Master's Thesis

Adaptive Curriculum Generation from Demonstrations

Lukas Hermann

June 05th, 2019

Albert-Ludwigs-University Freiburg

Department of Computer Science

Autonomous Intelligent Systems

Writing Period

25. 10. 2018 – 05. 06. 2019

Examiner

Prof. Dr. Wolfram Burgard

Second Examiner

Prof. Dr. Thomas Brox

Advisers

Andreas Eitel, Max Argus

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Abstract

Reinforcement learning (RL) from sparse rewards suffers from high sample complexity, which can make the learning of complex robot manipulation tasks infeasible or very time-consuming. Without a guiding reward signal, the agent has to randomly explore a potentially large environment in order to stumble upon a sparse goal state. We propose Adaptive Curriculum Generation from Demonstrations (ACGD), a model-free deep RL method that overcomes exploration difficulties by using a small set of human demonstrations to build a reverse curriculum of initial states. Our algorithm automatically schedules start states and adaptively sets the difficulty of task parameters such that the policy learns at the appropriate level of difficulty. In experiments, our approach shows superior performance in comparison to standard RL or other curriculum learning methods. We demonstrate successful zero-shot sim-to-real transfer of a policy for block stacking that was trained with domain randomization. The end-to-end visuomotor policy successfully manages to stack blocks in 67% of the cases and has a success rate of 93% for picking up blocks.

Contents

List of Figures

List of Tables

1 Introduction

Recently, reinforcement learning (RL) has had notable successes such as learning to play Atari games from pixel input (Mnih et al. (2013)) or beating the world champion in the board game Go (Silver et al. (2016)). In the field of robotic manipulation however, learning robot control policies with reinforcement learning still remains a challenging problem, especially when training on real hardware. This is due to noisy sensor data, high-dimensional continuous action spaces, infeasible parallelization because of high material costs, long training times and safety concerns.
Model-based methods theoretically promise to be more data efficient, but they require an accurate model of the environment, which can be prohibitively hard to learn for robotic tasks with complex, contact-rich dynamics. If no accurate model is available, model-free methods can be superior and are often preferred, but their high sample complexity makes training directly on real hardware extremely challenging and time-consuming. Training in simulation and transferring the learned policy on the physical robot can mitigate some of the problems, but in a sparse reward setting, naive reinforcement learning still suffers from low sample efficiency. This means that an agent needs a prohibitive amount of random exploration to encounter goal states that yield a sparse reward. A sparse reward function, as opposed to a dense reward function, is not informative about the direction in which to explore the state space. Nevertheless, sparse rewards tend to be preferred because of their simplicity and their unbiased nature. In order to enable learning of complex, episodic tasks in a reasonable amount of time, it is necessary to guide the exploration in such a way that states which exhibit reward signals are visited frequently. Intuitively, by starting the

exploration from states nearby a goal region, the agent will reach those states more often compared to starts from states that are further away. Using demonstrations to shape the distribution of initial states has been investigated in the context of robotics by Popov et al. (2017), Nair et al. (2017), Zhu et al. (2018) or Fan et al. (2018), but none of these methods fully exploit the potential of curriculum learning from demonstration states.

We propose a novel method that we call Adaptive Curriculum Generation from Demonstrations (ACGD). It only requires a small set of around a dozen task demonstration in simulation, which can be recorded in a short time. The algorithm automatically builds a reverse curriculum of increasingly more difficult start states, such that the RL agent always learns at the appropriate level of difficulty. Furthermore, it also uses the curriculum learning framework to gradually learn visual domain randomization and dynamics randomization until the policy exhibits the desired degree of domain invariance. The policy is learned end-to-end from raw pixels and proprioceptive features and produces actions for the robot controller.

We evaluate the performance of our approach on a real world block stacking task with the KUKA iiwa robot and demonstrate successful zero-shot (i.e. without additional training) sim-to-real transfer. Although there is a significant domain gap to bridge, the robot succeeds to stack the blocks in 67% of the cases, without ever being trained on real hardware.

Additionally, we present a modular RL Python framework for training, transfer and control of the real KUKA robot, which adheres to the OpenAI Gym (Brockman et al. (2016)) environment architecture standard. It features an extendable simulated robot learning environment with integrated domain and dynamics randomization. Its modularity enables to train and test any OpenAI Gym compatible RL algorithm in both the simulated and the real robot environment.

2 Related Work

In this chapter, we will review previous work that has been done in the context of reinforcement learning for robotics manipulation. We compare approaches that were learned directly on real hardware with approaches that involve training in simulation and domain transfer. In addition, we give an overview about methods to reduce sample complexity of RL, such as curriculum learning.

2.1 RL Training on a Physical Robot

Training directly on the physical device has the advantage that there is no domain gap to bridge for the deployment of the learned policy. This means that the training samples are generated from the same distribution as the data of the environment at test time, which is not the case for policies trained in simulation.

Generally, approaches can be distinguished according to their input modalities. Learning a neural network directly from pixels (end-to-end) is desirable, since the policy can autonomously choose the features that are relevant for a task. On the other hand, training on the state space requires an additional object recognition system, but it also drastically reduces the amount of network parameters and ultimately saves training time.

Since time is a limited resource on physical systems and deep RL policies generally need millions of interaction steps to explore the environment, earlier methods often completely avoided training neural networks. Instead, they relied on pre-structured policy representations such as splines or motor primitives to teach a robot simple behavior like hitting a baseball with a bat (Peters and Schaal (2006)). Deisenroth et al. (2011) propose a model-based RL technique to learn a probabilistic dynamics model for block stacking from scratch on a low-cost robotic arm in combination with a block tracking system. While having the advantage of being sample-efficient enough to fully train policies in reasonable amounts of time on real hardware, these approaches hardly generalize to more variable task setups and only perform well in narrow ranges of trajectories.

A more recent model-based approach uses Guided Policy Search to learn robot manipulation tasks like shape sorting end-to-end (from raw camera input to motor torques) (Levine et al. (2015)). Model-based methods can be more sample-efficient than model-free methods, but for complex tasks, learning the model is a hard problem on its own. Model-free methods need to make less assumptions about the environment and are often preferred for learning complex policies, although actions have to be taken to improve exploration.

Gu* et al. (2017) use multiple robotic platforms in parallel to train Normalized Advantage Functions (NAF) asynchronously from scratch for a door opening task, but they do not use raw pixel input and the door position is kept fix throughout the training.

