it is possible that the agent never learns to complete the task. This motivates the use of model-based control frameworks as baselines for the learning algorithm. In experimental settings, an additional drawback of using reinforcement learning alone is that random exploration of the action space on a real robot may not be safe. Although there are different exploration strategies in the RL literature where the agent is incentivized to explore novel regions in the observation space [10, 11, 12, 13, 14],

*6.881-Robotic Manipulation, Fall 2020, MIT. (LaTeXtemplate borrowed from NeuRIPS 2019)
†Equal Contribution

applying these models for robotic learning could be harmful where the exploratory behaviour of RL could be a safety concern. For these reasons, [15, 16] propose to learn residual policies.

A primary advantage of learning residuals over a conventional controller, compared to learning a policy from scratch is that, because the controller begins with prior information of useful (albeit suboptimal) actions, the agent is forced to visit states which are most relevant to the optimal solution. A random exploration in the policy-space in nearby regions of the controller actions can help to improve the sample efficiency of the RL agent to learn to complete the task especially in a sparse reward setting where the agent is rewarded only when it reaches the goal state. Another advantage is that the initial controller (which has been designed manually) would ensure the exploration of (relatively) safer regions in the policy-space.

One can think of RPL as biasing the exploration toward the state distribution of the controller. A reduction in the sample complexity in training can be explained by the reduction in exploration required to obtain the goal reward.

In this work, we use reinforcement learning over the residuals from basic hand-designed controllers for our tasks. In Section 3 we review the RL principles we use in our implementation. The RPL algorithm is defined in Section 4. In Section 5, we introduce the baseline controllers. Learning residuals from over imperfect baseline controllers enables significantly higher sample efficiency, as presented in Section 6. In addition to studying RPL we have also made small contributions to the robotics community by identifying and reporting bugs/issues in published tools and works. These are detailed in Appendix C.

2 Related Work

In [15] and [16], the authors propose to learn residual actions over conventional controllers to optimise policies. The authors demonstrate good performance with complex manipulation tasks where imperfect controllers are available. In [15], Silver et al. focus on a set of simulated manipulation tasks that highlight challenging conditions like noise and model misspecification. Our work is largely an implementation of the methods outlined in [15], featuring some of the same manipulation tasks. Johannink et al. [16] show simulated and experimental success in applying a controller using residuals found with reinforcement learning to a block assembly task. Zeng et al. [17] use deep networks to learn residuals on control parameters for the tossing task, resulting in 85% accuracy in tossing arbitrary objects into bins. This demonstrates that residual learning can be successful for impulsive tasks, in which the manipulator can only influence the object for a fraction of the episode over which the task is attempted.

There has also been work on using learning to improve physics models [18, 19]. This can be seen as a form of residual learning with a model-based controller. In contrast, our work focuses on learning residuals directly on the policy.

Schaff et al. [20] demonstrate an implementation of RPL that is algorithmically quite similar to the aforementioned work, but instead adjusts a human's actions in controlling a flight simulator (among other examples). This demonstrates the breadth of applicabilty of RPL approaches, from fully autonomous systems to assistive technology.

3 Preliminaries and Background

3.1 Reinforcement Learning

Reinforcement Learning (RL) tackles sequential decision-making problems with the goal of maximising the expected rewards. An agent learns how to make optimal actions by interacting with the environment and learning from trial and error. The stochastic process that emerges from the interaction with the environment is modelled as a Markov Decision Process (MDP) [21]. The MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the environment's state space, $\mathcal{A}$ is the agent's action space, $\mathcal{T}$ is environment dynamics also called as the transition function, $\mathcal{R}$ is the reward function, defining how the agent is rewarded for its actions and $\gamma \in [0, 1]$ is a factor used to discount rewards over time. At any given timestep the agent observes the state of the environment, $s \in \mathcal{S}$ and chooses an action $a \in \mathcal{A}$ based on the policy $\pi(s|\theta) : \mathcal{S} \rightarrow \mathcal{A}$ where θ contains the parameters of the policy. After taking the action, the agent moves on to the next state $s' \in \mathcal{S}$ according to the transition function

$s' = \mathcal{T}(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. Simultaneously, the agent receives a reward $r = \mathcal{R}(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. The aim of the agent is to maximise the expected return $J(\theta_\pi) = \mathbb{E}_{s_t, a_t \sim \pi} \sum_t \gamma^{t-1} \mathcal{R}(s_t, a_t)$ where the subscript t indicates the time step.

3.2 Deep Deterministic Policy Gradient

Policy Gradient algorithms maximise the expected return $J(\theta_\pi)$ by updating θ_π in the direction of $\nabla_{\theta_\pi} J(\theta_\pi)$. In particular, Deep Deterministic Policy Gradient (DDPG) [22] learns deterministic policies where deep artificial neural networks are used as function approximators. DDPG belongs to a class of algorithms called the actor-critic methods [23] where there are two networks: actor and critic. The critic learns the action-value function and the actor updates the policy in a direction as suggested by the critic. The actor (policy) network $a = \pi(s|\theta_\pi)$ maps states to deterministic actions and the critic (action-value) network $Q^\pi(s, a|\theta_{Q^\pi})$ where θ_{Q^π} contains the critic parameters returns an estimate of the total expected returns starting from state s and by taking an action a and then following the policy π. The algorithm alternates between two stages. First it collects experience using the current policy with an additional noise sampled from a random process $\mathcal{N}$ for random exploration, i.e. $a = \pi(s|\theta_\pi) + \mathcal{N}$. The transitions experienced $\langle s, a, r, s' \rangle$ are stored in a replay buffer $\mathcal{D}$. These transitions are later sampled randomly to learn the actor and critic networks. The critic is learnt by minimising the following loss to satisfy the Bellman equation:

$$\mathcal{L}(\theta_{\mathcal{Q}^\pi}) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \big[(Q^\pi(s, a|\theta_{Q^\pi}) - y)^2 \big] \tag{1}$$

$$y = r + \gamma Q^\pi(s', \pi(s')|\theta_{Q^\pi}) \tag{2}$$

In practice, minimising Eq 1 is numerically unstable since the target value is a function of Q^π, and is therefore updated in each iteration as well. Similarly to DQN [24], DDPG stabilises the learning by obtaining smoother target values $y = r + \gamma Q^{\pi'}(s', \pi'(s')|\theta_{Q^{\pi'}})$, where π' and $Q^{\pi'}$ are target networks. The weights of π' and $Q^{\pi'}$ are exponential moving averages of the weights of π and Q^π over iterations, respectively. The actor is updated using the following policy gradient:

$$\nabla_{\theta_\pi} J(\theta_\pi) = \mathbb{E}_{s \sim \mathcal{D}} \big[\nabla_a Q^\pi(s, \pi(s)|\theta_{Q^\pi}) \nabla_{\theta_\pi} \pi(s|\theta_\pi) \big] \tag{3}$$

Intuitively, this updates the actor network parameters in the direction which maximizes the discounted sum of future rewards.

