

Learning Dexterous In-Hand Manipulation

OpenAI* Marcin Andrychowicz, Bowen Baker, Maciek Chociej,
 Rafał Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron,
 Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor,
 Josh Tobin, Peter Welinder, Lilian Weng, Wojciech Zaremba

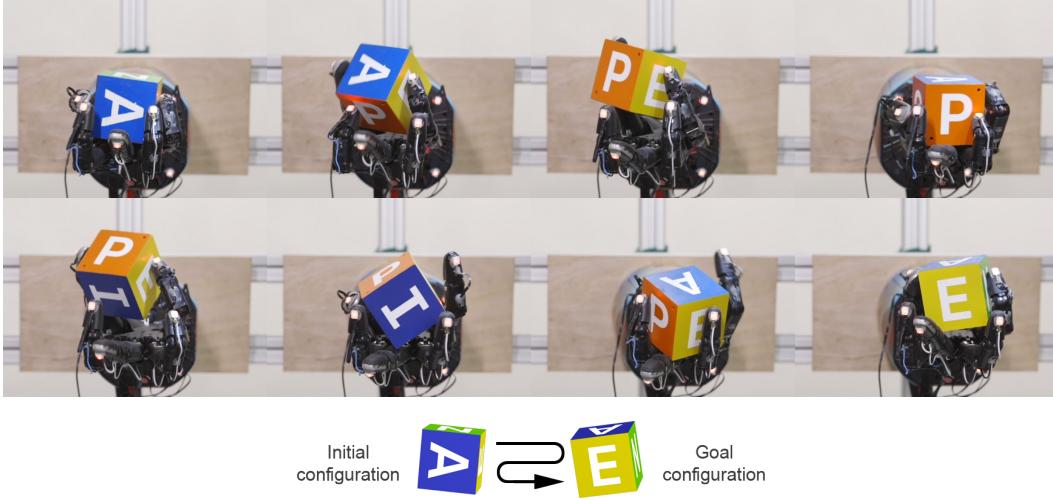


Figure 1: A five-fingered humanoid hand trained with reinforcement learning manipulating a block from an initial configuration to a goal configuration using vision for sensing.

Abstract

We use reinforcement learning (RL) to learn dexterous in-hand manipulation policies which can perform vision-based object reorientation on a physical Shadow Dexterous Hand. The training is performed in a simulated environment in which we randomize many of the physical properties of the system like friction coefficients and an object’s appearance. Our policies transfer to the physical robot despite being trained entirely in simulation. Our method does not rely on any human demonstrations, but many behaviors found in human manipulation emerge naturally, including finger gaiting, multi-finger coordination, and the controlled use of gravity. Our results were obtained using the same distributed RL system that was used to train OpenAI Five [43]. We also include a video of our results: <https://youtu.be/jwSbzNHGf1m>.

1 Introduction

While dexterous manipulation of objects is a fundamental everyday task for humans, it is still challenging for autonomous robots. Modern-day robots are typically designed for specific tasks in constrained settings and are largely unable to utilize complex end-effectors. In contrast, people are able to perform a wide range of dexterous manipulation tasks in a diverse set of environments, making the human hand a grounded source of inspiration for research into robotic manipulation.

*Please use the following bibtex for citation: <https://openai.com/bibtex/openai2018learning.bib>

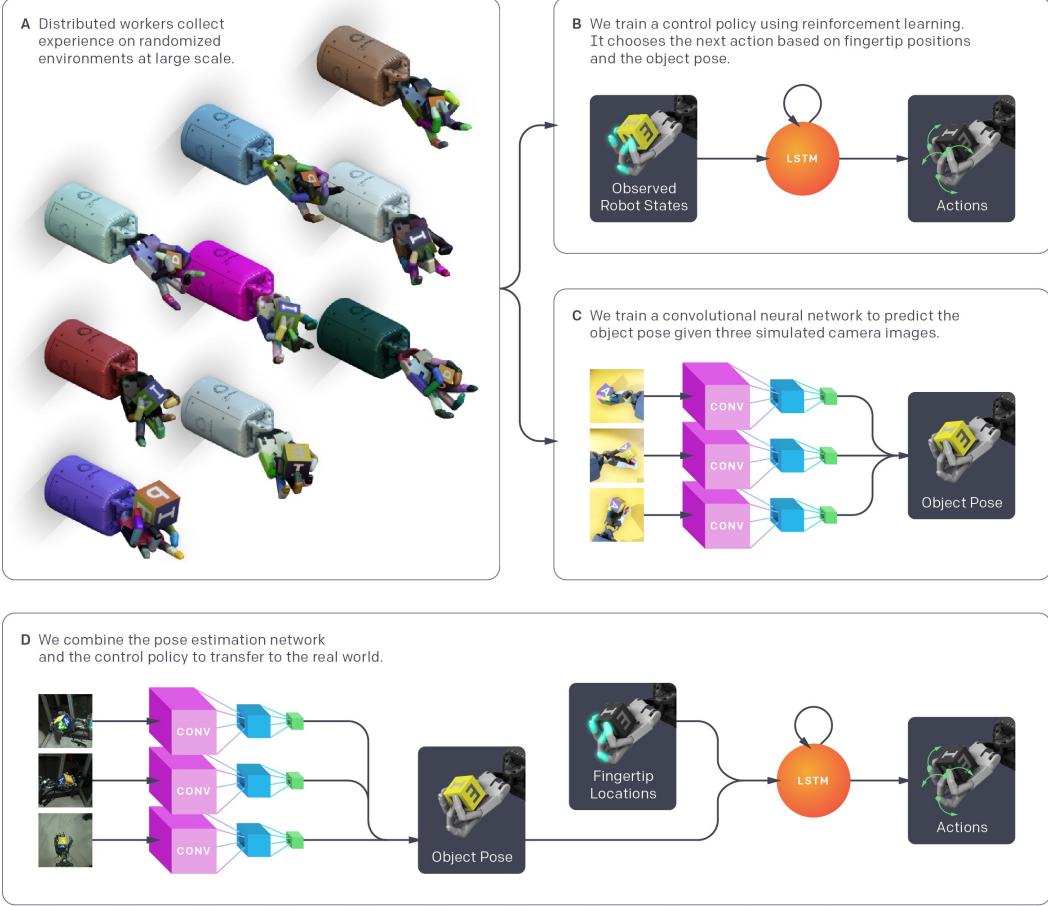


Figure 2: System Overview. (a) We use a large distribution of simulations with randomized parameters and appearances to collect data for both the control policy and vision-based pose estimator. (b) The control policy receives observed robot states and rewards from the distributed simulations and learns to map observations to actions using a recurrent neural network and reinforcement learning. (c) The vision based pose estimator renders scenes collected from the distributed simulations and learns to predict the pose of the object from images using a convolutional neural network (CNN), trained separately from the control policy. (d) To transfer to the real world, we predict the object pose from 3 real camera feeds with the CNN, measure the robot fingertip locations using a 3D motion capture system, and give both of these to the control policy to produce an action for the robot.

The Shadow Dexterous Hand [58] is an example of a robotic hand designed for human-level dexterity; it has five fingers with a total of 24 degrees of freedom. The hand has been commercially available since 2005; however it still has not seen widespread adoption, which can be attributed to the daunting difficulty of controlling systems of such complexity. The state-of-the-art in controlling five-fingered hands is severely limited. Some prior methods have shown promising in-hand manipulation results in simulation but do not attempt to transfer to a real world robot [5, 40]. Conversely, due to the difficulty in modeling such complex systems, there has also been work in approaches that only train on a physical robot [16, 67, 29, 30]. However, because physical trials are so slow and costly to run, the learned behaviors are very limited.

In this work, we demonstrate methods to train control policies that perform in-hand manipulation and deploy them on a physical robot. The resulting policy exhibits unprecedented levels of dexterity and naturally discovers grasp types found in humans, such as the tripod, prismatic, and tip pinch grasps, and displays contact-rich, dynamic behaviours such as finger gaiting, multi-finger coordination, the controlled use of gravity, and coordinated application of translational and torsional forces to the object. Our policy can also use vision to sense an object’s pose — an important aspect for robots that should ultimately work outside of a controlled lab setting.



Figure 3: (left) The "cage" which houses the robot hand, 16 PhaseSpace tracking cameras, and 3 Basler RGB cameras. (right) A rendering of the simulated environment.

Despite training entirely in a simulator which substantially differs from the real world, we obtain control policies which perform well on the physical robot. We attribute our transfer results to (1) extensive randomizations and added effects in the simulated environment alongside calibration, (2) memory augmented control policies which admit the possibility to learn adaptive behaviour and implicit system identification on the fly, and (3) training at large scale with distributed reinforcement learning. An overview of our approach is depicted in Figure 2.

The paper is structured as follows. Section 2 gives a system overview, describes the proposed task in more detail, and shows the hardware setup. Section 3 describes observations for the control policy, environment randomizations, and additional effects added to the simulator that make transfer possible. Section 4 outlines the control policy training procedure and the distributed RL system. Section 5 describes the vision model architecture and training procedure. Finally, Section 6 describes both qualitative and quantitative results from deploying the control policy and vision model on a physical robot.

2 Task and System Overview

In this work we consider the problem of in-hand object reorientation. We place the object under consideration onto the palm of a humanoid robot hand. The goal is to reorient the object to a desired target configuration in-hand. As soon as the current goal is (approximately) achieved, a new goal is provided until the object is eventually dropped. We use two different objects, a block and an octagonal prism. Figure 3 depicts our physical system as well as our simulated environment.

2.1 Hardware

We use the Shadow Dexterous Hand, which is a humanoid robotic hand with 24 degrees of freedom (DoF) actuated by 20 pairs of agonist–antagonist tendons. We use a PhaseSpace motion capture system to track the Cartesian position of all five finger tips. For the object pose, we have two setups: One that uses PhaseSpace markers to track the object and one that uses three Basler RGB cameras for vision-based pose estimation. This is because our goal is to eventually have a system that works outside of a lab environment, and vision-based systems are better equipped to handle the real world. We do not use the touch sensors embedded in the hand and only use joint sensing for implementing

low-level relative position control. We update the targets of the low level controller, which runs at roughly 1 kHz, with relative positions given by the control policy at roughly 12 Hz.