In general, there are two contrary control paradigms: in open-loop systems, the planning phase is decoupled from the actual execution of the task. This is especially common practise for grasping systems, Bousmalis et al. (2017) e.g. demonstrate successful grasping of unseen objects using a grasp proposal network together with a manually designed servoing function. On the other hand, although harder to train, a closed-loop method that learns perception, planning and acting jointly is expected to result in more robust behavior and in the emergence of intelligent manipulation policies (Kalashnikov et al. (2018)). It can directly react to unforeseen events, for instance directly regrasping an object after a failed grasping attempt. Since tasks that require a lot of object interaction such as block stacking heavily rely on direct feedback from the environment, our approach learns the whole pipeline from vision to action outputs jointly. Kalashnikov et al. (2018) use a setup of seven identical robots to learn closed-loop grasping of unknown objects in a self-supervised manner. A recent sample-efficient method proposed by Riedmiller et al. (2018) teaches a robot to perform a block pushing task in only about 10 hours training time on a single robot. They use a hierarchical architecture with a learned scheduling policy to pick auxiliary tasks in order to learn increasingly more complicated problems until it is able to complete the main task; thus, it can be understood as a form of implicit curriculum learning. Our method may also be seen as scheduling increasingly more difficult sub-tasks, however without explicitly defining auxiliary tasks.

2.2 Deep RL for Robotics using Simulation

The sample complexity of RL algorithms makes it very tricky and time-consuming to learn a neural network policy from scratch on a real robot. Therefore researchers have sought to train policies in simulation and deploy it on real hardware by means of domain transfer (Pinto et al. (2017), Peng et al. (2017), Zhu et al. (2018)) or fine-tuning (Rusu et al. (2016), James et al. (2018)). The advantages of this approach include drastically increased learning speed through parallelization and distributed training, no human involvement during training, no necessity for safety measures and new algorithmic possibilities having access to simulator state information. Nevertheless, even in simulation the efficient exploration of the state space remains the crucial factor for successful learning. Naive Reinforcement Learning in difficult environments is likely to fail or to converge to suboptimal solutions. A number of approaches exist to help exploration, they are summarized in the following sections.

2.2.1 Reward Shaping vs Sparse Rewards

Reward shaping is one way to improve exploration by continuously providing the agent with information about how to get closer to the goal, e.g. via a function that encodes the distance between gripper and object. Popov et al. (2017) use a shaped reward function consisting of multiple weighted terms for a block stacking task similar

to ours, albeit only in simulation. However, defining a dense reward function is non-trivial for most tasks and requires domain knowledge as well as a significant amount of hand engineering (Andrychowicz et al. (2017)). Furthermore, it can bias the policy and prevent it from finding innovative solutions. For some tasks it might be even unknown how the desired behavior should look like, whereas it is mostly easy to define criteria for task completion.
Hence, using sparse rewards is usually preferred, even though other measures have to be taken to reduce sample complexity.

2.2.2 Hindsight Experience Reply

Hindsight Experience Replay (HER) (Andrychowicz et al. (2017)) is a widely used method that enables learning from sparse rewards in cases where actually no reward was received by storing unsuccessful trajectories with different goals in the replay buffer. This may be seen as constructing a curriculum implicitly without having to manually design it. HER can be used with any off-policy algorithm such as DDPG (Lillicrap et al. (2015)), examples of successful application to robotics can be seen in Nair et al. (2017), Peng et al. (2017) or Pinto et al. (2017). While HER works very well for improving precision in task like reaching or pushing, it is less effective for tasks with multiple stages and bottleneck moments like stacking, which is why we did not consider it suitable for our task setup. Moreover, when training end-to-end from images storing a transition with a different (visual) goal introduces a new challenge, although recent findings propose to extend HER to handle visual goals using a GAN (Sahni et al. (2019)).

2.2.3 Learning from Demonstrations and Curriculum Learning

Including human demonstrations into the training process is another widely used practise to reduce sample complexity . This can be implemented in several ways. One direct example is to use behavior cloning (BC) to pretrain the policy or augmenting the objective function with a BC loss. (Nair et al. (2017), Rajeswaran et al. (2017)).

Due to the compounding-error problem, the performance of BC strongly depends on the availability of a sufficiently large amount of demonstration data, which can be time-consuming to collect. Additionally BC requires demonstrations with aligned state and action spaces and it may bias the agent to overfit to suboptimal policies in case of an imperfect teacher.

Zhu et al. (2018) show that a handful of demonstrations are enough to learn a block stacking task with general adversarial imitation learning (GAIL) and PPO. Their approach shares several characteristics with ours, for instance training the policy with a combination of camera images and proprioceptive state vectors, using domain randomization (DR) for zero-shot sim-to-real transfer or leveraging exploration by restarting episodes from demonstration states, although their focus lies principally in combining imitation learning and RL, while we further explore curriculum learning. Their reported success rate of 35% for the block stacking task on a physical robot is significantly lower than our success rate of 67%, even if it's hard to compare results across different robots and varying task setups.

It is common practise in the RL robot manipulation research to overcome exploration difficulties by altering the start state distribution with states visited in demonstrations, either by uniform sampling (Popov et al. (2017), Nair et al. (2017), Zhu et al. (2018)) or by constructing a reverse curriculum sampling linearly backwards from the end of recorded trajectories (Fan et al. (2018)). However, none of these works fully exploit the possibilities of using demonstrations for curriculum generation. In contrast, our approach adaptively selects the appropriate difficultly of the start state distribution in order to build an optimized reverse curriculum. Florensa et al. (2017) propose another method for reverse curriculum generation that requires only a single state in which the task is achieved. The curriculum is generated gradually by sampling actions to move further away from the given goal and thus reversely expanding the start state distribution with increasingly more difficult states. Our approach is built upon the same key insight, that reaching the goal from most start states with random exploration is infeasible in reasonable amounts of time, whereas it is usually easy to

accomplish from states nearby a goal. Furthermore, their notion of "good starts", i.e. starts from where the agent sometimes receives a reward, but neither too often nor too rarely, inspired us to define bounds for the desired success rate. While Florensa et al. (2017) explicitly maintain a set of good starts, expanding it gradually by sampling nearby and removing samples that fall out of the desired reward thresholds, we adapt a difficulty parameter to sample start states that ensure expected rewards within the bounds. The authors demonstrate the ability of their method on simulated robot manipulation tasks such as *Key Insertion* or *Ring on Peg*. However, these tasks are in principal pure locomotion tasks that do not contain any reverse bottlenecks, which is why randomly sampling backwards in action space works well. For object manipulation tasks like block stacking, the inverse of the stacking trajectory would have to be randomly discovered. Being a difficult exploration problem by itself, one may assume that this would fail. In our case, demonstration trajectories are able to guide the exploration through theoretically arbitrarily difficult maneuvers.
An novel approach for curriculum learning is presented by Held et al. (2017), who show that a GAN can be trained to automatically generate goals of intermediate difficulty. Their framework is able to handle multimodal goal distribution, without requiring any prior knowledge about the task or the environment. Even though it is yet to be tested in a robotics experiment and it cannot handle visual goals so far, we still think it is an interesting approach and might investigate further in future work.