3.3 Hindsight Experience Replay

Hindsight Experience Replay (HER) [1] was introduced to learn policies from sparse rewards, especially for robot manipulation tasks. The main idea of the algorithm is to view the states achieved in an episode as pseudo goals (achieved goals) to facilitate learning even when the desired goal has not been achieved during the episode. Specifically, we let $\langle s||g, a, r, s'||g \rangle$ be the original transition obtained in a rollout of an episode, where $||$ denotes the concatenation operation and g is the desired goal state of the task. In normal cases, the agent would be rewarded only when g is achieved, which may occur very rarely during learning, especially when the policy is far from optimal in the initial phases of the learning phase (random exploration). In HER, g is replaced by an achieved goal state g' which is randomly sampled from the states reached in an episode. This generates a new transition $\langle s||g', a, r, s'||g' \rangle$ which is more likely to be rewarded. The generated transitions are saved into an experience replay buffer and can be used by off-policy algorithms like DQN [24] and DDPG [22]. In essence, HER allows to learn from failures (not reaching the desired goals) and leverage them to ultimately learn to achieve the desired goal when $g' \rightarrow g$.

3.4 Soft Actor Critic

Soft Actor Critic (SAC) [7] is an actor-critic type algorithm which optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimizations and DDPG-style algorithms. SAC uses entropy regularization where entropy is a measure of randomness in the policy. The policy is trained to maximize the expected return and entropy. The entropy term has a relation to the exploration of the agent and hence increasing the entropy results in more exploration. Just like DDPG, the algorithm alternates between two stages. SAC has one policy network π_θ and two critic networks Q_{ϕ_1}, Q_{ϕ_2}. First it collects experience using the current policy π and the transitions

$\langle s, a, r, s' \rangle$ are stored in a replay buffer $\mathcal{D}$. Later these transitions are randomly sampled to learn the actor and critic networks. The critic is learnt by minimising the following loss:

$$\mathcal{L}(\phi_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}}\big[(Q(s, a|\phi_i) - y)^2\big] \quad \text{for } i = 1, 2 \tag{4}$$

$$y = r + \gamma\big(\min_{i=1,2} Q(s', \bar{a}'|\phi_i) - \alpha \log \pi_\theta(\bar{a}'|s')\big), \quad \bar{a}' \sim \pi_\theta(.|s') \tag{5}$$

To stabilise the training, target networks are used to calculate the target y instead of the current networks. $y = r + \gamma\big(\min_{i=1,2} Q(s', \bar{a}'|\phi'_i) - \alpha \log \pi_\theta(\bar{a}'|s')\big)$. The actor is updated by one step of gradient ascent using:

$$\nabla_{\theta_\pi} \mathbb{E}_{s \sim \mathcal{D}}\big[\min_{i=1,2} Q(s, \bar{a}'_\theta(s)|\phi_i) - \alpha \log \pi_\theta(\bar{a}'_\theta(s)|s))\big], \quad \bar{a}'_\theta(s) \sim \pi_\theta(')|s) \tag{6}$$

4 Residual Policy Learning

The algorithm underlying RPL is no different from any other RL algorithm. The key insight comes from the observation that the baseline policy (in our example, a simple hand-designed state machine controller) is not parameterized by the deep network, so the gradient of the policy (the sum of the baseline and the residual policy) with respect to the network parameters is just the gradient of the residual. The policy is given by:

$$\Pi_\theta(s) = \pi_\theta(s) + f(s) \tag{7}$$

where $\Pi_\theta(s)$ is the policy, $f(s)$ is the baseline controller, and $\pi_\theta(s)$ is the residual we aim to learn. As described in [15], we can think of the policy as a residual MDP, $M^{(\Pi)} = (S, A, R, T^{(\Pi)}, \gamma)$, where $T^{(\Pi)}(s, a, s') = T(s, f(s) + \pi_\theta(s), s')$. Therefore, the policy can be learnt simply by using standard DDPG + HER, as described in the preceding sections.

Despite using an identical algorithm, the RPL framework converges more quickly than learning from scratch. DDPG uses stochastic policy gradients to update the network parameters in the direction which maximizes reward. In addition to the actor network, a critic network is trained as a variance-reducing baseline that ensures, in essence, that each update to the network parameters improves policy performance with respect to the reward function. The critic is initialized with respect to the baseline policy through training for a "burn-in" period on the baseline (the action policy is not modified during this time). Equating the critic with the baseline policy through the use of a burn in period effectively means that parameter updates will be in the direction that improves upon the baseline's performance. This is a powerful variance-reduction technique, which explains the faster convergence for residual policies when compared with learning from scratch.

4.1 Initializing the Residual

One desirable property expected from RPL is that it should not make good policies worse. In essence, when the initial policy is perfect, we would like to have the residual policy to not have any influence on the action taken in the environment. ie. $\pi_\theta(s) = \vec{0}, \forall s \in \mathcal{S}$. Therefore as done in [15], the final layer of the actor is initialised with zero weights.

4.2 RPL with Actor-Critic Methods

RPL learns a residual over the output of the baseline controller. Actor-critic methods like DDPG, involve a policy (actor) as well as action-value function (critic). Thus if we initially have a perfect controller and a poor critic, the policy performance may degrade as the actor is trained in conjunction with the critic. Therefore, Silver et al. [15] propose to train the critic alone for a "burn-in" period by leaving the actor fixed. The burn-in length is determined automatically by monitoring the critic loss function. Although the paper suggests to start training the actor once the critic loss is below a threshold, we found out that this is not the case in their implementation code (more details in Appendix C.1). Instead we choose to monitor critic loss and start the actor training once the difference in the critic loss between successive epochs is less than β. We use $\beta = 0.005$ across all experiments.