More details on our hardware setup are available in Appendix B.

2.2 Simulation

We simulate the physical system with the MuJoCo physics engine [64], and we use Unity² to render the images for training the vision based pose estimator. Our model of the Shadow Dexterous Hand is based on the one used in the OpenAI Gym robotics environments [49] but has been improved to match the physical system more closely through calibration (see Appendix C.3 for details).

Despite our calibration efforts, the simulation is still a rough approximation of the physical setup. For example, our model directly applies torque to joints instead of tendon-based actuation and uses rigid body contact models instead of deformable body contact models. Modeling these and other effects seen in the real world is difficult or impossible in a rigid body simulator. These differences cause a "reality gap" and make it unlikely for a policy trained in a simulation with these inaccuracies to transfer well.

We describe additional details of our simulation in Appendix C.1.

3 Transferable Simulations

As described in the previous section, our simulation is a coarse approximation of the real world. We therefore face a dilemma: we cannot train on the physical robot because deep reinforcement learning algorithms require millions of samples; conversely, training only in simulation results in policies that do no transfer well due to the gap between the simulated and real environments. To overcome the reality gap, we modify the basic version of our simulation to a *distribution over many simulations* that foster transfer [54, 62, 45]. By carefully selecting the sensing modalities and by randomizing most aspects of our simulated environment we are able to train policies that are less likely to overfit to a specific simulated environment and more likely to transfer successfully to the physical robot.

3.1 Observations

We give the control policy observations of the fingertips using PhaseSpace markers and the object pose either from PhaseSpace markers or the vision based pose estimator. Although the Shadow Dexterous Hand contains a broad array of built-in sensors, we specifically avoided providing these as observations to the policy because they are subject to state-dependent noise that would have been difficult to model in the simulator. For example, the fingertip tactile sensor measures the pressure of a fluid stored in a balloon inside the fingertip, which correlates with the force applied to the fingertip but also with a number of confounding variables, including atmospheric pressure, temperature, and the shape of the contact and intersection geometry. Although it is straightforward to determine the existence of contacts in the simulator, it would be difficult to model the distribution of sensor values. Similar considerations apply to the joint angles measured by Hall effect sensors, which are used by the low-level controllers but not provided to the policy due to their tendency to be noisy and hard to calibrate.

3.2 Randomizations

Following previous work on *domain randomization* [54, 62, 45], we randomize most of the aspects of the simulated environment in order to learn both a policy and a vision model that generalizes to reality. We briefly detail the types of randomizations below, and Appendix C.2 contains a more detailed discussion of the more involved randomizations and provides hyperparameters.

Observation noise. To better mimic the kind of noise we expect to experience in reality, we add Gaussian noise to policy observations. In particular, we apply a correlated noise which is sampled once per episode as well as an uncorrelated noise sampled at every timestep.

²Unity game engine website: <https://unity3d.com/>

Table 1: Ranges of physics parameter randomizations.

Parameter	Scaling factor range	Additive term range
object dimensions	uniform([0.95, 1.05])	
object and robot link masses	uniform([0.5, 1.5])	
surface friction coefficients	uniform([0.7, 1.3])	
robot joint damping coefficients	loguniform([0.3, 3.0])	
actuator force gains (P term)	loguniform([0.75, 1.5])	
joint limits		$\mathcal{N}(0, 0.15)$ rad
gravity vector (each coordinate)		$\mathcal{N}(0, 0.4)$ m/s ²

Physics randomizations. Physical parameters like friction are randomized at the beginning of every episode and held fixed. Many parameters are centered on values found during model calibration in an effort to make the simulation distribution match reality more closely. Table 1 lists all physics parameters that are randomized.

Unmodeled effects. The physical robot experiences many effects that are not modeled by our simulation. To account for imperfect actuation, we use a simple model of motor backlash and introduce action delays and action noise before applying them in simulation. Our motion capture setup sometimes loses track of a marker temporarily, which we model by freezing the position of a simulated marker with low probability for a short period of time in simulation. We also simulate marker occlusion by freezing its simulated position whenever it is close to another marker or the object. To handle additional unmodeled dynamics, we apply small random forces to the object. Details on the concrete implementation are available in Appendix C.2.

Visual appearance randomizations. We randomize the following aspects of the rendered scene: camera positions and intrinsics, lighting conditions, the pose of the hand and object, and the materials and textures for all objects in the scene. Figure 4 depicts some examples of these randomized environments. Details on the randomized properties and their ranges are available in Appendix C.2.



Figure 4: Simulations with different randomized visual appearances. Rows correspond to the renderings from the same camera, and columns correspond to renderings from 3 separate cameras which are simultaneously fed into the neural network.

4 Learning Control Policies From State

4.1 Policy Architecture

Many of the randomizations we employ persist across an episode, and thus it should be possible for a memory augmented policy to identify properties of the current environment and adapt its own behavior accordingly. For instance, initial steps of interaction with the environment can reveal the weight of the object or how fast the index finger can move. We therefore represent the policy as a recurrent neural network with memory, namely an LSTM [25] with an additional hidden layer with ReLU [41] activations inserted between inputs and the LSTM.

The policy is trained with Proximal Policy Optimization (PPO) [57]. We provide background on reinforcement learning and PPO in greater detail in Appendix A. PPO requires the training of two networks — a policy network, which maps observations to actions, and a value network, which predicts the discounted sum of future rewards starting from a given state. Both networks have the same architecture but have independent parameters. Since the value network is only used during training, we use an Asymmetric Actor-Critic [46] approach. Asymmetric Actor-Critic exploits the fact that the value network can have access to information that is not available on the real robot system.³ This potentially simplifies the problem of learning good value estimates since less information needs to be inferred. The list of inputs fed to both networks can be found in Table 2.

Table 2: Observations of the policy and value networks, respectively.

Input	Dimensionality	Policy network	Value network
fingertip positions	15D	✓	✓
object position	3D	✓	✓
object orientation	4D (quaternion)	✗ ⁴	✓
target orientation	4D (quaternion)	✗	✓
relative target orientation	4D (quaternion)	✓	✓
hand joints angles	24D	✗	✓
hand joints velocities	24D	✗	✓
object velocity	3D	✗	✓
object angular velocity	4D (quaternion)	✗	✓

4.2 Actions and Rewards

Policy actions correspond to desired joints angles relative to the current ones⁵ (e.g. rotate this joint by 10 degrees). While PPO can handle both continuous and discrete action spaces, we noticed that discrete action spaces work much better. This may be because a discrete probability distribution is more expressive than a multivariate Gaussian or because discretization of actions makes learning a good advantage function potentially simpler. We discretize each action coordinate into 11 bins.

The reward given at timestep t is $r_t = d_t - d_{t+1}$, where d_t and d_{t+1} are the rotation angles between the desired and current object orientations before and after the transition, respectively. We give an additional reward of 5 whenever a goal is achieved and a reward of -20 (a penalty) whenever the object is dropped. More information about the simulation environment can be found in Appendix C.1.

³This includes noiseless observation and additional observations like joint angles and angular velocities, which we cannot sense reliably but which are readily available in simulation during training.

⁴We accidentally did not include the current object orientation in the policy observations but found that it makes little difference since this information is indirectly available through the relative target orientation.

⁵The reason for using *relative* targets is that it is hard to precisely measure absolute joints angles on the physical robot. See Appendix B for more details.

4.3 Distributed Training with Rapid

We use the same distributed implementation of PPO that was used to train OpenAI Five [43] without any modifications. Overall, we found that PPO scales up easily and requires little hyperparameter tuning. The architecture of our distributed training system is depicted in Figure 5.

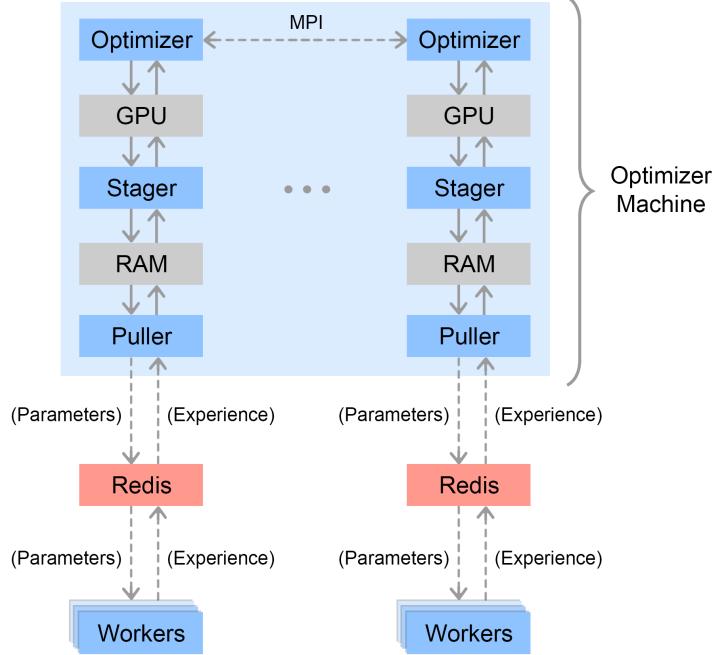


Figure 5: Our distributed training infrastructure in Rapid. Individual threads are depicted as blue squares. Worker machines randomly connect to a Redis server from which they pull new policy parameters and to which they send new experience. The optimizer machine has one MPI process for each GPU, each of which gets a dedicated Redis server. Each process has a *Puller* thread which pulls down new experience from Redis into a buffer. Each process also has a *Stager* thread which samples minibatches from the buffer and stages them on the GPU. Finally, each *Optimizer* thread uses a GPU to optimize over a minibatch after which gradients are accumulated across threads and new parameters are sent to the Redis servers.