2.2.4 Sim-to-Real Transfer

Policies that were exclusively learned on simulated data usually cannot be straightforwardly applied to a physical robot due to the domain shift caused by sensor noise, modeling inaccuracies or unknown real-world dynamics. In this section we distinguish two major sim-to-real paradigms: zero-shot transfer and fine-tuning. In the former, policies are trained exclusively in simulation and run directly on the physical robot, while the latter involves additional real world training.
Previous zero-shot approaches like James and Johns (2016) tried to reduce the reality

gap by using highly realistic synthetic images from a 3D-simulator, but high-fidelity rendering does not only drastically slow down training time, it also narrows down the applicability of the policy to a very specific setup.

Tobin et al. (2017) and James et al. (2017) concurrently demonstrated that domain transfer greatly benefits from using visual domain randomization, i.e. randomizing visual features such as the camera position, lighting or object textures, because it comes at low cost and helps the policy learn scene invariant features. Especially the method of Tobin et al. (2017) is extensively used in robotics, for instance in Pinto et al. (2017) for learning zero-shot sim-to-real transfer of a block picking and block moving task on a 7-DOF Fetch robot. Our approach also uses scene and vision randomization, but we use slightly varying object colors instead of randomized object textures, similar to Zhu et al. (2018). Since our vision system operates in first-person view with the camera mounted on the robot flange, there is significantly less background clutter, thus more extensive DR would be unnecessary and can be even detrimental for the agent as it makes the learning problem much harder.

Apart from vision, also the dynamics and physical properties of the robot play an important role in sim-to-real transfer. Peng et al. (2017) show that by randomizing dynamics parameters (95 in total) such as mass, joint damping or action timestep, a policy can be robustly transferred to a real robot even under poor calibration and unfamiliar dynamics. They concentrate solely on non-visual domain transfer and therefore use a motion capture system to track the object location. Preliminary experiments have shown that dynamics randomization is important for successful policy transfer, but we do not randomize the setup to the same extent and do not feed the dynamics parameters to the value function.

Fine-tuning is another strategy to adapt policies to the real world. However, there is the risk of *catastrophic forgetting*, in which the performance degrades due to the destructing of previously learned features. Rusu et al. (2016) present a framework to bridge the reality gap using Progressive Nets, that mitigates some of the downsides of naive fine-tuning by retaining columns of originally learned features to prevent their destruction. They demonstrate on a block reaching task, that a policy pretrained in simulation, can be transferred to a real robot in less than 4 hours training time without using any domain randomization. Progressive nets prove to achieve higher scores than fine-tuning, while training from scratch on the real robot fails completely. So far our approach does not change the policy after the training in simulation is completed and it remains part of future work to investigate potential improvements using fine-tuning or progressive nets.

3 Background

3.1 Reinforcement Learning

We define our reinforcement learning (RL) setting as a discrete-time finite-horizon Markov decision process (MDP). An agent interacts with an environment that is described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0, T)$. At every timestep t, the agent is in state $s_t \in \mathcal{S}$, executes an action $a_t \in \mathcal{A}$, transitions to the next state s_{t+1} with the probability $\mathcal{P}(s_{t+1}|s_t, a_t)$ and receives a scalar reward $r_t(s_t, a_t) \in \mathcal{R}$. At the beginning of an episode, the initial state s_0 is sampled from the start state distribution ρ_0. In a finite-horizon setting, after T timesteps, the environment is reset to an initial state. Note that in order to simplify notation, we use the term state (i.e. the full underlying state containing all information) and state observation (e.g. a camera image) interchangeably and denote both with an s (Riedmiller et al. (2018)).

The agent selects actions according to a policy π, which can be either deterministic or stochastic. In the following, we will assume the policy to be stochastic, with action drawn as samples from a distribution over actions $\pi(a_t|s_t)$. We define the return R_t as the sum of discounted rewards $R(\tau_{t:T}) = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i))$, where $\tau_{t:T}$ denotes a trajectory $\tau = (s_t, a_t, ..., s_T, a_T)$ with actions $a_t \sim \pi(\cdot|s_t)$ and γ is a discount factor for future rewards. The goal of reinforcement learning is to learn a policy π to pick actions maximizing the expected return $\mathbb{E}_{s_0 \sim \rho_0, \pi}[R_0]$.

The difficulty of the problem is, that the agent does not know the environment at the beginning of the training and is unsure about which actions to take. Thus, it has to explore the environment to find the potentially sparse regions that yield a (high) reward. If the reward does not encode information about how to get to these goal states, exploration effectively becomes random search in the environment. While off-policy methods can use an arbitrary explorative strategy, on-policy methods use their current policy for exploration. As a result, off-policy methods can learn from past experience and tend to be more sample-efficient than on-policy methods, but can also be unstable when combined with function approximation and bootstrapping (Sutton and Barto (1998)).

A value function $V^{\pi}(s_t) = \mathbb{E}_{\pi}[R_t|s_t]$ is defined as the expected return from a state s_t when following policy π. Similarly, an action-value function $Q^{\pi}(a_t, s_t) = \mathbb{E}_{\pi}[R_t|a_t, s_t]$ is the expected return when taking action a_t in state s_t and then following the policy π. Using both definitions, one can define the advantage function $A^{\pi}(a_t, s_t) = Q^{\pi}(a_t, s_t) - V^{\pi}(s_t)$, which measures how much better or worse it is to take action a_t instead of the policy's preferred action.

3.1.1 Reward Functions

A reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is an arbitrary mapping from states and actions to a scalar. If the reward function is non-zero in only a sparse amount of states it is called a sparse reward. For the purpose of this work we define a sparse reward function as

$$r(s, a) = \begin{cases} \psi(s), & \text{if } s \in \mathcal{G} \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

where $\mathcal{G}$ denotes the set of goal states, in which the agent receives a reward and $\psi(s)$ defines the reward surface. In the simplest case $\psi(s) = 1$, which is called a binary reward. On the other hand, with a dense reward function (also called shaped reward), the agent constantly experiences a reward signal, that guides it to the goal, e.g. by means of a distance function.