5 Tasks and Basic Controllers

5.1 Slide Task

The goal of the slide task is to slide a cylindrical puck on a frictional table from a randomized initial position on the table to a randomized goal position on the table. The slide task is the only impulsive task considered in this work. That is, the hand-designed controller pushes the puck for a small fraction of the distance from its initial state to the goal state, then allows the puck to slide alone. The slide environment is shown in Figure 2.

5.1.1 FetchSlideSlap

The *FetchSlideSlap* controller assumes that the object instantaneously takes on the velocity of the end effector at contact (elastic collision, where the puck and the end effector are of the same mass), and the object slides under the influence of kinetic friction to its goal. See implementation details in Appendix B.1.

5.1.2 FetchSlideFriction

The *FetchSlideFriction* controller assumes that the object slides with the end effector for a short stretch (1/5 of the distance between the object's initial position and the goal), giving the object the opportunity to take on the velocity of the end effector over a distance of constant acceleration. The object then slides under the influence of kinetic friction to its goal. See Appendix B.2 for equations used for the motion of this controller.

The idea is to improve controller accuracy by eliminating the reliance on a sloppy impulse model (such as the one used in *FetchSlideSlap*). In practice, however, the slap controller performs better than the friction controller. This is because for some initial conditions, the end effector makes contact with the object along an edge, causing the object to slide relative to the end effector and get released at a slower velocity than intended. This unstable behavior during pushing is a drawback of controlling only the end effector position and not orientation, which is a feature of the OpenAI environment we chose to use. While it is possible to stabilize this unstable behavior with a more sophisticated baseline controller, our intention was to write simple controllers by hand and allow RPL to learn to correct the policies.

5.2 Pick-and-place Task

The goal of the pick-and-place task is to bring a cube object from a random initial position on a table to a random goal position in three dimensions. The pick-and-place environment is shown in Figure 4.

5.2.1 FetchPickAndPlacePerfect

FetchPickAndPlacePerfect uses a proportional controller to move the gripper toward the object until it is within a threshold distance. *FetchPickAndPlacePerfect* then opens the gripper, lowers the gripper within a height threshold, and closes the gripper. Finally, *FetchPickAndPlacePerfect* uses a proportional controller to move the gripper with the object to the goal position. Because this controller is executed in (an almost) closed-loop fashion with perfect state information, it is virtually always successful. Note: the state of whether the object has been picked up or not is not checked in a closed-loop fashion. Rather it just executes the grabbing action for one time-step.

5.2.2 FetchPickAndPlaceSticky

FetchPickAndPlaceSticky is similar to *FetchPickAndPlacePerfect* but takes the same action as in the previous time-step with a probability of 0.5. This was implemented for the purpose of introducing imperfection to the controller so that the qualities of the residual learning could be studied.

5.3 Push Task

The push task is similar to the slide task, but the end effector slides with the object through the duration of the episode. The push environment is shown in Figure 3.

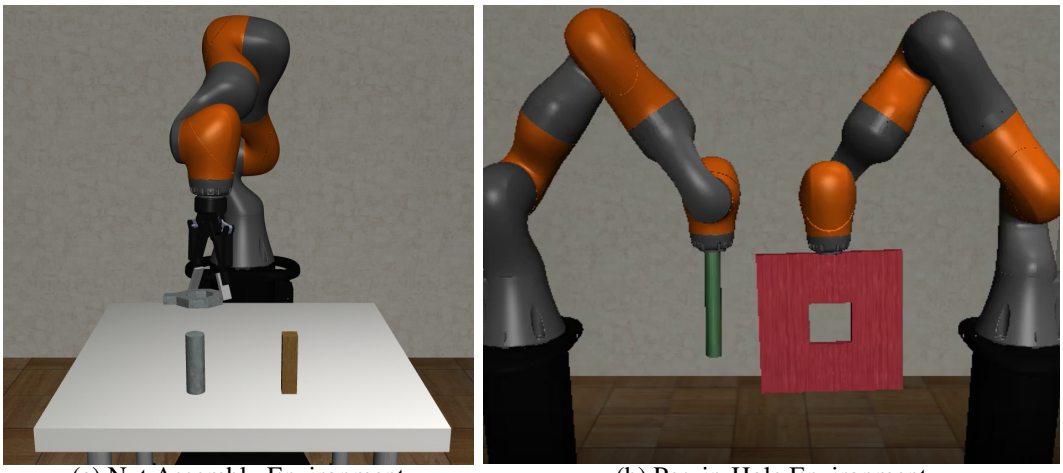
5.3.1 FetchPushImperfect

At each time step, the end effector moves the object in the direction of the goal with a small displacement proportional to the distance remaining to the goal. The commanded displacement is sufficiently small, and the goal within the workspace of the robot, such that the end effector remains in contact with the puck as it guides the puck to its goal position.

5.3.2 FetchPushSlippery

This implementation uses the same controller as in *FetchPushImperfect*, but the friction coefficient between the table and the puck is changed from 1.0 to 0.1. This causes the commanded displacement to move the puck by a larger amount at each timestep, occasionally resulting in loss of contact between the end effector and the puck between actions.

5.4 Nut assembly Task



(a) Nut Assembly Environment (b) Peg-in-Hole Environment

Figure 1: Visualizations of the two robosuite environments, Nut Assembly and Peg-in-Hole, we used to experiment with pose controllers.

The nut assembly task drops an octagonal nut (with a tab on one face) into the workspace at a random position and orientation (orientation only varies about world z, since the nut is constrained to land on the table). The robot arm must then pick up the nut and drop it onto a corresponding peg (see Figure 1a). The controller we implement is a simple proportional controller in pose space + state machine which commands the robot to move until it is sufficiently close in x and y to the nut, then reorient the gripper such that it will have an antipodal grasp on the nut, then pick up the nut. Finally, the robot carries the nut to the peg and drops it onto the peg once it is sufficiently close the the peg in x and y (see Appendix B.3 for implementation details). The baseline controller achieves a roughly 60% success rate at the nut assembly task. The primary failure modes we noted were from the gripper grasping the nut too quickly (and therefore the nut would slip or snap out of the gripper in the process of being picked up), or getting lodged on the peg after being dropped onto it. Both of these failures could be mitigated through more careful tuning of gains/thresholds for the state machine, but we leaned on the residual network to make those adjustments for us.