In our implementation, a pool of 384 worker machines, each with 16 CPU cores, generate experience by rolling out the current version of the policy in a sample from distribution of randomized simulations. Workers download the newest policy parameters from the optimizer at the beginning of every epoch, generate training episodes, and send the generated episodes back to the optimizer. The communication between the optimizer and workers is implemented using the Redis in-memory data store. We use multiple Redis instances for load-balancing, and workers are assigned to an instance randomly. This setup allows us to generate about 2 years of simulated experience per hour.

The optimization is performed on a single machine with 8 GPUs. The optimizer threads pull down generated experience from Redis and then stage it to their respective GPU’s memory for processing. After computing gradients locally, they are averaged across all threads using MPI, which we then use to update the network parameters.

The hyperparameters that we used can be found in Appendix D.1.

5 State Estimation from Vision

The policy that we describe in the previous section takes the object’s position as input and requires a motion capture system for tracking the object on the physical robot. This is undesirable because tracking objects with such a system is only feasible in a lab setting where markers can be placed

on each object. Since our ultimate goal is to build robots for the real world that can interact with arbitrary objects, sensing them using vision is an important building block. In this work, we therefore wish to infer the object’s pose from vision alone. Similar to the policy, we train this estimator only on synthetic data coming from the simulator.

5.1 Model Architecture

To resolve ambiguities and to increase robustness, we use three RGB cameras mounted with differing viewpoints of the scene. The recorded images are passed through a convolutional neural network, which is depicted in Figure 6. The network predicts both the position and the orientation of the object. During execution of the control policy on the physical robot, we feed the pose estimator’s prediction into the policy, which in turn produces the next action.

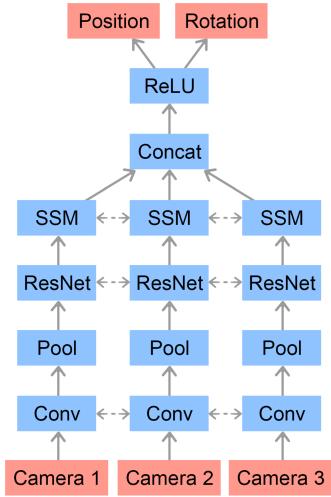


Figure 6: Vision network architecture. Camera images are passed through a convolutional feature stack that consists of two convolutional layers, max-pooling, 4 ResNet blocks [24], and spatial softmax (SSM) [19] layers with shared weights between the feature stacks for each camera. The resulting representations are flattened, concatenated, and fed to a fully connected network. All layers use ReLU [41] activation function. Linear outputs from the last layer form the estimates of the position and orientation of the object.

5.2 Training

We run the trained policy in the simulator until we gather one million states. We then train the vision network by minimizing the mean squared error between the normalized prediction and the ground-truth with minibatch gradient descent. For each minibatch, we render the images with randomized appearance before feeding them to the network. Moreover, we augment the data by modifying the object pose. We use 2 GPUs for rendering and 1 GPU for running the network and training.

Additional training details are available in Appendix D.2 and randomization details are in Appendix C.2.

6 Results

In this section, we evaluate the proposed system. We start by deploying the system on the physical robot, and evaluating its performance on in-hand manipulation of a block and an octagonal prism. We then focus on individual aspects of our system: We conduct an ablation study of the importance of randomizations and policies with memory capabilities in order to successfully transfer. Next, we consider the sample complexity of our proposed method. Finally, we investigate the performance of the proposed vision pose estimator and show that using only synthetic images is sufficient to achieve good performance.

6.1 Qualitative Results

During deployment on the robot as well as in simulation, we notice that our policies naturally exhibit many of the grasps found in humans (see Figure 7). Furthermore, the policy also naturally discovers many strategies for dexterous in-hand manipulation described by the robotics community [37] such as

finger pivoting, finger gaiting, multi-finger coordination, the controlled use of gravity, and coordinated application of translational and torsional forces to the object. It is important to note that we did not incentivize this directly: we do not use any human demonstrations and do not encode any prior into the reward function.

For precision grasps, our policy tends to use the little finger instead of the index or middle finger. This may be because the little finger of the Shadow Dexterous Hand has an extra degree of freedom compared to the index, middle and ring fingers, making it more dexterous. In humans the index and middle finger are typically more dexterous. This means that our system can rediscover grasps found in humans, but adapt them to better fit the limitations and abilities of its own body.

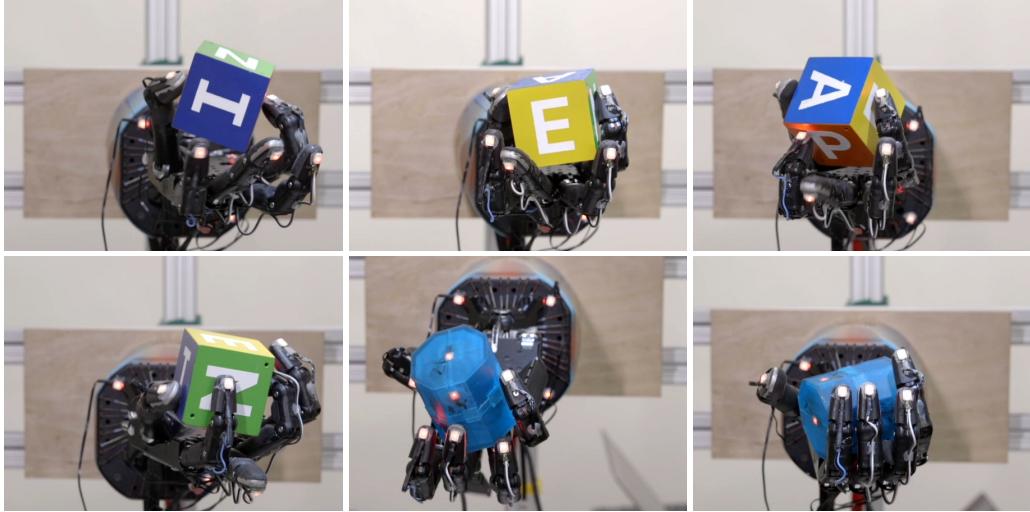


Figure 7: Different grasp types learned by our policy. From top left to bottom right: Tip Pinch grasp, Palmar Pinch grasp, Tripod grasp, Quadpod grasp, 5-Finger Precision grasp, and a Power grasp. Classified according to [18].

We observe another interesting parallel between humans and our policy in finger pivoting, which is a strategy in which an object is held between two fingers and rotate around this axis. It was found that young children have not yet fully developed their motor skills and therefore tend to rotate objects using the proximal or middle phalanges of a finger [44]. Only later in their lives do they switch to primarily using the distal phalanx, which is the dominant strategy found in adults. It is interesting that our policy also typically relies on the distal phalanx for finger pivoting.

During experiments on the physical robot we noticed that the most common failure mode was dropping the object while rotating the wrist pitch joint down. Moreover, the vertical joint was the most common source of robot breakages, probably because it handles the biggest load. Given these difficulties, we also trained a policy with the wrist pitch joint locked.⁶ We noticed that not only does this policy transfer better to the physical robot but it also seems to handle the object much more deliberately with many of the above grasps emerging frequently in this setting. Other failure modes that we observed were dropping the object shortly after the start of a trial (which may be explained by incorrectly identifying some aspect of the environment) and getting stuck because the edge of an object got caught in a screw hole (which we do not model).

We encourage the reader to watch the accompanying video to get a better sense of the learned behaviors.⁷

⁶We had trouble training in this environment from scratch, so we fine-tuned a policy trained in the original environment instead.

⁷Real-time video of 50 successful consecutive rotations: <https://youtu.be/DKe8FumoD4E>

6.2 Quantitative Results

In this section we evaluate our results quantitatively. To do so, we measure the number of *consecutive* successful rotations until the object is either dropped, a goal has not been achieved within 80 seconds, or until 50 rotations are achieved. All results are available in Table 3.

Table 3: The number of successful consecutive rotations in simulation and on the physical robot. All policies were trained on environments with all randomizations enabled. We performed 100 trials in simulation and 10 trials per policy on the physical robot. Each trial terminates when the object is dropped, 50 rotations are achieved or a timeout is reached. For physical trials, results were taken at different times on the physical robot.

Simulated task	Mean	Median	Individual trials (sorted)
Block (state)	43.4 ± 13.8	50	-
Block (state, locked wrist)	44.2 ± 13.4	50	-
Block (vision)	30.0 ± 10.3	33	-
Octagonal prism (state)	29.0 ± 19.7	30	-

Physical task	Mean	Median	Individual trials (sorted)
Block (state)	18.8 ± 17.1	13	50, 41, 29, 27, 14, 12, 6, 4, 4, 1
Block (state, locked wrist)	26.4 ± 13.4	28.5	50, 43, 32, 29, 28, 19, 13, 12, 9
Block (vision)	15.2 ± 14.3	11.5	46, 28, 26, 15, 13, 10, 8, 3, 2, 1
Octagonal prism (state)	7.8 ± 7.8	5	27, 15, 8, 8, 5, 5, 4, 3, 2, 1

Our results allow us to directly compare the performance of each task in simulation and on the real robot. For instance, manipulating a block in simulation achieves a median of 50 successes while the median on the physical setup is 13. This is the overall trend that we observe: Even though randomizations and calibration narrow the reality gap, it still exists and performance on the real system is worse than in simulation. We discuss the importance of individual randomizations in greater detail in Section 6.3.

When using vision for pose estimation, we achieve slightly worse results both in simulation and on the real robot. This is because even in simulation, our model has to perform transfer because it was only trained on images rendered with Unity but we use MuJoCo rendering for evaluation in simulation (thus making this a sim-to-sim transfer problem). On the real robot, our vision model does slightly worse compared to pose estimation with PhaseSpace. However, the difference is surprisingly small, suggesting that training the vision model only in simulation is enough to achieve good performance on the real robot. For vision pose estimation we found that it helps to use a white background and to wipe the object with a tack cloth between trials to remove detritus from the robot hand.