3.1.2 Value-based Methods

Value-based methods do not directly learn a policy π, but they try to approximate an optimal value function $V^*(s)$ or $Q^*(a, s)$ and use it to reconstruct the policy. The policy can be obtained by selecting the action, for which the action-value function predicts the highest expected return with a certain probability and a random action otherwise, in order to account for the stochasticity of the policy.

3.1.3 Policy Gradient Methods

Policy gradient methods directly learn a parameterized stochastic policy $\pi_\theta(a_t|s_t)$ with parameters θ to maximize the expected return. These methods have the advantage, that they scale well to large or continuous action spaces. The goal is to find the parameters θ to maximize the objective function

$$\arg\max_\theta J(\theta) = \mathbb{E}_{s_0 \sim \rho_0}[R(\tau_{0:T})]. \tag{2}$$

One way to solve this is by taking a stochastic gradient ascent step in the direction of the estimated policy gradient $g \coloneqq \nabla_\theta J(\theta)$ (Schulman et al. (2016)). There are several ways to formulate the policy gradient estimator, the most commonly used form is

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t], \tag{3}$$

where $\hat{A}_t$ is an estimate of the advantage function at timestep t (Schulman et al. (2017)). In practise, both policy and advantage function are represented by neural networks with parameters θ and ϕ respectively. Such a method, that estimates both policy and value function, is also called actor-critic method.

3.2 Proximal Policy Optimization (PPO)

In the naive form, vanilla policy gradient methods often suffer from low sample-efficiency and high variance, which can lead to instability during training. Trust region methods have been proposed to improve the stability by constraining the size of the policy updates:

$$\begin{aligned} &\arg\max_{\theta} J(\theta) \\ &\text{s.t. } \hat{\mathbb{E}}[KL(\pi_{\theta_{old}}(\cdot|s_t) \parallel \pi_{\theta}(\cdot|s_t))] \leq \delta. \end{aligned} \tag{4}$$

The trust region radius δ is a constraint on the maximum KL-divergence between the current policy π_θ and the previous policy $\pi_{\theta_{old}}$ and prevents distructively large policy updates (Peng et al. (2018)). However, it can be difficult to optimize Eq. (4) directly and would require a second-order method. Proximal Policy Optimization (PPO) (Schulman et al. (2017)) replaces the hard constraint by a clipped surrogate objective

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{5}$$

where $r_t(\theta)$ denotes the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ between the current and the old policy, which can be interpreted as an approximation to the KL-divergence, because it also measures the similarity between the policies. The second term of the objective function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$ clips the probability ratio $r_t(\theta)$, which prevents the policy update from becoming too large, similar to a trust region method. The whole term serves as a lower bound on the unclipped objective that limits the change, but only in the case it would make the objective improve.
This enables PPO to speed up learning with multiple epochs of minibatch updates while still retaining stability, which would not be possible with standard policy gradient methods. At the same time, PPO is easier to implement than other trust region methods like trust region policy optimization (TRPO) and can be more generally applied (Schulman et al. (2017)).

3.3 Behavior Cloning

As opposed to reinforcement learning, imitation learning requires the availability of expert demonstrations in order to learn optimal behavior in the environment. A demonstration dataset $\mathcal{D}_{BC} = \{(s_i, a_i)\}_{i=1...N}$ consists of pairs of states and actions. Behavior cloning (BC) formulates the imitation learning problem as supervised learning by optimizing the loss function

$$L_{BC} = \sum_{i=1}^{N} \|\pi(s_i|\theta) - a_i\|^2. \tag{6}$$

Note that the learned policy can only be as good as the provided demonstrations and for a good performance, the size of the dataset has to be large enough.

4 Robot Setup

Our setup consists of a KUKA LBR iiwa robot equipped with a 2-finger parallel gripper and a camera mounted on the gripper (see Fig. 1). We present an easy-to-use Python user interface based upon the OpenAI Gym environment architecture (Brockman et al. (2016)) for real-time end-to-end control of the robot. Fig. 2 shows an overview of the setup.
Additionally we provide a modular OpenAI Gym reinforcement learning environment for training robots in simulation to learn manipulation tasks such as stacking. The policies that are learned in simulation can be deployed on the physical KUKA robot. This chapter will explain the various software and hardware components in detail.

4.1 Robot

The *KUKA LBR iiwa 14 R820* is a lightweight industrial robotic arm with seven revolute joints. Its control server runs the system software KUKA Sunrise.OS which provides the KUKA RoboticsAPI, a Java interface offering flexible high-level control functionality. Since there is no Python interface, we designed a Java wrapper library called `IIWAJavaController`. It works as an intermediary module, that receives actions from Python via UDP messages and calls RoboticsAPI functions. Our controller offers two distinct control modes: Joint position control and Cartesian position control. With the former, the desired position of all seven joints is sent to the robot, which allows for a maximum of flexibility, but also increases the difficulty of the learning problem. Therefore we only use joint position control to preposition the robot precisely at the beginning of a task.

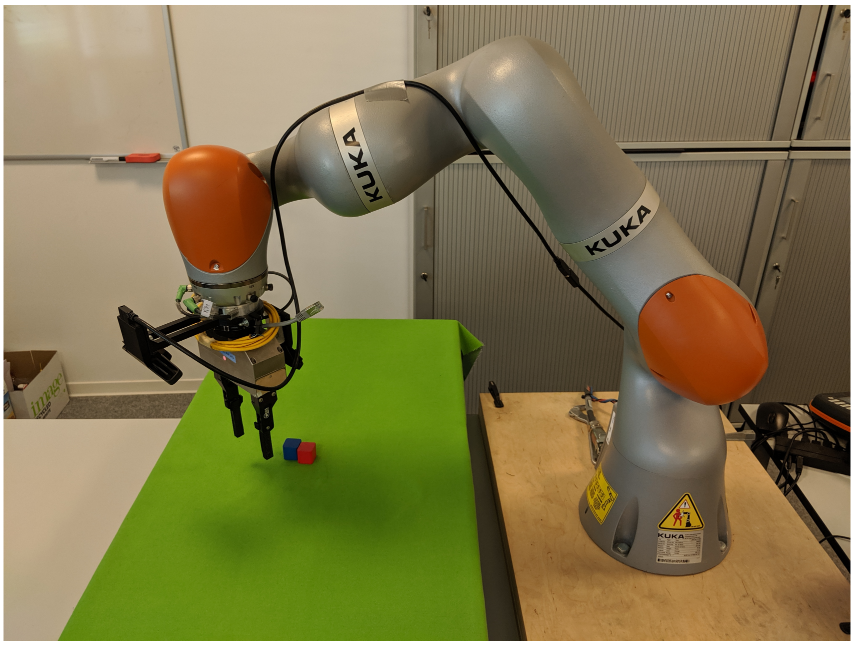


Figure 1: Photo of robot setup. It shows a KUKA LBR iiwa robot, equipped with a Schunk WSG 50 2-finger parallel gripper and an Intel Realsense camera mounted on the flange.