5.5 Peg-In-Hole

The peg-in-hole task begins with two robot arms separately holding a cylindrical peg and a square panel with a hole, as shown in Figure 1b. The robot arms must be controlled such that the peg enters the hole. We implemented trajectory optimization to plan three keyframe poses for each of the robots to pass through, where the keypoints satisfy the following requirements (respectively):

1. The peg is sufficiently in far in front of the panel (distance is greater than peg length).

2. The peg distance from panel is still greater than the peg length. The peg is orthogonal to panel. The peg is centered over hole.
3. The peg is inside the hole.

We use the sum of the magnitudes of dispacements between keyframes as a cost, including the movement from the initial position to the first keyframe. While we do not enforce in the optimization that the keyframes are within the robot workspaces, the cost encourages them to stay close to the initial position which in practice usually results in the keyframes found being within the workspace. A detailed description of the optimization problem that is posed and solved in this controller is included in Appendix B.4.

Ultimately we were unable to implement and simulate this controller because of a bug in Robosuite that prevents execution of absolute poses that are commanded. Appendix C.3 details the discovery of this bug. We also attempted to control to the desired poses via small "control deltas," or relative pose commands, which are properly executed in Robosuite. However, this approach was not successful because, without solving the full constrained trajectory planning problem, there was no guarantee that the path chosen for the incremental movements between keyframe poses would be feasible. Appendix B.4 contains a detailed description of a method we attempted for breaking the reorientation between keyframes into control deltas.

6 Results

For each of the tasks with which we performed RPL (slide, push, and pick-and-place), we present success rate as a function of number of simulation steps for:

- Baseline controllers (hand-designed, with constant success rate)
- RPL over each baseline controller
- A policy learned from scratch (DDPG + HER)

All empirical results are presented with median and one standard error across five different seeds. Although we did attempt to learn policies from scratch and learn residues over controllers for the nut assembly task and the peg-in-hole task as mentioned earlier, we do not have the plots ready for those experiments as they are still in progress (training not yet completed).

For push, slide and pick-and-place tasks, we see that, as expected, learning from scratch starts with a success rate near 0, and learns more slowly than does RPL. Success rates for all learning-based policies eventually converge near 1 for push and pick-and-place, but below 0.6 for slide. As noted in Section 5.1, the slide task is fundamentally different from the other tasks because it is impulsive; the end effector influences the object for a small fraction of the episode.

The RPL success rate shows an initial dip in performance once the burn-in of the critic is done. This same phenomenon is seen and discussed by Silver et al. [15]. Silver et al. explain this dip by saying that the critic is initialized poorly with respect to the baseline policy, and may degrade the policy once the actor starts learning. As discussed in Section 4, this is the purpose of the burn-in period. However, even after the burn-in period, the critic still causes the policy to degrade before correcting and compensating. Some hyperparameter tuning for the burn-in parameter β may alleviate the extent of the initial dip, but since (as noted by Silver et al. [15]) this initial dip in success rate does not significantly delay success rate convergence, we did not pursue this line of inquiry. While Silver et al. have proposed this reason for the initial policy dip, we believe the dip could be caused by a sudden change in the actor's learning rate from 0 to 0.001 at the end of the burn-in period. Immediately following this change, the actor samples actions far from the baseline controller, and as a result sees degraded performance. Because the replay buffer still contains good state transitions, the actor is able to recover quickly.

6.1 Slide Task

Figure 2 shows results for the slide task, using the *FetchSlideSlap* and *FetchSlideFriction* controllers as baselines. As discussed in Section 5.1, *FetchSlideSlap* provides a more successful baseline. However, the RPL success rates over these baselines converge at approximately the same rate following the

initial dip. The slide task may be relatively less successful than the other two tasks because it is an impulsive task in a sparse reward setting. This means that the success rate suffers greatly from imperfect action choices, because the opportunity for feedback is so limited. If dense rewards were used instead, we might see higher success rates as a result of modifying the definition of success: in a dense reward setting, slides that got close to the target position would be rewarded to some extent. This would benefit the other two tasks comparitively less. Since they are feedback policies, they have the opportunity to correct imperfect actions in subsequent timesteps.

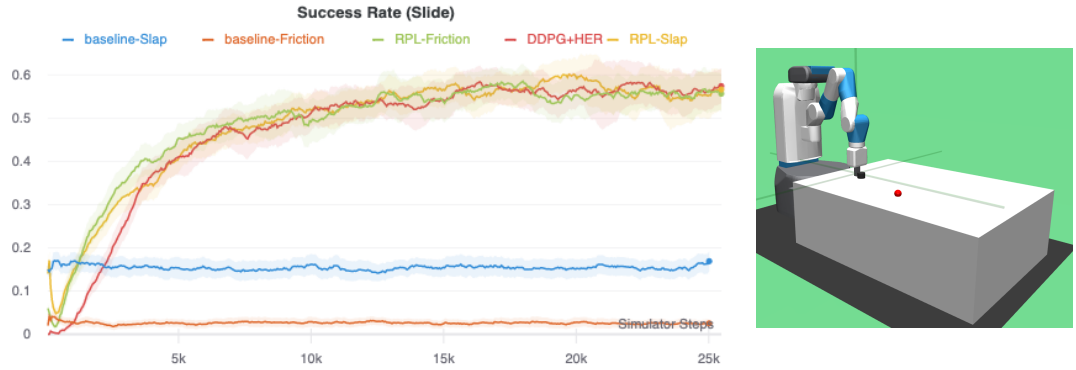


Figure 2: Residual Learning results (left) for the Slide task (visualized at right).

6.2 Push Task

Figure 3 shows results for the push task, using *FetchPushImperfect* and *FetchPushSlippery* as baselines. The success rates from the two baselines start near each other around 0.8. The RPL over *FetchPushSlippery* shows a greater initial dip, and subsequently takes slightly longer to reach success rates near 1. Compared to the slide task, we see a larger delay in the DDPG+HER (from-scratch) policy catching up in performance, likely because the baseline policies were more successful in the push task.