We also evaluate the performance on a second type of object, an octagonal prism. To do so, we finetuned a trained block rotation control policy to the same randomized distribution of environments but with the octagonal prism as the target object instead of the block. Even though our randomizations were all originally designed for the block, we were able to learn successful policies that transfer. Compared to the block, however, there is still a performance gap both in simulation and on the real robot. This suggests that further tuning is necessary and that the introduction of additional randomization could improve transfer to the physical system.

We also briefly experimented with a sphere but failed to achieve more than a few rotations in a row, perhaps because we did not randomize any MuJoCo parameters related to rolling behavior or because rolling objects are much more sensitive to unmodeled imperfections in the hand such as screw holes. It would also be interesting to train a unified policy that can handle multiple objects, but we leave this for future work.

Obtaining the results in Table 3 proved to be challenging due to robot breakages during experiments. Repairing the robot takes time and often changes some aspects of the system, which is why the results were obtained at different times. In general, we found that problems with hardware breakage were one of the key challenges we had to overcome in this work.

6.3 Ablation of Randomizations

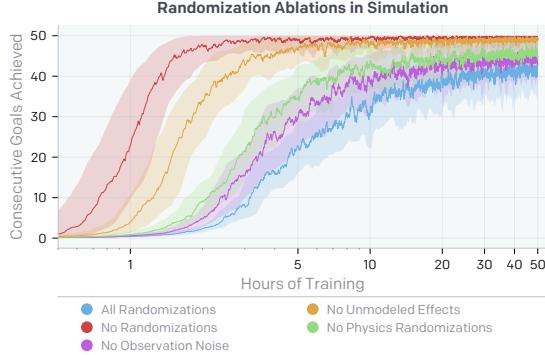


Figure 8: Performance when training in environments with groups of randomizations held out. All runs show exponential moving averaged performance and 90% confidence interval over a moving window of the RL agent in the environment it was trained. We see that training is faster in environments that are easier, e.g. *no randomizations* and *no unmodeled effects*. We only show one seed per experiment; however, in general we have noticed almost no instability in training.

In Section 3.2 we detail a list of parameters we randomize and effects we add that are not already modeled in the simulator. In this section we show that these additions to the simulator are vital for transfer. We train 5 separate RL policies in environments with various randomizations held out: *all randomizations* (baseline), *no observation noise*, *no unmodeled effects*, *no physics randomizations*, and *no randomizations* (basic simulator, i.e. no domain randomization).

Adding randomizations or effects to the simulation does not come without cost; in Figure 8 we show the training performance in simulation for each environment plotted over wall-clock time. Policies trained in environments with a more difficult set of randomizations, e.g. *all randomizations* and *no observation noise*, converge much slower and therefore require more compute and simulated experience to train in. However, when deploying these policies on the real robot we find that training with randomizations is critical for transfer. Table 4 summarizes our results. Specifically, we find that training with all randomizations leads to a median of 13 consecutive goals achieved, while policies trained with *no randomizations*, *no physics randomizations*, and *no unmodeled effects* achieve only median of 0, 2, and 2 consecutive goals, respectively.

Table 4: The number of successful consecutive rotations on the physical robot of 5 policies trained separately in environments with different randomizations held out. The first 5 rows use PhaseSpace for object pose estimation and were run on the same robot at the same time. Trials for each row were interleaved in case the state of the robot changed during the trials. The last two rows were measured at a different time from the first 5 and used the vision model to estimate the object pose.

Training environment	Mean	Median	Individual trials (sorted)
All randomizations (state)	18.8 ± 17.1	13	50, 41, 29, 27, 14, 12, 6, 4, 4, 1
No randomizations (state)	1.1 ± 1.9	0	6, 2, 2, 1, 0, 0, 0, 0, 0, 0
No observation noise (state)	15.1 ± 14.5	8.5	45, 35, 23, 11, 9, 8, 7, 6, 6, 1
No physics randomizations (state)	3.5 ± 2.5	2	7, 7, 7, 3, 2, 2, 2, 2, 2, 1
No unmodeled effects (state)	3.5 ± 4.8	2	16, 7, 3, 3, 2, 2, 1, 1, 0, 0
All randomizations (vision)	15.2 ± 14.3	11.5	46, 28, 26, 15, 13, 10, 8, 3, 2, 1
No observation noise (vision)	5.9 ± 6.6	3.5	20, 12, 11, 6, 5, 2, 2, 1, 0, 0

When holding out *observation noise* randomizations, the performance gap is less clear than for the other randomization groups. We believe that is because our motion capture system has very little noise. However, we still include this randomization because it is important when the vision and control policies are composed. In this case, the pose estimate of the object is much more noisy, and, therefore, training with observation noise should be more important. The results in Table 4 suggest that this is indeed the case, with a drop from median performance of 11.5 to 3.5 if the observation noise randomizations are withheld.

The vast majority of training time is spent making the policy robust to different physical dynamics. Learning to rotate an object in simulation without randomizations requires about 3 years of simulated

experience, while achieving the same performance in a fully randomized simulation requires about 100 years of experience. This corresponds to a wall-clock time of around 1.5 hours and 50 hours in our simulation setup, respectively.

6.4 Effect of Memory in Policies

We find that using memory is helpful to achieve good performance in the randomized simulation. In Figure 9 we show the simulation performance of three different RL architectures: the baseline which has a LSTM policy and value function, a feed forward (FF) policy and a LSTM value function, and both a FF policy and FF value function. We include results for a FF policy with LSTM value function because it was plausible that having a more expressive value function would accelerate training, allowing the policy to act robustly without memory once it converged. However, we see that the baseline outperforms both variants, indicating that it is beneficial to have some amount of memory in the actual policy.

Moreover, we found out that LSTM state is predictive of the environment randomizations. In particular, we discovered that the LSTM hidden state after 5 seconds of simulated interaction with the block allows to predict whether the block is bigger or smaller than average in 80% of cases.

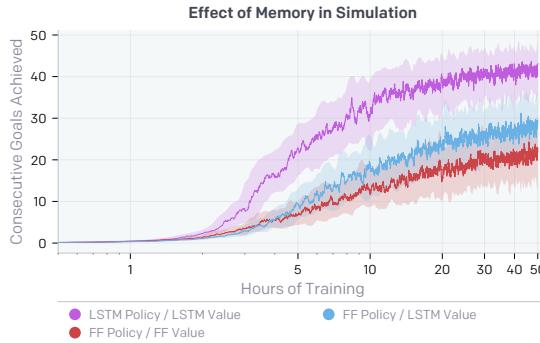


Figure 9: Performance when comparing LSTM and feed forward (FF) policy and value function networks. We train on an environment with all randomizations enabled. All runs show exponential moving averaged performance and 90% confidence interval over a moving window for a single seed. We find that using recurrence in both the policy and value function helps to achieve good performance in simulation.

To investigate the importance of memory-augmented policies for transfer, we evaluate the same three network architectures as described above on the physical robot. Table 5 summarizes the results. Our results show that having a policy with access to memory yields a higher median of successful rotations, suggesting that the policy may use memory to adapt to the current environment.⁸ Qualitatively we also find that FF policies often get stuck and then run out of time.

Table 5: The number of successful consecutive rotations on the physical robot of 3 policies with different network architectures trained on an environment with all randomizations. Results for each row were collected at different times on the physical robot.

Network architecture	Mean	Median	Individual trials (sorted)
LSTM policy / LSTM value (state)	18.8 ± 17.1	13	50, 41, 29, 27, 14, 12, 6, 4, 4, 1
FF policy / LSTM value (state)	4.7 ± 4.1	3.5	15, 7, 6, 5, 4, 3, 3, 2, 2, 0
FF policy / FF value (state)	4.6 ± 4.3	3	15, 8, 6, 5, 3, 3, 2, 2, 0

6.5 Sample Complexity & Scale

In Figure 10 we show results when varying the number of CPU cores and GPUs used in training, where we keep the batch size per GPU fixed such that overall batch size is directly proportional to number of GPUs. Because we could linearly slow down training by simply using less CPU machines and having the GPUs wait longer for data, it is more informative to vary the batch size. We see

⁸When training in an environment with no randomizations, the FF and LSTM policy converge to the same performance in the same amount of time. This shows that a FF policy has the capacity and observations to solve the non-randomized task but cannot solve it reliably with all randomizations, plausibly because it cannot adapt to the environment.

that our default setup with an 8 GPU optimizer and 6144 rollout CPU cores reaches 20 consecutive achieved goals approximately 5.5 times faster than a setup with a 1 GPU optimizer and 768 rollout cores. Furthermore, when using 16 GPUs we reach 40 consecutive achieved goals roughly 1.8 times faster than when using the default 8 GPU setup. Scaling up further results in diminishing returns, but it seems that scaling up to 16 GPUs and 12288 CPU cores gives close to linear speedup.

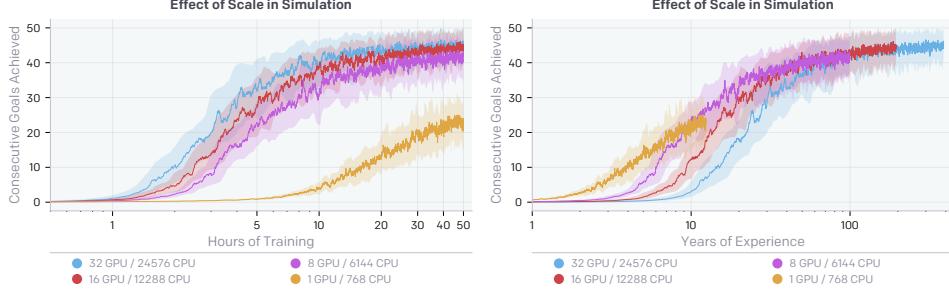


Figure 10: We show performance in simulation when varying the amount of compute used during training versus wall clock training time (left) and years of experience consumed (right). Batch size used is proportional to the number of GPUs used, such that time per optimization step should remain constant apart from slow downs due to gradient syncing across optimizer machines.