In Cartesian position control, the desired pose of the end effector is used to control the robot via inverse kinematics. For this purpose we make use of the *SmartServo* module of the RoboticsAPI. It features an impedance mode for a compliant behavior of the robot, implemented as a virtual spring damper system with configurable values for stiffness and damping. This is important for the safety of the robot and the workspace in case of unpredictable behavior of an RL policy. Additionally, we limit the orientation of the end effector to always face down, fixing pitch and roll as they are not necessary to solve our tasks. Thus, the robot's degrees of freedom are reduced to four, which significantly simplifies the task learning. Accordingly, an action message consists of three components for the X,Y and Z-coordinates and one component for the rotation around the Z-axis.

The `IIWAJavaController` can handle both absolute and relative coordinates. In our experiments with closed-loop policies learned by RL, we use relative control. Relative actions are specified with respect to the camera frame, which means that the

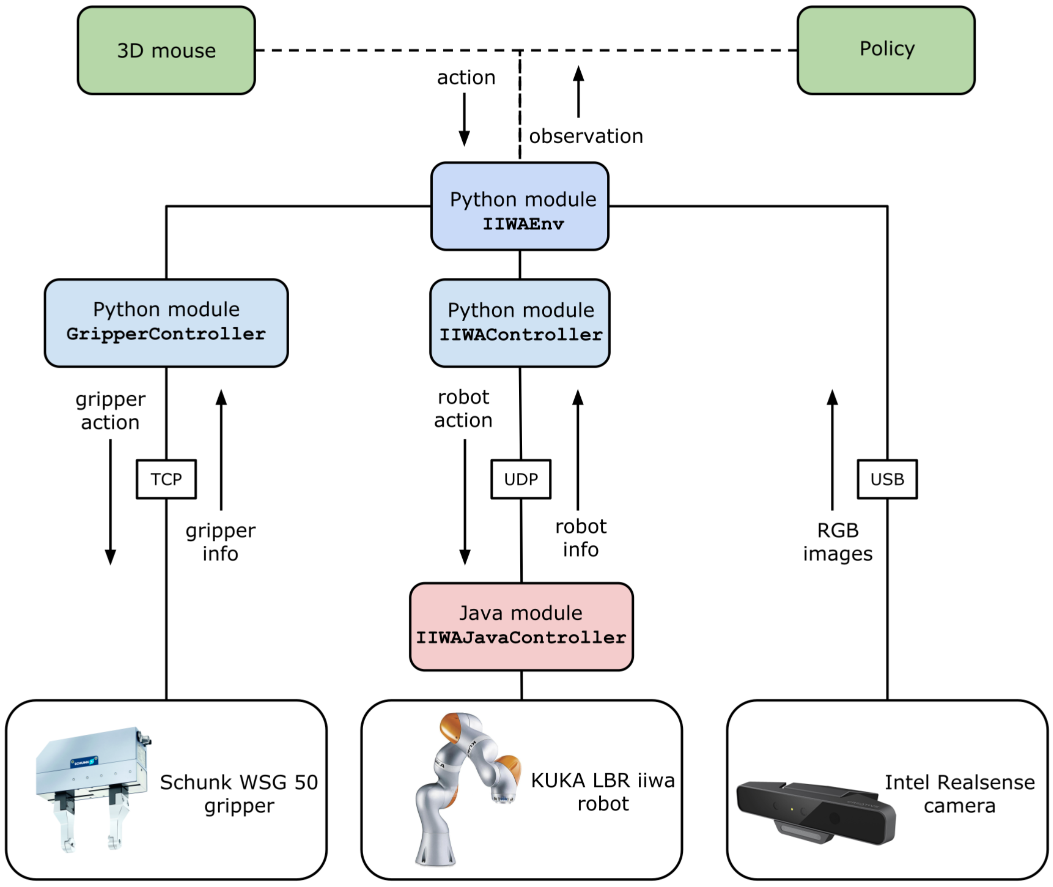


Figure 2: This figure shows the components of our modular robotic setup. The Python module `IIWAEnv` is the central interface of the system, using the OpenAI Gym environment architecture. It receives input actions either by a learned policy or manually by a 3D mouse, sends commands to robot and gripper and receives camera images.

X-coordinate of an action corresponds to a lateral movement perpendicular to the direction of the camera view. This is important since the camera coordinate system is the only reference frame of a vision-based policy.

In theory it would be beneficial to control the robot using velocity control, because many tasks require both fine-grained manipulation and straightforward locomotion over longer distances, but KUKA currently does not provide a velocity control interface. However, our implementation of relative Cartesian position control implicitly allows for variable velocities and can be seen as an approximation of velocity control. The range of the individual relative increments lies in $[-1, 1]$ cm for the position and in $[-0.2, 0.2]$ rad for the rotation of the end effector. The larger a component becomes, the faster the internal robot controller will accelerate the joint motors, because it plans to move along a longer trajectory. Action messages are sent with a fixed frame rate of 20 Hz. When a new action is received, the robot controller seamlessly switches over to pursue the current target position without stopping, which results in a smooth trajectory.

The `IIWAJavaController` also queries the robot's state and returns an info message containing the Cartesian end effector pose and the joint angles, which supplement the camera images as part of the observation for the RL agent.

4.2 Gripper

The robot is equipped with a *Schunk WSG 50* 2-finger parallel gripper. We mounted 3D-printed extensions to the gripper fingers to improve the camera perspective of the tool center point. The insides of the fingers are padded with a thin layer of rubber foam so that objects are less likely to slip away while being grasped.

For interfacing the gripper, we developed a custom Python module optimized for high-speed control using the WSG command protocol. It sends actions as binary messages via a TCP connection to the gripper and receives information about the gripper state, such as the opening width, in return.