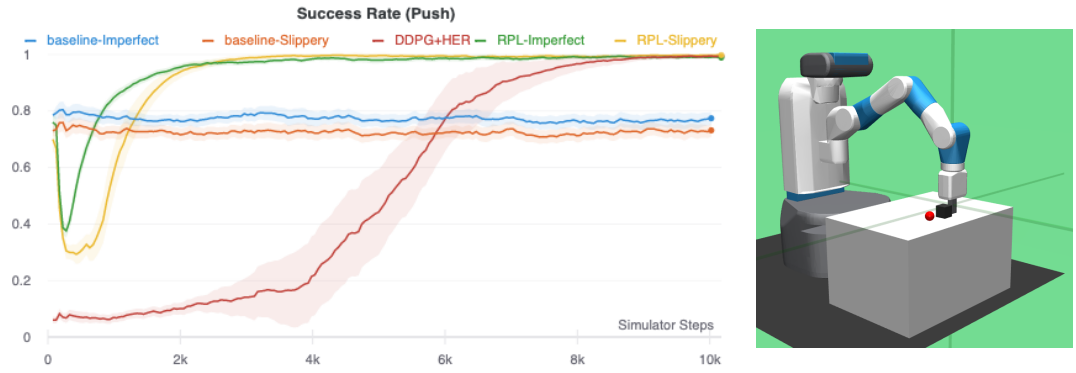


Figure 3: Residual Learning results (left) for the Push task (visualized at right).

6.3 Pick and Place Task

Figure 4 shows results for the pick-and-place task, using *FetchPickAndPlacePerfect* and *FetchPickAndPlaceSticky* as baselines. RPL over *FetchPickAndPlacePerfect*, which starts with a near-perfect success rate, sees a much larger dip than does RPL over *FetchPickAndPlaceSticky*, resulting in RPL over *FetchPickAndPlacePerfect* regaining high success rates only slightly faster than RPL over *FetchPickAndPlaceSticky*. In pick-and-place, compared to push, the learning-based policies do not quite converge to perfect success rates. They converge slightly below the success rate of the *FetchPickAndPlacePerfect* baseline controller; the initial "perfect" policy has been forgotten. This is an extension of the imperfect critic phenomena we have already discussed. If the critic were

initialized perfectly, or fully "burned in" before the actor started learning, then the critic would know that the initial policy was already perfect. Instead, RPL adds small residuals to the baseline policy. Unlike in the push task, completion of the pick-and-place task relies on a binary element: whether or not the object is grabbed. While the critic in the push task is similarly imperfect, small changes in the policy lead to small changes in the object trajectory, which can still lead to goal satisfaction. But in the pick and place task, failing to pick up the object will never lead to goal satisfaction. If the gripper is not aligned perfectly with the object when attempting to close the gripper, the grasp may fail. Additionally, the action used to define the gripper state is a number between -1 and 1. In the baseline controller we always used a value of 1 for open gripper states, but if RPL reduces this number slightly, the gripper may not open wide enough to fit around the object.

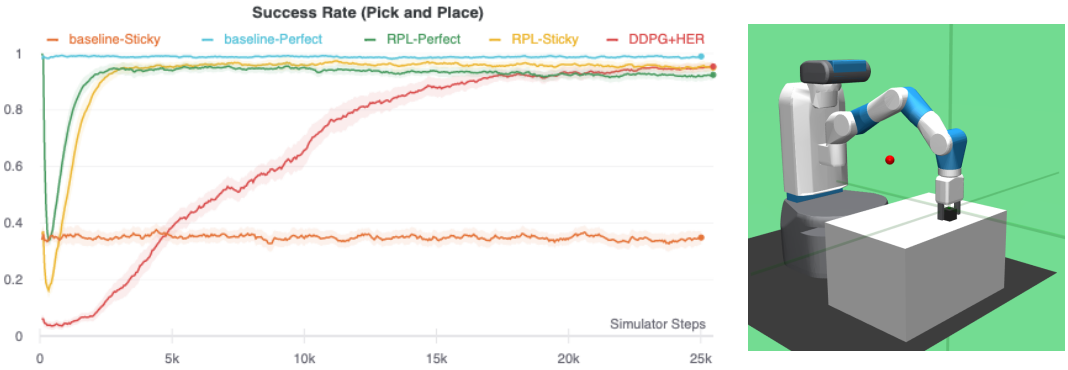


Figure 4: Residual Learning results (left) for the Pick and Place task (visualized at right).

Ultimately, our results for the Nut Assembly and Peg-in-Hole environments are incomplete for two reasons. Our implementation of SAC + dense rewards on these more complicated tasks suffered as a result of having a larger action space (controlling orientations as well as positions) and longer time horizon (500 steps per episode, rather than 100). The convergence rate of policy gradient methods is a counting game, and is dependent on the variance in the gradient. Although good baselines from analytical controllers can help drive variance down, doubling the action space and increasing the time horizon by a factor of 5 increases variance significantly, and we have not been able to get the policy for the Nut Assembly to converge yet. Our attempt to use trajectory optimization to plan keyframes to solve the Peg-in-Hole task was halted by a bug in robosuite (see Section 5.5 for details).

7 Future Work

As noted throughout this work, our hand-designed controllers were frequently limited by the simulation environments we chose to work in. Being unable to command end effector orientations in OpenAI, and being unable to command absolute poses in Robosuite, led to some loose ends in our work that could be tied up to lead to better results. In particular, the peg-in-hole task presented in Section 5.5, may have been an interesting demonstration of motion planning if we had been able to properly command absolute poses in Robosuite. Learning residuals on this task would be an interesting study on how RPL interacts with an optimization-based baseline controller, how RPL handles complex tasks that need to be done in a sequence of phases, and how RPL behaves with larger action spaces.

We have also posited that the reason the slide task never reached a high success rate with RPL is that, in contrast to our other tasks, this task is impulsive. However, Zeng et al. have shown successful use of RPL in tossing [17], which is also impulsive. A future work could investigate what factors influence whether RL is able to successfully learn residuals for an impulsive task.

Finally, methods like DDPG + HER are powerful in stochastic settings (real-world scenarios), where it is impossible to reproduce the same experiment twice. Good baselines (hand-designed controllers, critic networks) can reduce variance and lead to fast convergence, even under these conditions. Therefore, residual policy methods lend themselves naturally to experimentation and learning on real robot systems. Ultimately, we would like to try these methods on real robots, and put these stochastic gradient descent methods to the test in more random environments.