6.6 Vision Performance

In Table 3 we show that we can combine a vision-based pose estimator and the control policy to successfully transfer to the real robot without embedding sensors in the target object. To better understand why this is possible, we evaluate the precision of the pose estimator on both synthetic and real data. Evaluating the system in simulation is easy because we can generate the necessary data and have access to the precise object’s pose to compare against. In contrast, real images had to be collected by running a state-based policy on our robot platform. We use PhaseSpace to estimate the object’s pose, which is therefore subject to errors. The resulting collected test set consists of 992 real samples.⁹ For simulation, we use test sets rendered using Unity and MuJoCo. The MuJoCo renderer was not used during training, thus the evaluation can be also considered as an instance of sim-to-sim transfer. Table 6 summarizes our results.

Table 6: Performance of a vision based pose estimator on synthetic and real data.

Test set	Rotation error	Position error
Rendered images (Unity)	$2.71^\circ \pm 1.62$	$3.12\text{mm} \pm 1.52$
Rendered images (MuJoCo)	$3.23^\circ \pm 2.91$	$3.71\text{mm} \pm 4.07$
Real images	$5.01^\circ \pm 2.47$	$9.27\text{mm} \pm 4.02$

Our results show that the model achieves low error for both rotation and position prediction when tested on synthetic data.¹⁰ On the images rendered with MuJoCo, there is only a slight increase in error, suggesting successful sim-to-sim transfer. The error further increases on the real data, which is due to the gap between simulation and reality but also because the ground truth is more challenging to obtain due to noise, occlusions, imperfect marker placement, and delayed sensor readings. Despite that the prediction error is bigger than the observation noise used during policy training (Table 7), the vision-based policy performs well on the physical robot (Table 3).

⁹A sample contains 3 images of the same scene. We removed a few samples that had no object in them after it being dropped.

¹⁰For comparison, PhaseSpace is rated for a position accuracy of around $20\text{ }\mu\text{m}$ but requires markers and a complex setup.

7 Related Work

In order to make it easier to understand the state-of-the-art in dexterous in-hand manipulation we gathered a representative set of videos from related work, and created a playlist¹¹ out of them.

7.1 Dexterous Manipulation

Dexterous manipulation has been an active area of research for decades [17, 52, 7, 42, 37]. Many different approaches and strategies have been proposed over the years. This includes rolling [8, 22, 23, 9, 13], sliding [9, 59], finger gaiting [23], finger tracking [51], pushing [11], and re-grasping [65, 12]. For some hand types, strategies like pivoting [3], tilting [15], tumbling [55], tapping [26], two-point manipulation [2], and two-palm manipulation [14] are also options. These approaches use planning and therefore require exact models of both the hand and object. After computing a trajectory, the plan is typically executed open-loop, thus making these methods prone to failure if the model is not accurate.¹²

Other approaches take a closed-loop approach to dexterous manipulation and incorporate sensor feedback during execution, e.g. tactile sensing [60, 34, 35, 36]. While those approaches allow to correct for mistakes at runtime, they still require reasonable models of the robot kinematics and dynamics, which can be challenging to obtain for under-actuated hands with many degrees of freedom.

Deep reinforcement learning has also been used successfully to learn complex manipulation skills on physical robots. Guided policy search [31, 33] learns simple local policies directly on the robot and distills them into a global policy represented by a neural network. An alternative is to use many physical robots simultaneously in order to be able to collect sufficient experience [20, 32, 27].

7.2 Dexterous In-Hand Manipulation

Since a very large body of past work on dexterous manipulation exists, we limit the more detailed discussion to setups that are most closely related to our work on dexterous in-hand manipulation.

Mordatch et al. [40] and Bai et al. [5] propose methods to generate trajectories for complex and dynamic in-hand manipulation, but results are limited to simulation. There has also been significant progress in learning complex in-hand dexterous manipulation [49, 6] and even tool use [50] using deep reinforcement learning but those approaches were only evaluated in simulation as well.

In contrast, multiple authors learn policies for dexterous in-hand manipulation directly on the robot. Hoof et al. [67] learn in-hand manipulation for a simple 3-fingered gripper whereas Kumar et al. [30, 29] and Falco et al. [16] learn such policies for more complex humanoid hands. While learning directly on the robot means that modeling the system is not an issue, it also means that learning has to be performed with only a handful of trials. This is only possible when learning very simple (e.g. linear or local) policies that, in turn, do not exhibit sophisticated behaviors.

7.3 Sim to Real Transfer

Domain adaption methods [66, 21], progressive nets [53], and learning inverse dynamics models [10] were all proposed to help with sim to real transfer. All of these methods assume access to real data. An alternative approach is to make the policy itself more adaptive during training in simulation using *domain randomization*. Domain randomization was used to transfer object pose estimators [62] and vision policies for fly drones [54]. This idea has also been extended to dynamics randomization [4, 61, 68] to learn a robust policy that transfers to similar environments but with different dynamics. Domain randomization was also used to plan robust grasps [38, 39, 63] and to transfer learned locomotion [61] and grasping [69] policies for relatively simple robots. Pinto et al. [48] propose to use *adversarial training* to obtain more robust policies and show that it also helps with transfer to physical robots [47].

¹¹Related work playlist: <https://bit.ly/2u0K21Q>

¹²Some methods use iterative re-planning to partially mitigate this issue.

8 Conclusion

In this work, we demonstrate that in-hand manipulation skills learned with RL in a simulator can achieve an unprecedented level of dexterity on a physical five-fingered hand. This is possible due to extensive randomizations of the simulator, large-scale distributed training infrastructure, policies with memory, and a choice of sensing modalities which can be modelled in the simulator. Our results demonstrate that, contrary to a common belief, contemporary deep RL algorithms can be applied to solving complex real-world robotics problems which are beyond the reach of existing non-learning-based approaches.

Acknowledgements

We would like to thank Rachel Fong, Ankur Handa and a former OpenAI employee for exploratory work and helpful discussions, a former OpenAI employee for advice and some repairs on hardware and contributions to the low-level PID controller, Pieter Abbeel for helpful discussions, Gavin Cassidy and Luke Moss for their support in maintaining the Shadow hand, and everybody at OpenAI for their help and support.

We would also like to thank the following people for providing feedback on earlier versions of this manuscript: Pieter Abbeel, Joshua Achiam, Tamim Asfour, Aleksandar Botev, Greg Brockman, Rewon Child, Jack Clark, Marek Cygan, Harri Edwards, Ron Fearing, Ken Goldberg, Anna Goldie, Edward Mehr, Azalia Mirhoseini, Lerrel Pinto, Aditya Ramesh, Ian Rust, John Schulman, Shubho Sengupta, and Ilya Sutskever.

References

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [2] T. Abell and M. A. Erdmann. Stably supported rotations of a planar polygon with two frictionless contacts. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 1995, August 5 - 9, 1995, Pittsburgh, PA, USA*, pages 411–418, 1995.
- [3] Y. Aiyama, M. Inaba, and H. Inoue. Pivoting: A new method of graspless manipulation of object by robot fingers. In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 1993, Tokyo, Japan, July 26 - 30, 1993*, pages 136–143, 1993.
- [4] R. Antonova, S. Cruciani, C. Smith, and D. Kragic. Reinforcement learning for pivoting task. *CoRR*, abs/1703.00472, 2017.
- [5] Y. Bai and C. K. Liu. Dexterous manipulation using both palm and fingers. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 1560–1565, 2014.
- [6] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap. Distributed distributional deterministic policy gradients. *CoRR*, abs/1804.08617, 2018.
- [7] A. Bicchi. Hands for dexterous manipulation and robust grasping: a difficult road toward simplicity. *IEEE Trans. Robotics and Automation*, 16(6):652–662, 2000.
- [8] A. Bicchi and R. Sorrentino. Dexterous manipulation through rolling. In *Proceedings of the 1995 International Conference on Robotics and Automation, Nagoya, Aichi, Japan, May 21-27, 1995*, pages 452–457, 1995.
- [9] M. Cherif and K. K. Gupta. Planning quasi-static fingertip manipulations for reconfiguring objects. *IEEE Trans. Robotics and Automation*, 15(5):837–848, 1999.
- [10] P. F. Christiano, Z. Shah, I. Mordatch, J. Schneider, T. Blackwell, J. Tobin, P. Abbeel, and W. Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *CoRR*, abs/1610.03518, 2016.
- [11] N. C. Dafle and A. Rodriguez. Sampling-based planning of in-hand manipulation with external pushes. *CoRR*, abs/1707.00318, 2017.
- [12] N. C. Dafle, A. Rodriguez, R. Paolini, B. Tang, S. S. Srinivasa, M. A. Erdmann, M. T. Mason, I. Lundberg, H. Staab, and T. A. Fuhlbrigge. Extrinsic dexterity: In-hand manipulation with external forces. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 1578–1585, 2014.
- [13] Z. Doulgeri and L. Droukas. On rolling contact motion by robotic fingers via prescribed performance control. In *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*, pages 3976–3981, 2013.