Compared to the robot which receives continuous actions, the gripper is controlled in

Figure 3: Intel Realsense SR300 RGB-D Camera.

a discrete manner, i.e. an action can either open or close the fingers. If the output of a policy is continuous, values above or below a certain threshold are assigned to open or close the gripper respectively.
The intended usage of the gripper is executing pre-planned grasping trajectories, rather than doing high-frequency closed-loop iterative control as done by our reinforcement learning policy. As a consequence, the gripper imposes a significant delay, that has to be considered when transferring policies from simulation.

4.3 Camera

Our setup uses an *Intel Realsense SR300* RGB-D camera (Fig. 3). Currently only RGB images are being used for reinforcement learning, because the quality of depth images suffers from close ranges. We mounted the camera on the gripper flange with a 3D-printed camera mount to have an egocentric view of the scene. Compared to a static camera position, this allows being closer to the relevant parts of the scene, always focusing on the tool center point and having a perspective with less background clutter, which ultimately facilitates the sim-to-real domain transfer. The disadvantages of an egocentric camera position on the other hand are possible occlusions of task objects through gripper fingers and the risk of losing sight of objects. Additionally, a moving camera can result in blurred images, which can partially be compensated by shorter exposure times.
The camera provides a resolution of up to 1920×1080 pixels, but we downscale and

crop images to a size of 84 × 84 pixels to facilitate the training of RL networks. The camera is connected to the computer via USB and its driver offers Python bindings.

4.4 Workspace

To prevent damage to itself and the environment, the robot operates in a compliant workspace. It is mounted on a table, next to an area covered by a layer of rubber foam topped by a styrofoam board. This yields a plane surface for precise manipulation tasks, but still allows for a maximum of safety. Even in case of unforeseen behavior, a collision would not have damaging consequences. Furthermore, we define limits to constrain the allowed end effector positions within box-shaped boundaries of 50 × 30 × 17 cm, such that the end effector cannot touch the table and all locations within the limits are accessible for the robot.

4.5 3D mouse

The *3Dconnexion SpaceMouse Wireless* (Fig. 4) is used as an input device for both the real robot and the simulation. It seamlessly integrates in our robot environments and offers a precise control of robot and gripper, e.g. for the purpose of recording demonstrations or manually testing the feasibility of a task.

Figure 4: 3Dconnexion SpaceMouse Wireless

4.6 Python Gym Environment for Robot Control

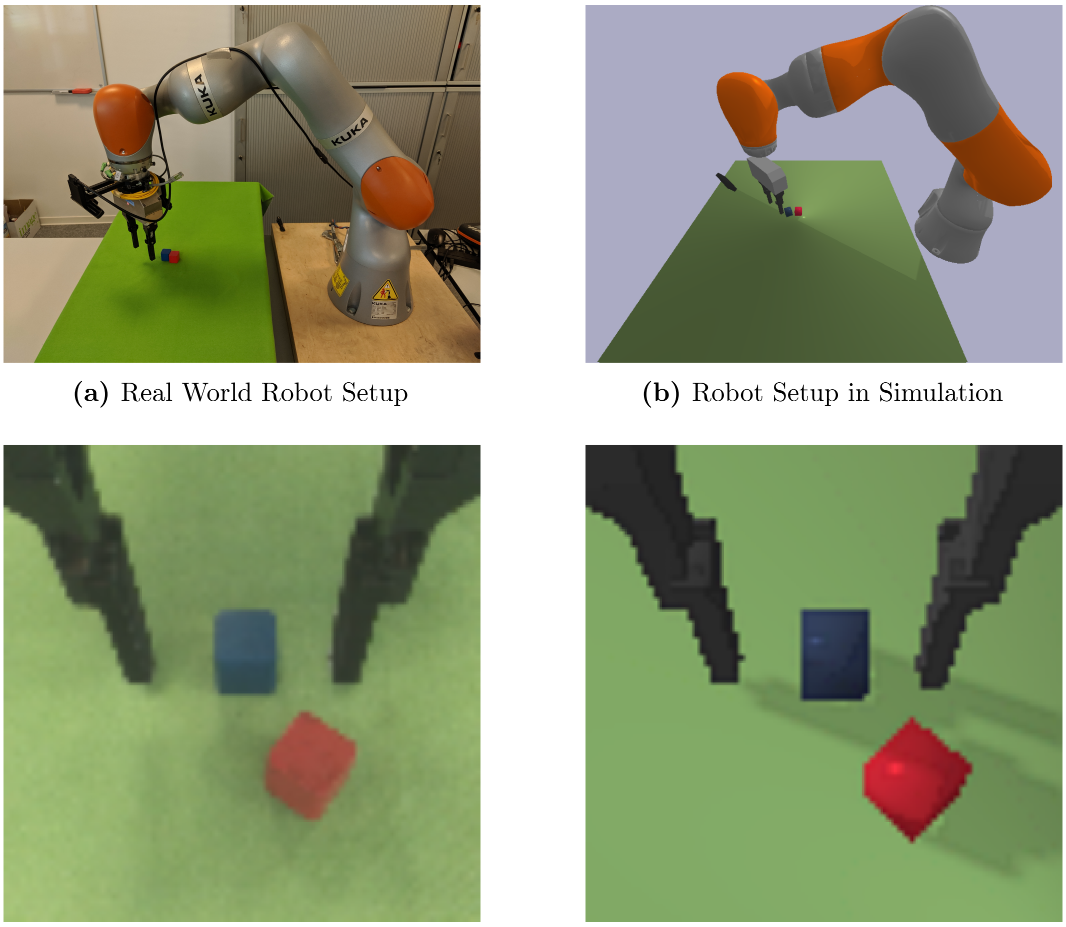
The Python module `IIWAEnv` is the central control interface of our setup. It adheres to the architecture of the OpenAI Gym environment for reinforcement learning. Thus, it can be used in combination with a large variety of RL frameworks. A trained reinforcement learning policy can be straightforwardly deployed on the robot just by calling the `step` function with the policy's action and feeding the obtained observation back into the policy. Additionally we provide code to control the robot with a 3D mouse instead, which is important for testing, data collection or designing new tasks. See Table 1 for an overview of the module.

4.7 Simulation

Training on the physical robot is infeasible due to the sample-complexity of RL algorithms. We created the `IIWASimEnv`, an OpenAI Gym training environment with a simulated KUKA iiwa robot using the PyBullet physics simulator (Coumans (2015)). PyBullet offers fast GPU rendering, optimized collision handling and an integrated inverse kinematics library. Several simulation instances can be run simultaneously allowing parallel training. Our Gym environment features a modular architecture with submodules for the definition of new tasks or the integration of different robot manipulators. The simulated setup closely resembles the real one, which facilitates the sim-to-real transfer of policies (see Fig. 5 for a comparison). The dynamics parameters and the exact camera position of the simulated robot are calibrated automatically to match the behavior of the real robot adequately. To further reduce the domain gap between simulation and reality, the environment includes flexible domain randomization of both the visual domain and the robot dynamics. For an overview of the randomized quantities, see Table 2. Task difficulties (e.g. the initial gripper height or the randomness of start states) and the degree of domain randomization can be individually set to adapt the complexity of the learning problem.