8 Acknowledgements

We would like to thank Tom Silver, one of the authors of the Residual Policy Learning paper for helping us understand the critic burn-in parameter in the paper. We would also like to thank MIT Supercloud [25] for providing us with compute resources to run our experiments. Finally, we would like to thank the 6.881 Fall 2020 course staff for helpful discussions and comments on our project scope and progress.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[2] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *CoRR*, abs/1806.10293, 2018.

[3] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

[4] A. Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. *CoRR*, abs/1809.07731, 2018.

[5] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[7] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *CoRR*, abs/1812.05905, 2018.

[8] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[9] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016.

[10] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[11] Justin Fu, John D. Co-Reyes, and Sergey Levine. EX2: exploration with exemplar models for deep reinforcement learning. *CoRR*, abs/1703.01260, 2017.

[12] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018.

[13] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. *CoRR*, abs/1605.09674, 2016.

[14] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. *CoRR*, abs/1611.04717, 2016.

[15] Tom Silver, Kelsey Allen, Josh Tenenbaum, and Leslie Kaelbling. Residual policy learning. 2018.

[16] 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. *CoRR*, abs/1812.03201, 2018.

[17] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. TossingBot: Learning to Throw Arbitrary Objects with Residual Physics. *arXiv e-prints*, page arXiv:1903.11239, March 2019.

[18] Alina Kloss, Stefan Schaal, and Jeannette Bohg. Combining Learned and Analytical Models for Predicting Action Effects from Sensory Data. *arXiv e-prints*, page arXiv:1710.04102, October 2017.

[19] 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. *arXiv e-prints*, page arXiv:1808.03246, August 2018.

[20] Charles Schaff and Matthew R. Walter. Residual policy learning for shared autonomy, 2020.

[21] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[22] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[23] Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Cambridge, MA, USA, 2002. AAI0804543.

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

[25] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[26] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[27] Verena Elisabeth Kremer. Quaternions and slerp, 2008.

A Implementational details

A.1 Model Hyperparameters for DDPG

Most of the hyperparameter values for the RPL experiments were taken from [15] with a few exceptions.

- Actor and critic networks: three layers with 256 units each and $ReLU$ nonlinearities and $tanh$ for the final layer of the actor
- Adam Optimizer [26] with learning rate 0.001 for both actor and critic
- Batch Size: 256
- Buffer Size: 10^6 transitions
- Polyak-averaging coefficient: 0.95
- Number of epochs: 500 (200 for push task)
- Cycles per epoch: 40
- Batches per cycle: 50
- Test rollouts per epoch: 50
- Probability of random actions: 0.3
- Scale of Gaussian noise: 0.2
- Observation Clipping: $[-200, 200]$
- Action L2 norm coefficient: 1.0
- Discount Factor γ: 0.98
- Burn-in parameter β: 0.005

A.2 Model Hyperparameters for SAC

Although our experiments with SAC were not completed before the deadline, we are listing the hyperparameters used for it:

- Actor and critic networks: two layers with 256 units each and $ReLU$ nonlinearities and $tanh$ for the final layer of the actor
- Adam Optimizer [26] with learning rate 0.0003 for both actor and critic
- Actor network type: Gaussian policy
- Actor Gaussian log standard deviation clipping: $[-20, 2]$
- Batch Size: 256
- Total number of simulator timesteps: 10^7
- Buffer Size: 10^6 transitions
- Number of simulator steps to start training: 10000
- Polyak-averaging coefficient: 0.995
- Evaluation frequency: 10 episodes
- Test rollouts per epoch: 10 (due to the long horizon of the task, we ran only 10 times for faster training)
- Discount Factor γ: 0.99
- Temperature parameter for entropy term α: 0.2
- Burn-in parameter β: 0.005

A.3 Environments

We implemented controllers for three Fetch environments from OpenAI (Slide, Push, and Pick and Place) and two environments from Robosuite (Nut Assembly and Peg-in-Hole). The details of the state and action spaces as well as the reward structure and the criteria for success for the Fetch and Robosuite environments are detailed below.

Fetch environments:

- States are length-31 vectors containing:
  - Gripper xyz position
  - Object xyz position
  - Object ypr orientation
  - Object relative position to gripper
  - Gripper finger joint state
  - Object velocity
  - Object rotational velocity
  - Gripper joint velocities
  - Gripper velocity
  - Achieved goal: current position of the object
  - Desired goal: goal position of the object
- Actions are length-4 vectors containing:
  - Gripper Displacement $[dx, dy, dz]$
  - Gripper state (single value in the range $[-1, 1]$ which specfies the degree to which the gripper is opened or closed)
- Rewards are sparse and binary: a reward of 0 is given when the object is within a small radius around the target location and -1 otherwise.
- Criteria for success: The episode is counted as a success if the last reward achieved in an episode is 0. Episode lengths are 50 and do not terminate early.

Robosuite Nut Assembly environment:

- States are length-20 vectors containing:
  - Robot end effector xyz position
  - Robot end effector orientation (quaternion)
  - Nut xyz position
  - Nut orientation (quaternion)
  - Achieved goal: current position of the nut
  - Desired goal: goal position for the nut (peg position)
- Action are length-7 vector containing:
  - End effector displacement $[dx, dy, dz]$
  - End effector orientation change (axis-angle rotation) $\delta[a_1, a_2, a_3]$

  - End-effector state (single value in the range $[-1, 1]$ which specifies the degree to which the gripper is opened or closed)
- Sparse reward setting (DDPG+HER): a reward of 0 is given when the object is within a small radius around the target location and -1 otherwise.
- Dense reward setting (SAC): Rewards are staged sequentially according to what stage the agent is in.
  - Reaching: $r_t \in [0, 0.1]$, proportional to the distance between the gripper and the nut.
  - Grasping: $r_t \in \{0, 0.35\}$, non-zero if the gripper is grasping the nut.
  - Lifting: $r_t \in \{0, [0.35, 0.5]\}$, non-zero only if nut is grasped; proportional to lifting height.
  - Hovering: $r_t \in \{0, [0.5, 0.7]\}$, non-zero only if nut is lifted; proportional to distance from nut to peg.
- Criteria for success: The episode is counted as a success if the last reward achieved in an episode is 0. Episode lengths are 500 and do not terminate early.