- [14] M. A. Erdmann. An exploration of nonprehensile two-palm manipulation. *I. J. Robotics Res.*, 17(5):485–503, 1998.
- [15] M. A. Erdmann and M. T. Mason. An exploration of sensorless manipulation. *IEEE J. Robotics and Automation*, 4(4):369–379, 1988.
- [16] P. Falco, A. Attawia, M. Saveriano, and D. Lee. On policy learning robust to irreversible events: An application to robotic in-hand manipulation. *IEEE Robotics and Automation Letters*, 3(3):1482–1489, 2018.
- [17] R. S. Fearing. Implementing a force strategy for object re-orientation. In *Proceedings of the 1986 IEEE International Conference on Robotics and Automation, San Francisco, California, USA, April 7-10, 1986*, pages 96–102, 1986.
- [18] T. Feix, J. Romero, H.-B. Schmiedmayer, A. Dollar, and D. Kragic. The grasp taxonomy of human grasp types. *Human-Machine Systems, IEEE Transactions on*, 46(1):66–77, 2016.
- [19] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning. *arXiv preprint arXiv:1509.06113*, 2015.
- [20] S. Gu, E. Holly, T. P. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 3389–3396, 2017.
- [21] A. Gupta, C. Devin, Y. Liu, P. Abbeel, and S. Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *CoRR*, abs/1703.02949, 2017.
- [22] L. Han, Y. Guan, Z. X. Li, S. Qi, and J. C. Trinkle. Dextrous manipulation with rolling contacts. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation, Albuquerque, New Mexico, USA, April 20-25, 1997*, pages 992–997, 1997.
- [23] L. Han and J. C. Trinkle. Dextrous manipulation by rolling and finger gaiting. In *Proceedings of the IEEE International Conference on Robotics and Automation, ICRA-98, Leuven, Belgium, May 16-20, 1998*, pages 730–735, 1998.
- [24] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [25] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [26] W. H. Huang and M. T. Mason. Mechanics, planning, and control for tapping. *I. J. Robotics Res.*, 19(10):883–894, 2000.
- [27] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. *ArXiv e-prints*, June 2018.
- [28] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [29] V. Kumar, A. Gupta, E. Todorov, and S. Levine. Learning dexterous manipulation policies from experience and imitation. *CoRR*, abs/1611.05095, 2016.
- [30] V. Kumar, E. Todorov, and S. Levine. Optimal control with learned local models: Application to dexterous manipulation. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, pages 378–383, 2016.
- [31] S. Levine and V. Koltun. Guided policy search. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1–9, 2013.
- [32] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *I. J. Robotics Res.*, 37(4-5):421–436, 2018.
- [33] S. Levine, N. Wagener, and P. Abbeel. Learning contact-rich manipulation skills with guided policy search. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 156–163, 2015.
- [34] M. Li, Y. Bekiroglu, D. Kragic, and A. Billard. Learning of grasp adaptation through experience and tactile sensing. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 3339–3346, 2014.
- [35] M. Li, H. Yin, K. Tahara, and A. Billard. Learning object-level impedance control for robust grasping and dexterous manipulation. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 6784–6791, 2014.
- [36] Q. Li, M. Meier, R. Haschke, H. J. Ritter, and B. Bolder. Rotary object dexterous manipulation in hand: a feedback-based method. *IJMA*, 3(1):36–47, 2013.
- [37] R. R. Ma and A. M. Dollar. On dexterity and dexterous manipulation. In *15th International Conference on Advanced Robotics: New Boundaries for Robotics, ICAR 2011, Tallinn, Estonia, June 20-23, 2011*, pages 1–7, 2011.
- [38] J. Mahler, J. Liang, S. Niyaz, M. Laskey, R. Doan, X. Liu, J. A. Ojea, and K. Goldberg. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. In *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017.
- [39] J. Mahler, M. Matl, X. Liu, A. Li, D. V. Gealy, and K. Goldberg. Dex-net 3.0: Computing robust robot suction grasp targets in point clouds using a new analytic model and deep learning. *CoRR*, abs/1709.06670, 2017.

- [40] I. Mordatch, Z. Popovic, and E. Todorov. Contact-invariant optimization for hand manipulation. In *Proceedings of the 2012 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, SCA 2012, Lausanne, Switzerland, 2012*, pages 137–144, 2012.
- [41] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [42] A. M. Okamura, N. Smaby, and M. R. Cutkosky. An overview of dexterous manipulation. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, April 24-28, 2000, San Francisco, CA, USA*, pages 255–262, 2000.
- [43] OpenAI. OpenAI Five. <https://blog.openai.com/openai-five/>, 2018.
- [44] C. Pehoski, A. Henderson, and L. Tickle-Degnen. In-hand manipulation in young children: rotation of an object in the fingers. *American Journal of Occupational Therapy*, 51(7):544–552, 1997.
- [45] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. *CoRR*, abs/1710.06537, 2017.
- [46] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *arXiv preprint arXiv:1710.06542*, 2017.
- [47] L. Pinto, J. Davidson, and A. Gupta. Supervision via competition: Robot adversaries for learning tasks. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 1601–1608, 2017.
- [48] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta. Robust adversarial reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2817–2826, 2017.
- [49] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.
- [50] A. Rajeswaran, V. Kumar, A. Gupta, J. Schulman, E. Todorov, and S. Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *CoRR*, abs/1709.10087, 2017.
- [51] D. Rus. Dexterous rotations of polyhedra. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation, Nice, France, May 12-14, 1992*, pages 2758–2763, 1992.
- [52] D. Rus. In-hand dexterous manipulation of piecewise-smooth 3-d objects. *I. J. Robotics Res.*, 18(4):355–381, 1999.
- [53] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, pages 262–270, 2017.
- [54] F. Sadeghi and S. Levine. CAD2RL: real single-image flight without a single real image. In *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017.
- [55] N. Sawasaki and H. INOUE. Tumbling objects using a multi-fingered robot. *Journal of the Robotics Society of Japan*, 9(5):560–571, 1991.
- [56] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.
- [57] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [58] ShadowRobot. ShadowRobot Dexterous Hand. <https://www.shadowrobot.com/products/dexterous-hand/>, 2005.
- [59] J. Shi, J. Z. Woodruff, P. B. Umbohanowar, and K. M. Lynch. Dynamic in-hand sliding manipulation. *IEEE Trans. Robotics*, 33(4):778–795, 2017.
- [60] K. Tahara, S. Arimoto, and M. Yoshida. Dynamic object manipulation using a virtual frame by a triple soft-fingered robotic hand. In *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, pages 4322–4327, 2010.
- [61] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *CoRR*, abs/1804.10332, 2018.
- [62] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *arXiv preprint arXiv:1703.06907*, 2017.
- [63] J. Tobin, W. Zaremba, and P. Abbeel. Domain randomization and generative models for robotic grasping. *CoRR*, abs/1710.06425, 2017.
- [64] E. Todorov, T. Erez, and Y. 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.
- [65] P. Tournassoud, T. Lozano-Pérez, and E. Mazer. Regrasping. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, USA, March 31 - April 3, 1987*, pages 1924–1928, 1987.
- [66] E. Tzeng, C. Devin, J. Hoffman, C. Finn, X. Peng, S. Levine, K. Saenko, and T. Darrell. Towards adapting deep visuomotor representations from simulated to real environments. *CoRR*, abs/1511.07111, 2015.
- [67] H. van Hoof, T. Hermans, G. Neumann, and J. Peters. Learning robot in-hand manipulation with tactile features. In *15th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2015, Seoul, South Korea, November 3-5, 2015*, pages 121–127, 2015.

- [68] W. Yu, J. Tan, C. K. Liu, and G. Turk. Preparing for the unknown: Learning a universal policy with online system identification. In *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017.
- [69] Y. Zhu, Z. Wang, J. Merel, A. A. Rusu, T. Erez, S. Cabi, S. Tunyasuvunakool, J. Kramár, R. Hadsell, N. de Freitas, and N. Heess. Reinforcement and imitation learning for diverse visuomotor skills. *CoRR*, abs/1802.09564, 2018.

Appendices

A Reinforcement Learning Background	20
A.1 Reinforcement Learning (RL)	20
A.2 Generalized Advantage Estimator (GAE)	20
A.3 Proximal Policy Optimization (PPO)	20
B Hardware Description	21
B.1 ShadowRobot Dexterous Hand	21
B.2 PhaseSpace Visual Tracking	21
B.3 RGB Cameras	21
B.4 Control	21
B.5 Joint Sensor Calibration	21
C Simulated Environment	22
C.1 Deterministic Environment	22
C.2 Randomizations	23
C.3 MuJoCo Model Calibration	24
D Optimization Details	25
D.1 Control Policy	25
D.2 Vision Model	26

A Reinforcement Learning Background

A.1 Reinforcement Learning (RL)

We consider the standard reinforcement learning formalism consisting of an agent interacting with an environment. To simplify the exposition we assume in this section that the environment is fully observable.¹³ An environment is described by a set of states \mathcal{S} , a set of actions \mathcal{A} , a distribution of initial states $p(s_0)$, a reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, transition probabilities $p(s_{t+1}|s_t, a_t)$, and a discount factor $\gamma \in [0, 1]$.

A policy π is a mapping from state to a distribution over actions. Every episode starts by sampling an initial state s_0 . At every timestep t the agent produces an action based on the current state: $a_t \sim \pi(\cdot|s_t)$. In turn, the agent receives a reward $r_t = r(s_t, a_t)$ and the environment's new state s_{t+1} , which is sampled from the distribution $p(\cdot|s_t, a_t)$. The discounted sum of future rewards, also referred to as the *return*, is defined as $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. The agent's goal is to maximize its expected return $\mathbb{E}[R_0|s_0]$, where the expectation is taken over the initial state distribution, policy, and environment transitions accordingly to the dynamics specified above. The *Q-function* or *action-value* function is defined as $Q^\pi(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]$, while the *V-function* or *state-value* function is defined as $V^\pi(s_t) = \mathbb{E}[R_t|s_t]$. The value $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ is called the *advantage* and tells whether the action a_t is better or worse than an average action the policy π takes in the state s_t .

A.2 Generalized Advantage Estimator (GAE)

Let V be an approximator to the value function of some policy, i.e. $V \approx V^\pi$. The value

$$\hat{V}_t^{(k)} = \sum_{i=t}^{t+k-1} \gamma^{i-t} r_i + \gamma^k V(s_{t+k}) \approx V^\pi(s_t, a_t)$$

is called the k -step return estimator. The parameter k controls the bias-variance tradeoff of the estimator with bigger values resulting in an estimator closer to empirical returns and having less bias and more variance. *Generalized Advantage Estimator (GAE)* [56] is a method of combining multi-step returns in the following way:

$$\hat{V}_t^{\text{GAE}} = (1 - \lambda) \sum_{k>0} \lambda^{k-1} \hat{V}_t^{(k)} \approx V^\pi(s_t, a_t),$$

where $0 < \lambda < 1$ is a hyperparameter. Using these to estimate the *advantage*:

$$\hat{A}_t^{\text{GAE}} = \hat{V}_t^{\text{GAE}} - V(s_t) \approx A^\pi(s_t, a_t).$$

It is possible to compute the values of this estimator for all states encountered in an episode in linear time [56].