`def init`	Initialize environment Set up action and observation space Establish network connections to robot and gripper **inputs:** `dpos (0.01)` - position offset for relative Cartesian control `drot (0.2)` - rotation offset for relative Cartesian control `joint_vel (0.1)` - maximum joint velocity (except wrist joint) `gripper_rot_vel (0.3)` - maximum velocity for wrist joint `joint_acc (0.3)` - maximum joint acceleration `freq (20)` - frequency `act_type (continuous)` - action space (multi-discrete or continuous) `obs_type (img_state)` - observation space (image or image-state) `use_impedance (True)` - use impedance mode
`def step`	Call Python modules `GripperController` and `IIWAController` to send action message to gripper and `IIWAJavaController` via TCP and UDP respectively Get camera image and query robot and gripper state **inputs:** `action` - 5D numpy array in range $[-1, 1]$ **returns:** `observation` - camera image (and robot state)
`def reset`	Reset robot to fixed or random position Open gripper **inputs:** `reset_at_random_pos (False)` - reset at fixed or random position **returns:** `observation` - camera image (and robot state)
`def render`	Display camera image

Table 1: Methods of Python module `IIWAEnv`, an OpenAI Gym environment for controlling the real KUKA iiwa robot. Default values of inputs are in parentheses.



(a) Real World Robot Setup

(b) Robot Setup in Simulation

(c) Real World Camera View

(d) Camera View in Simulation

Figure 5: Comparison of the physical robot setup with the robot setup in simulation. Our approach uses the images from the wrist-mounted camera (**(c)** and **(d)**) as policy input.

4.8 Simulation Calibration

The goal of calibration is to make the simulation as similar to the real robot as possible. Our approach calibrates both the dynamics and the camera position of the simulation. This minimizes the domain gap that needs to be covered by domain randomization. However, there is no accurate open-source dynamics model of the KUKA iiwa robot publicly available and it would be tedious to do a full model identification without having insight into the KUKA RoboticsAPI.
As a workaround, we tune a small set of parameters that are primarily responsible for the dynamics of the robot in simulation:

- Maximum position change for relative Cartesian position control
- Maximum rotation change for relative Cartesian position control
- Joint velocity of joints 1-6
- Joint velocity of wrist joint
- Velocity of gripper fingers
- Gripper delay

As a first step, we record a random demonstration trajectory on the real robot and manually precalibrate the parameters, such that replaying the same actions in simulation leads to a coarsely similar trajectory. Then, we fine-tune the parameters iteratively with differential evolution (Storn and Price (1997)), optimizing the trajectory similarity. The resulting calibration does not exactly replicate the real robot but because the policies are learned with closed-loop control, they are inherently self-correcting. In addition, since we train the simulation with dynamics randomization (see Chapter 5), the policies become to some extend invariant of a specific calibration.
For the calibration of the camera pose and intrinsics, we apply a similar procedure and also use a mixture of manual pretuning and differential evolution.

5 Method

We present Adaptive Curriculum Generation from Demonstrations (ACGD), a simple framework to overcome exploration difficulties of Reinforcement Learning from sparse rewards in simulated environments. It requires only a handful of human demonstrations to generate a reverse curriculum of start states for the RL agent. In order to achieve fast learning speeds, ACGD adaptively schedules more and more difficult subtasks and gradually increases the degree of domain randomization. This allows learning complex robot control policies end-to-end in simulation that transfer to the real world without any fine-tuning or training on the physical robot (zero-shot transfer).
We demonstrate the potential of our method on a real-world block stacking task (see Chapter 6). Fig. 6 illustrates the overview of our robot learning method.

5.1 Assumptions

Our method is motivated by the following assumptions:

1. A policy trained with a limited amount of task instances will provide a good initialization for learning to solve similar task instances (see Held et al. (2017)).

2. A small set of human demonstrations should be easily obtainable in simulation and provide efficient guidance.

3. The training environment can be reset to an arbitrary state of the demonstration trajectories.

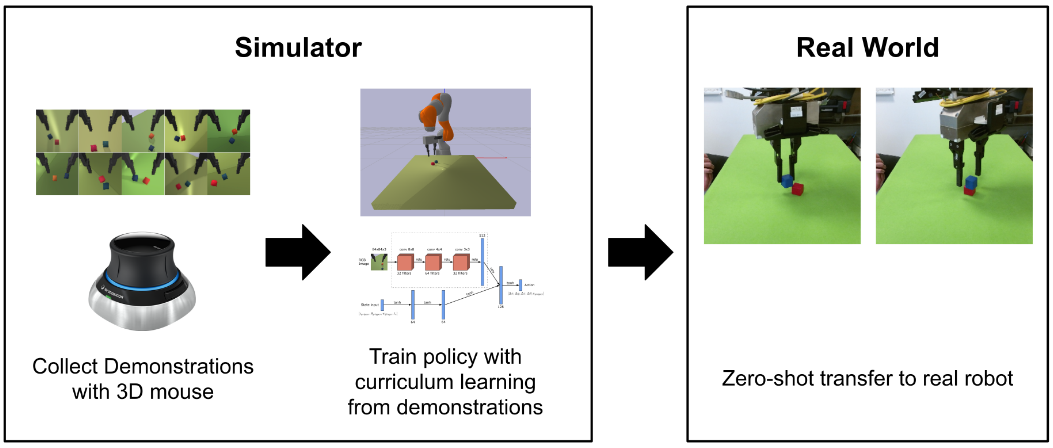

Simulator
Collect Demonstrations with 3D mouse
Train policy with curriculum learning from demonstrations
Real World
Zero-shot transfer to real robot

Figure 6: We overcome exploration for robot manipulation tasks with sparse rewards, by collecting a handful of human demonstrations and training a policy in simulation with curriculum learning from demonstrations and domain randomization. The learning is bootstrapped through resets to states visited by demonstrations. The resulting policy can be run on the real robot without further fine-tuning and achieves a success rate of 67% in a block stacking task.

4. The agent is more likely to experience a reward starting from a state taken from the end of a demonstration, than from a state taken from the beginning.