Robosuite Peg-in-Hole environment:

- States are length-14 vectors containing:
  - Robot 0 end effector xyz position
  - Robot 0 end effector orientation (quaternion)
  - Robot 1 end effector xyz position
  - Robot 1 end effector orientation (quaternion)
- Actions are length-12 vectors containing:
  - Robot 0 end effector xyz position
  - Robot 0 end effector orientation (axis-angle orientation) $\theta[a_1, a_2, a_3]$
  - Robot 1 end effector xyz position
  - Robot 1 end effector orientation (axis-angle orientation) $\theta[a_1, a_2, a_3]$
- Criteria for success: The episode is counted as a success if the parallel distance between the peg and hole is less than 0.06, the perpendicular distance between -0.12 and 0.14, and the cosine of the angle between the peg and hole greater than 0.95.

B Controllers

In this section we define the mathematical equations used to design our baseline controllers.

B.1 FetchSlideSlap controller

Define the distance from the initial puck position to the goal as d. To simplify the impulse model, we assume $v_{ee} = v_{puck}$ at the moment of impact. With that in mind, we solve for v_{ee} that will allow the puck to travel a distance d under the action of kinetic friction, before coming to rest. The kinetic energy of the puck at the start of the episode (equivalently, the kinetic energy of the end effector before impact) is equal to the work done by friction over the course of the slide:

$$\frac{1}{2} m v_{ee}^2 = \mu m g d$$

$$v_{ee} = \sqrt{2 \mu g d}$$

Since the slide environments take end effector displacements as actions, we prescribe that the end effector must accelerate constantly from a small distance $disp$ behind the end effector. From basic kinematics, we know:

$$a_{ee} = \frac{v_{ee}^2}{2 * disp}$$

At each timestep before the end effector makes contact with the puck (we track the position of the end effector relative to the goal, and once it is closer to the goal the puck initially was, we conclude that the end effector must have made contact with the puck), we command a displacement according to:

$$\Delta d = \frac{v_{t-1} + v_t}{2} \Delta t$$

Where v_{t-1} is the desired speed at the previous timestep, v_t is the the desired speed at the current timestep, and Δt is the time between actions (this value is a parameter of the environment, and in our case, is = 0.04 s).

B.2 FetchSlideFriction controller

Define the distance from the initial puck position to the goal as d, and let d_1 and d_2 be the distances the puck slides with and without the end effector respectively, such that $d_1 + d_2 = d$. The energy lost during the sliding period is $E_1 = d_2 \mu m g$ where μ is the coefficient of friction between the puck and the table, m is the puck mass, and g is gravity. To frictionally deplete this energy to reach speed $v_2 = 0$ at the goal state, the puck speed at the time of puck release must be $v_1 = \sqrt{2E_1/m} = \sqrt{2d_2\mu g}$.

Define the distance traveled along the vector from initial puck position to goal position as path parameter s. Starting at rest ($s_0 = \dot{s}_0 = 0$) and accelerating uniformly ($\ddot{s}$ constant) until the release state at $s_1 = d_1, \dot{s_1} = v_1$, we find

$$\ddot{s} = \frac{v_1^2}{2d_1} = \frac{d_2 \mu g}{d_1}$$

Because the actions in this task are applied as discrete displacements between controller calls, we must convert this controller to discrete displacements. Integrating once, we have

$$\dot{s} = \int \ddot{s}\, dt = \frac{d_2 \mu g}{d_1} t$$

Where $t = 0$ corresponds to the start of the push. To achieve this $\dot{s}$ value via a displacement over a timestep of size Δt, we need

$$\Delta s = \dot{s} \Delta t = \frac{d_2 \mu g}{d_1} t \Delta t$$

In simulation, we use $d_1 = d/5$.

B.3 Nut Assembly

The nut assembly task baseline controller is an operational space pose controller. The action and observation spaces are delineated in Appendix A.3. We use a proportional controller to drive the end effector pose to setpoints which serve as threshold values in a hand-designed state machine. The controller first aligns the position and orientation of the gripper to make antipodal contact with the nut, then lowers and closes the gripper, then pulls the gripper up and moves it, holding the nut, toward the peg. Finally, when the gripper is sufficiently close to the goal position, it drops the nut onto the peg. The details are listed below:

- Before grasp, align positions: at each timestep, check if the distance between the (x,y) position of the object and the (x,y) position of the gripper is within 0.01 m. If not, move in the direction of the (x,y) position of the object with gain 20.
- Before grasp, align orientations: The object is initialized with a random rotation about world z. We measure this rotation and, at each timestep, reorient the gripper with gain 0.2 until it has matched the object's rotation about world z, modulo $2\pi/8$ (gripping the object with no rotation about world z would lead to an antipodal grasp, and since the object is an octagon, there are other grasps which would be equally good). We track the change in orientation of the end effector by recording the intial orientation of the end effector, and calculating the relative quaternion (in the frame of the initial orientation of the end effector) at each timestep as ${}^A q^B = ({}^A q^W)({}^W q^B)$, where $({}^A q^W)$ is the inverse of the intial gripper pose, and $({}^W q^B)$ is the gripper pose measured at each timestep. We transform ${}^A q^B$ to axis-angle representation, then calculate the distance between the gripper axis-angle vector and the object axis-angle vector. Once the distance is less than 0.1, we conclude that the gripper and object are aligned.

- Close gripper: Lower the gripper as quickly as possible (command displacement 1 at each timestep) until the z position of the gripper frame is within 0.01 of the z position of the object frame. Then, close the gripper.
- Move gripper up and toward the goal: Lift the gripper 0.1m above the table as quickly as possible (command displacement 1 at each timestep), then move the gripper in the direction of the goal with gain 20 at each time step. Once the object frame is within 0.0225m in (x,y) of the goal frame, let go of the object and drop it onto the peg.

One of the benefits of RPL is that the various gains and thresholds of the hand-designed state machine do not need to be laboriously tuned to improve performance. Going from a position controller to a pose controller between the pick and place task and the nut assembly task is a small increase in problem complexity, but even this increase introduced a significant number of new parameters and state machine thresholds to tune at once. As problem complexity continues to increase, it becomes increasingly tedious and challenging to hand-design better performance; with RPL, it is reasonable to design a sloppy controller and expect the network to optimize its behavior for you.