A.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [57] is one of the most popular on-policy RL algorithms. It simultaneously optimizes a stochastic policy as well as an approximator to the value function. PPO interleaves the collection of new episodes with policy optimization. After a batch of new transitions is collected, optimization is performed with minibatch stochastic gradient descent to maximize the objective

$$L_{\text{PPO}} = \mathbb{E} \min \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \hat{A}_t^{\text{GAE}}, \text{clip} \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_t^{\text{GAE}} \right),$$

where $\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ is the ratio of the probability of taking the given action under the current policy π to the probability of taking the same action under the old behavioral policy that was used to generate the data, and ϵ is a hyperparameter (usually $\epsilon \approx 0.2$). This loss encourages the policy to take actions which are better than average (have positive advantage) while clipping discourages bigger changes to

¹³The environments we consider in the paper are only partially observable.

the policy by limiting how much can be gained by changing the policy on a particular data point. The value function approximator is trained with supervised learning with the target for $V(s_t)$ being \hat{V}_t^{GAE} . To boost exploration, it is a common practice to encourage the policy to have high entropy of actions by including an entropy bonus in the optimization objective.

B Hardware Description

B.1 ShadowRobot Dexterous Hand

We use the ShadowRobot Dexterous Hand. Concretely, we use the version with electric motor actuators, EDC hand (EtherCAT-Dual-CAN).

The hand has 24 degrees of freedom between its links and is actuated by 40 Spectra tendons controlled by 20 DC motors in the base of the hand, each actuating a pair of agonist–antagonist tendons connected via a spool. 16 degrees of freedom can be controlled independently whereas the remaining 8 joints (which are the joints between the non-thumb finger proximal, middle, and distal segments) form 4 pairs of coupled joints.

B.2 PhaseSpace Visual Tracking

We use a 3D tracking system to localize the tips of the fingers, to perform calibration procedures, and as ground truth for the RGB image-based tracking. The PhaseSpace Impulse X2E tracking system uses active LED markers that blink to transmit a unique ID code and linear detector arrays in the cameras to detect the positions and IDs. The system features capture speeds of up to 960 Hz and positional accuracies of below $20 \mu\text{m}$. The data is exposed as a 3D point cloud together with labels associating the points with stable numerical IDs. Our setup uses 16 cameras distributed spherically around the hand and centered on the palm with a radius of approximately 0.8 meters.

B.3 RGB Cameras

We also use RGB images to track the objects for manipulation. We perform object pose estimation using 3 Basler acA640-750uc RGB cameras with a resolution of 640x480 placed approximately 50 cm from the Shadow hand. We use 3 cameras to resolve pose ambiguities that may occur with monocular vision. We chose these cameras for their flexible parameterization and low latency. Figure 11 shows the placement of the cameras relative to the hand.

B.4 Control

The high-level controller is implemented as a Python program running a neural network policy implemented in Tensorflow [1] on a GPU. Every 80ms it queries the Phasespace sensors and then runs inference with the neural network to obtain the action, which takes roughly 25ms. The policy outputs an action that specifies the change of position for each actuator, relative to the current position of the joints controlled by the actuator. It then sends the action to the low-level controller.

The low-level controller is implemented in C++ as a separate process on a different machine which is connected to the Shadow hand via an Ethernet cable. The controller is written as a real-time system — it is pinned to a CPU core, has preallocated memory, and does not depend on any garbage collector to avoid non-deterministic delays. The controller receives the relative action, converts it to an absolute joint angle and clips to the valid range, then sets each component of the action as the target for a PD controller. Every 5ms, the PD controller queries the Shadow hand joint angle sensors, then attempts to achieve the desired position.

Surprisingly, decreasing time between actions to 40ms increased training time but did not noticeably improve performance in the real world.

B.5 Joint Sensor Calibration

The hand contains 26 Hall effect sensors that sense magnetic field rotations along the joint axis. To transform the raw magnetic measurements from the Hall sensors into joint angles, we use a piecewise linear function interpolated from 3-5 truth points per joint. To calibrate this function,

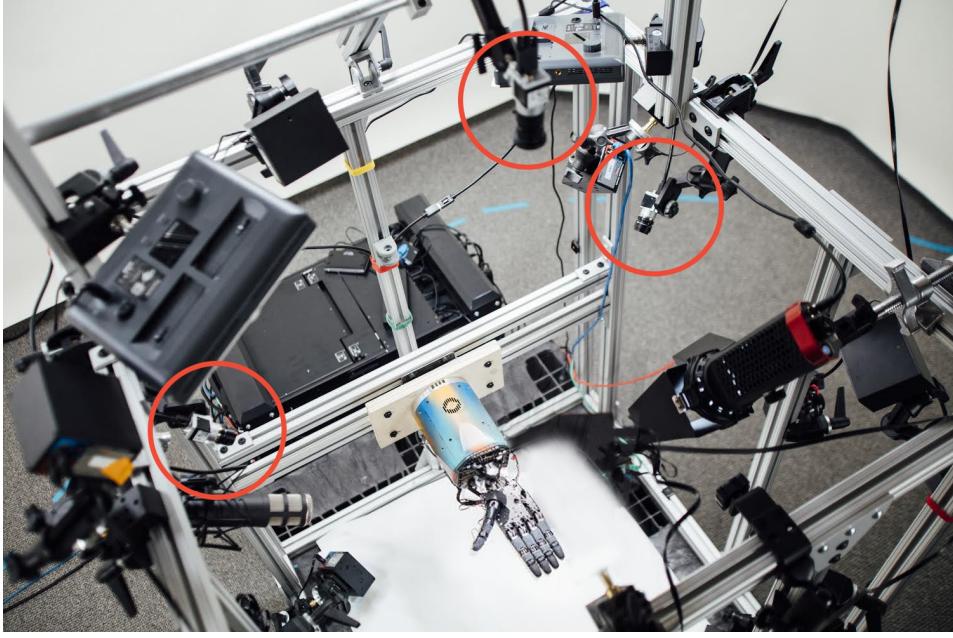


Figure 11: Our 3-camera setup for vision-based state estimation.

we initialize to the factory default created using physical calibration jigs. For further accuracy, we attach PhaseSpace markers to the fingertips, and minimize error between the position reported by the PhaseSpace markers and the position estimated from the joint angles. We estimate these linear functions by minimizing reprojection error with `scipy.optimize.minimize`.

C Simulated Environment

C.1 Deterministic Environment

Simulation. Our environment is based on the OpenAI Gym robotics environments described in [49]. We use MuJoCo for simulation [64].

States. The state of the system is 60-dimensional and consists of angles and velocities of all robot joints as well as the position, rotation and velocities (linear and angular) of the object. Initial states are sampled by placing the object on the palm in a random orientation and applying random actions for 100 steps (we discard the trial if the object is dropped in the meantime).

Goals. The goal is the desired orientation of the object represented as a quaternion. A new goal is generated after the current one has been achieved within a tolerance of 0.4 rad.¹⁴

Observations. Described in Table 2.

Rewards. The reward given at timestep t is $r_t = d_t - d_{t+1}$, where d_t and d_{t+1} are the rotation angles between the desired and current object orientations before and after the transition, respectively. We give an additional reward of 5 whenever a goal is achieved with the tolerance of 0.4 rad (i.e. $d_{t+1} < 0.4$) and a reward of -20 (penalty) whenever the object is dropped.

Actions. Actions are 20-dimensional and correspond to the desired angles of the hand joints. We discretize each action coordinate into 11 bins of equal size. Due to the inaccuracy of joint angle sensors on the physical hand (see Appendix B), actions are specified relative to the current hand state.

¹⁴I.e. we consider a goal as achieved if there exists a rotation of the object around an arbitrary axis with an angle smaller than 0.4 rad which transforms the current orientation into the desired one.

Table 7: Standard deviation of observation noise.

Measurement	Correlated noise	Uncorrelated noise
fingertips positions	1mm	2mm
object position	5mm	1mm
object orientation	0.1rad	0.1rad
fingertip marker positions	3mm	
hand base marker position	1mm	

Table 8: Standard deviation of action noise.

Noise type	Percentage of the action range
uncorrelated additive	5%
correlated additive	1.5%
uncorrelated multiplicative	1.5%

In particular, the torque applied to the given joint in simulation is equal to $P * (s_t + a - s_{t'})$, where s_t is the joint angle at the time when the action was specified, a is the corresponding action coordinate, $s_{t'}$ is the current joint angle, and P is the proportionality coefficient. For the coupled joints, the desired and actual positions represent the sum of the two joint angles. All actions are rescaled to the range $[-1, 1]$. To avoid abrupt changes to the action signal, which could harm a physical robot, we smooth the actions using an exponential moving average¹⁵ before applying them (both in simulation and during deployments on the physical robot).

Timing. Each environment step corresponds to 80 ms of real time and consists of 10 consecutive MuJoCo steps, each corresponding to 8 ms. The episode ends when either the policy achieves 50 consecutive goals, the policy fails to achieve the current goal within 8 seconds of simulated time, or the object is dropped.

C.2 Randomizations

A variety of randomizations are applied to the simulator, shrinking the reality gap between the simulated environment and the physical world in order to learn a policy that generalizes to reality.

Physical parameters. The physical parameters are sampled at the beginning of every episode and held fixed for the whole episode. The full set of randomized values are displayed in Table 1.

Observation noise. We use two types of noise — *correlated* noise which is sampled once per episode and kept fixed, and an *uncorrelated* Gaussian one. Apart from Gaussian correlated noise, we also add more structured noise coming from inaccurate placement of the motion capture markers by computing the observations using slightly misplaced markers in the simulator. The configuration of noise levels is described in Table 7. The observation noise is only applied to the policy inputs and not to the value network inputs as the value function is not used during the deployment on the physical system.

PhaseSpace tracking errors. Noise aside, readings of the motion capture markers from the PhaseSpace system might be occasionally unavailable for a short period of time due to instability of the service. To model such error in the simulator, we mask the fingertip markers with a small probability (0.2 per second) for 1 second so that the policy has a chance to learn how to interact with the environment while the system temporarily loses track of some markers.

¹⁵We use a coefficient of 0.3 per 80ms.

Furthermore, the markers might be occluded while in motion, causing a brief delay of readings of some fingertip positions. In the simulator, a small weightless cuboid site¹⁶ is attached to the back of each nail and we consider a marker occluded whenever a collision with the site is detected as another finger or object is getting too close. If a fingertip marker is deemed occluded, we use its last available position readings instead of the current one.

Action noise and delay. We add correlated and uncorrelated Gaussian noise to all actions to account for an imperfect actuation system. The detailed noise levels can be found in Table 8. Moreover, the real system contains many potential sources of delays between the time that observations are sensed and actions are executed, from network delay to the computation time of the neural network. Therefore, we introduce a simple model of action delay to the simulator. At the beginning of every episode we sample for every actuator whether it is going to be delayed (with probability 0.5) or not. The actions corresponding to delayed actuator are delayed by one environment step, i.e. approximately 80ms.

Timing randomization. We also randomize the timing of environment steps. Every environment step is simulated as 10 MuJoCo physics simulator steps with $\Delta t = 8\text{ms} + \text{Exp}(\lambda)$, where $\text{Exp}(\lambda)$ denotes the exponential distribution and the coefficient λ is once per episode sampled uniformly from the range [1250, 10000].

Backlash model. The physical Shadow Dexterous Hand is tendon-actuated which causes a substantial amount of backlash, while the MuJoCo model assumes direct actuation on the joints. In order to account for it, we introduce a simple model of backlash which modifies actions before they are sent to MuJoCo. In particular, for every joint we have two parameters which specify the amount of backlash in each direction, and are denoted δ_{-1} and δ_{+1} , as well as a time varying variable s denoting the current state of slack. We obtained the values of δ_{-1}, δ_{+1} through calibration. At the beginning of every episode we sample the values of δ_{-1}, δ_{+1} from the Gaussian distribution centered around the calibrated values with the standard deviation of 0.1. Let $a_{\text{in}} \in [-1, 1]$ be an action specified by the policy. Our backlash model works as follows: we compute the new value of the slack variable $s' = [s + a_{\text{in}} \delta_{\text{sgn}(a_{\text{in}})} \Delta t]_{-1}^{+1}$, compute the scaling factor $\alpha = 1 - \left[\frac{|\text{sgn}(a_{\text{in}}) - s|}{|s' - s| + \epsilon} \right]_0^1$, where $\epsilon = 10^{-12}$ is a constant used for numerical stability, and finally multiply the action by α : $a_{\text{out}} = \alpha a_{\text{in}}$.

Random forces on the object. To represent unmodeled dynamics, we sometimes apply random forces on the object. The probability p that a random force is applied is sampled at the beginning of the episode from the loguniform distribution between 0.1% and 10%. Then, at every timestep, with probability p we apply a random force from the 3-dimensional Gaussian distribution with the standard deviation equal to 1 m/s^2 times the mass of the object on each coordinate and decay the force with the coefficient of 0.99 per 80ms.

Randomized vision appearance. We randomize the visual appearance of the robot and object, as well as lighting and camera characteristics. The materials and textures are randomized for every visible object in the scene. We randomize the hue, saturation, and value for the object faces around calibrated values from real-world measurements. The color of the robot is uniformly randomized. Material properties such as glossiness and shininess are randomized as well. Camera position and orientation are slightly randomized around values calibrated to real-world locations. Lights are randomized individually, and intensities are scaled based on a randomly drawn total intensity. After rendering the scene to images from the three separate cameras, additional augmentation is applied. The images are linearly normalized to have zero mean and unit variance. Then the image contrast is randomized, and finally per-pixel Gaussian noise is added. Details are in Table 9.

C.3 MuJoCo Model Calibration

The MuJoCo XML model of the hand requires many parameters, which are then used as the mean of the randomized distribution of each parameter for the environment. Even though substantial

¹⁶A site represents a location of interest relative to the body frame in MuJoCo. Also see <http://mujoco.org/book/modeling.html#site>.

¹⁷In units used by Unity. See <https://unity3d.com/learn/tutorials/s/graphics>.

Table 9: Vision randomizations.

Randomization type	Range
number of cameras	3
camera position	± 1.5 mm
camera rotation	0–3° around a random axis
camera field of view	$\pm 1^\circ$
robot material colors	RGB
robot material metallic level	5%–25% ¹⁷
robot material glossiness level	0%–100% ¹⁷
object material hue	calibrated hue $\pm 1\%$
object material saturation	calibrated saturation $\pm 15\%$
object material value	calibrated value $\pm 15\%$
object metallic level	5%–15% ¹⁷
object glossiness level	5%–15% ¹⁷
number of lights	4–6
light position	uniform over upper half-sphere
light relative intensity	1–5
total light intensity	0–15 ¹⁷
image contrast adjustment	50%–150%
additive per-pixel Gaussian noise	$\pm 10\%$

randomization is required to achieve good performance on the physical robot, we have found that it is important for the mean of the randomized distributions to correspond to reasonable physical values. We calibrate these values by recording a trajectory on the robot, then optimizing the default value of the parameters to minimize error between the simulated and real trajectory.

To create the trajectory, we run two policies in sequence against each finger. The first policy measures behavior of the joints near their limits by extending the joints of each finger completely inward and then completely outward until they stop moving. The second policy measures the dynamic response of the finger by moving the joints of each finger inward and then outward in a series of oscillations. The recorded trajectory across all fingers lasts a few minutes.

To optimize the model parameters, these trajectories are then replayed as open-loop action sequences in the simulator. The optimization objective is to match joint angles after 1 second of running actions. Parameters are adjusted using iterative coordinate descent until the error is minimized. We exclude modifications to the XML that does not yield improvement over 0.1%.

For each joint, we optimize the damping, equilibrium position, static friction loss, stiffness, margin, and the minimum and maximum of the joint range. For each actuator, we optimize the proportional gain, the force range, and the magnitude of backlash in each direction. Collectively, this corresponds to 264 parameter values.

D Optimization Details

D.1 Control Policy

We normalize all observations given to the policy and value networks with running means and standard deviations. We then clip observations such that they are within 5 standard deviations of the mean. We normalize the advantage estimates within each minibatch. We also normalize targets for the value function with running statistics. The network architecture is depicted in Figure 12.

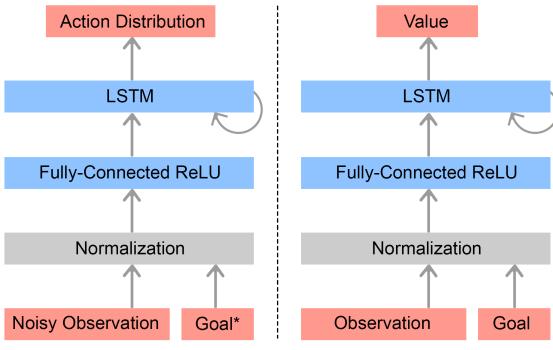


Figure 12: Policy network (left) and value network (right). Each network consists of an input normalization, a single fully-connected hidden layer with ReLU activations [41] and a recurrent LSTM block [25]. The normalization block subtracts the mean value of each coordinate (across all data gathered so far), divides by the standard deviation, and removes outliers by clipping. There is no weight sharing between the two networks. The goal provided to the policy is the noisy relative target orientation (see Table 2 for details).

Table 10: Hyperparameters used for PPO.

Hyperparameter	Value
hardware configuration	8 NVIDIA V100 GPUs + 6144 CPU cores
action distribution	categorical with 11 bins for each action coordinate
discount factor γ	0.998
Generalized Advantage Estimation λ	0.95
entropy regularization coefficient	0.01
PPO clipping parameter ϵ	0.2
optimizer	Adam [28]
learning rate	3e-4
batch size (per GPU)	80k chunks x 10 transitions = 800k transitions
minibatch size (per GPU)	25.6k transitions
number of minibatches per step	60
network architecture	dense layer with ReLU + LSTM
size of dense hidden layer	1024
LSTM size	512

D.2 Vision Model

Vision training hyperparameters are given in Table 11 and the details of the model architecture are given in Table 12.

We also apply data augmentation for training. More specifically, we leave the object pose as is with 20% probability, rotate the object by 90° around its main axes with 40% probability, and “jitter” the object by adding Gaussian noise to both the position and rotation independently with 40% probability.

¹⁸Two GPUs are used for rendering and one for the optimization.

Table 11: Hyperparameters used for the vision model training.

Hyperparameter	Value
hardware configuration	3 NVIDIA P40 GPUs ¹⁸ + 32 CPU cores
optimizer	Adam [28]
learning rate	0.0005, halved every 20 000 batches
minibatch size	$64 \times 3 = 192$ RGB images
image size	200 × 200 pixels
weight decay regularization	0.001
number of training batches	400 000
network architecture	shown in Figure 6

Table 12: Hyperparameters for the vision model architecture.

Layer	Details
Input RGB Image	200 × 200 × 3
Conv2D	32 filters, 5 × 5, stride 1, no padding
Conv2D	32 filters, 3 × 3, stride 1, no padding
Max Pooling	3 × 3, stride 3
ResNet	1 block, 16 filters, 3 × 3, stride 3
ResNet	2 blocks, 32 filters, 3 × 3, stride 3
ResNet	2 blocks, 64 filters, 3 × 3, stride 3
ResNet	2 blocks, 64 filters, 3 × 3, stride 3
Spatial Softmax	
Flatten	
Concatenate	all 3 image towers combined
Fully Connected	128 units
Fully Connected	output dimension (3 position + 4 rotation)