B.4 Peg-in-hole

The optimization problem formulated to solve the peg-in-hole task is given below.

$$
\begin{aligned}
\underset{p_i^0, p_i^1, q_i^0, q_i^1, i=1,2,3}{\text{minimize}} \quad & \sum_i \left(||p_i^0 - p_{i-1}^0|| + ||p_i^1 - p_{i-1}^1|| \right) \\
\text{subject to} \quad & ||q_i^0|| = 1 \; \forall i \\
& ||q_i^1|| = 1 \; \forall i \\
& \left(q_1^1 p_1^0 q_1^{1-1} \cdot [0,0,1]^T \right) > l_{peg} \\
& \left(q_2^1 p_1^0 q_2^{1-1} \cdot [0,0,1]^T \right) > l_{peg} \\
& \left(q_2^1 p_1^0 q_2^{1-1} \cdot [1,0,0]^T \right) = 0 \\
& \left(q_2^1 p_1^0 q_2^{1-1} \cdot [0,1,0]^T \right) = 0 \\
& q_2^0 [0,0,1]^T q_2^{0-1} + q_2^1 [0,0,1]^T q_2^{1-1} = 0 \\
& \left(q_3^1 p_1^0 q_3^{1-1} \cdot [1,0,0]^T \right) = 0 \\
& \left(q_3^1 p_1^0 q_3^{1-1} \cdot [0,1,0]^T \right) = 0 \\
& q_3^0 [0,0,1]^T q_3^{0-1} + q_3^1 [0,0,1]^T q_3^{1-1} = 0 \\
& \left(q_3^1 p_1^0 q_3^{1-1} \cdot [0,0,1]^T \right) = l_{peg}/2
\end{aligned}
$$

Here, q_i^0 is the quaternion for the peg at keyframe i, and p_i^0 is the position of the peg at keyframe i. p_i^1 and q_i^1 correspond to the position and quaternion for the hole. The poses of the robots 0 and 1 can be directly calculated from these values because the transformation from the robot 0 and 1 end effectors to the peg and hole respectively are constant in the peg and hole frames respectively. l_{peg} is the length of the peg. The given constraints ensure that:

- All quaternions have unit magnitude
- The peg, represented in the hole frame, at keyframes 1 and 2, has z-component larger than l_{peg}.
- The peg, represented in the hole frame, at keyframes 2 and 3, has 0 x- and y-component (the peg is centered over the hole).
- The z directions of the hole and peg frames are antiparallel at keyframes 2 and 3 (the peg is aligned with the hole).
- The peg, represented in the hole frame at keyframe 3, has z-component equal to $l_{peg}/2$ (the peg has entered the hole).

Delta pose controller

Because we could not command absolute poses for the peg-in-hole task, we implemented a method to incrementally command interpolated delta poses to try to get to the absolute desired pose. For two

poses described by quaternions ${}^{W}q^{1}$ and ${}^{W}q^{2}$, representing transformations between the world frame W and poses 1 and 2, the delta rotation, or the transformation between poses 1 and 2, is given by

$$ {}^{1}q^{2} = {}^{1}\ q^{W}{}^{W}q^{2} = ({}^{W}q^{1})^{-1}{}^{W}q^{2} $$

${}^{1}q^{2}$ can be converted to an axis-angle representation, $\{a_{1,2},\ \theta_{1,2}\}$. To command an incremental rotation in the direction of the desired pose 2, we can scale $\theta_{1,2}$ by some factor δ: $\{a_{1,2},\ \delta\theta_{1,2}\}$. Here, $a_{1,2}$, which is found using a typical quaternion-to-axis-angle converter, will be represented in the frame associated with pose 1, but we want to command poses relative to the world frame, so the orientation we must command is

$$ \{{}^{W}q^{1}(a_{1,2})({}^{W}q^{1})^{-1},\ \delta\theta_{1,2}\} $$

The interpolation performed in this method (via scaling the angle in the axis-angle representation) is spherical linear interpolation (Slerp) [27]. Position changes can be interpolated simply and commanded incrementally as well. While this interpolation would work for small pose changes, we frequently needed to command large pose changes for the peg-in-hole task, and faced the problem that the intermediate/interpolated poses were not necessarily a feasible path by which to get to the goal pose.

C Bugs found

In this section we describe the different bugs we found *in other works* (not our own) while implementing this work. The value and importance of their work to our project should not be underestimated; we simply wanted to report these bugs in case others come across them, and explain how they modified our approach to the project.

C.1 Burn-in Parameter (β) in RPL

While we were trying to choose the value of the burn-in parameter β, we asked the authors of the Residual Policy Learning paper for some insights and they pointed us to their repository. There, we noticed that instead of monitoring the critic loss (as mentioned in their paper) for burning the critic they were monitoring the the difference in actor losses in successive epochs. To this end, we reported the authors regarding this and the authors agreed that it was a mistake on their side and opened an issue in their repository to warn others regarding this. Link to Issue #14 in the repository. In our experiments, because we monitor the difference in critic losses in successive epochs(instead of actor losses), we had to change the value of the burn-in parameter $\beta = 0.005$ from the value of $\beta = 1$ they used in the paper. This decrease in the value of β was partly because of the scale of actor losses was more than the critic losses.

C.2 Changes in friction parameters in robots

While we were trying to figure out how to change the friction coefficient parameters in MuJoCo, we referred the Residual Policy Learning authors' repository and found that they were changing the friction coefficient values between all the joints instead of just the friction coefficient values between the table and the puck (Permalink to the line). Although, this does not cause much of a difference in the training of the residual policies, we felt that it was important to point this out for anyone intending to work with MuJoCo as it can be easily overlooked while modifying the environments.

C.3 Robosuite absolute pose controller

One of the robot arm controllers available in Robosuite is *OSC_POSE*, which is meant to control the robot arm to an absolute position and orientation commanded. When trying to use *OSC_POSE*, we found erratic arm flailing. The Robosuite developers pushed an update to correct the behavior we reported, and while the updated version correctly positions the end effector at the commanded position, it does not match the commanded orientation. We reported this followup issue but did not receive a response. Link to Issue #139 in the Robosuite repository.