5. There exists a set of environment parameters that allow to gradually increase the task difficulty during training. This may include parameters like the initial gripper height or the degree of domain randomization.

5.2 Curriculum Generation from Demonstrations

5.2.1 Collecting Demonstrations

The first step of our approach is to record a set $\mathcal{D} = \{s_t\}_{t=1...T}$ of human demonstration trajectories of the task in simulation, where s_t denotes the full state of the environment at time t. This can be done e.g. by using a 3D mouse to control the robot's end effector. It is not necessary for the demonstrations to be optimal, since the method only uses the recorded states to create a curriculum of start states. Demonstration actions are discarded and will not prevent the learned policy from surpassing human performance. For instance, our learned policies are able to stack much faster than the human teachers. In contrast to behavior cloning methods that need a large batch of demonstrations to work well, our method does not require more than a dozen demonstrations.

5.2.2 Reverse Curriculum Learning

The intuition behind our method is as follows: At the beginning of training the agent does not have any knowledge about the task and without a dense reward function, there is no information about which actions it should take. The probability of experiencing a reward by random exploration depends on the amount of actions the agent needs to reach a goal state.
If the agent starts exploring the environment from a state $s_{t \leq T} \in d$ of a demonstration trajectory $d \in \mathcal{D}$ of length T, it is possible to reach the goal in at least $T - t$ steps. At the beginning of a training, we sample start states from the end of the demonstrations where we expect to obtain rewards by uninformed exploration. In the course of the training, we gradually shift the sampling region towards the beginning of the demonstration trajectories until the start state distribution coincides with the original initial state distribution ρ_0. We call this a reverse curriculum because the policy learns to solve the task backwards, starting in the vicinity of goal states and learning to master increasingly more difficult subtasks until the complete task is learned.
Whereas a policy trained with behavior cloning tries to mimic demonstrator actions as closely as possible and is prone to inherit suboptimal behavior, a policy trained with reverse curriculum learning is still free to find innovative solutions to the task. We show this empirically by comparing policies which, despite being trained on the same demonstration data, develop completely different block stacking strategies.
As opposed to Fan et al. (2018), who claim that it is important for reverse curriculum learning to include previously sampled sections throughout the training in order to prevent catastrophic forgetting, our experiments suggest that this is not the case.

Starting the exploration from a new set of demonstration start states, the agent is likely to revisit previous start states, from where it already learned to reach the goal.

5.2.3 Adaptive Curriculum Generation

The challenge of curriculum learning from demonstrations is to decide a good strategy to choose the appropriate difficulty of start states. Previous approaches sampled states uniformly (Popov et al. (2017), Nair et al. (2017), Zhu et al. (2018)) or linearly backwards (Fan et al. (2018)).
However, sampling states from the end of the demonstration trajectories for too long unnecessarily slows down the training because the policy is trained on subtasks that it has already learned to master. On the other hand, sampling more difficult start states too fast may prevent the policy from experiencing any reward at all.
Intuitively, we want the probability of experiencing a reward to be neither too high, nor too low. Instead, our goal is to confine the probability within the interval $[\alpha, \beta]$: We want to choose t such that

$$\alpha \leq \mathbb{P}(R_t > 0|\pi) \leq \beta, \tag{7}$$

where R_t denotes the return of a rollout started from a state sampled from the demonstration data at timestep t. This is inspired by the *Goals of Intermediate Difficulty* in Held et al. (2017). For sparse reward functions, the probability $\mathbb{P}(R_t > 0|\pi)$ corresponds to the expected success rate of the task.
We introduce the parameter $\delta_d \in [0, 1]$ for setting the difficulty of the demonstration resets, where a δ_d close to 0 corresponds to easier start states sampled from the end of the demonstration trajectories and a δ_d close to 1 corresponds to earlier start states. At the beginning of the training, δ_d is initialized with 0. By recording the last n training returns, we are estimating the expected success rate of Eq. (7), which will increase, as soon as the policy starts to learn. If at one point the success rate sr_d surpasses the upper bound β, we increase the difficulty of the task by adding a small

ϵ to δ_d. Likewise, if the policy over a period of n episodes on average fails to achieve higher returns than the lower bound α, we subtract ϵ from δ_d:

$$\delta_d^{(i+1)} = \begin{cases} \delta_d^{(i)} + \epsilon, & \text{if } sr_d > \beta \\ \delta_d^{(i)} - \epsilon, & \text{if } sr_d < \alpha \\ \delta_d^{(i)}, & \text{otherwise.} \end{cases} \tag{8}$$

This update rule allows the policy to adaptively choose the best learning speed throughout the training. Especially tasks with long episode lengths often do not have a constant difficulty at every stage. Consider for instance a stacking task: it consists of both easier parts that only require straight locomotion and more difficult bottleneck moments like grasping and placing the object. By adaptively generating a curriculum of demonstration start states, the learning algorithm automatically dedicates more time on hard parts of the task, while not wasting time at straightforward sections.

5.2.4 Mixing Demonstration Resets and Regular Resets

Since there are possibly infinitely many random task configurations (e.g. random initial object positions) and the start states are drawn from only a handful of demonstrations, there is the risk of overfitting to the demonstration data set. Therefore we mix demonstration resets with regular resets from the original start state distribution ρ_0. We call the mixing ratio $\lambda \in [0, 1]$, it is a hyperparameter that controls the probability of doing a regular reset.
Intuitively, we want λ to grow during training. Initially, the curriculum schedules start states sampled from the end of the demonstrations. Adding regular starts at this point would most likely result in the agent not receiving any reward. Later, if the curriculum samples earlier starts and the policy learns to master longer sections of the task, there is also an increased probability of reaching the goal from an original initial state.

At iteration i of in total N iterations, we empirically choose λ to be

$$\lambda^{(i)} := 0.5(sr_r + (i/N)), \tag{9}$$

where sr_r denotes the success rate of episodes initialized with regular resets and i/N is the training progress.

5.3 Adaptive Task Difficulty and Domain Randomization

Apart from the distance between start states and goal states, the difficulty of a task also depends a set of task specific factors, such as the degree of domain randomization, intrinsics of the physics simulator or criteria that define the task completion. As an example, the complexity of block stacking depends significantly on the bounciness of the blocks. Therefore, we design our tasks such that their difficulty can be controlled by a set of parameters $\mathcal{H}$ with each parameter $h \in [0, 1]$. See Table 2 for a list of parameters that control the difficulty of the stacking task.

In addition to the purpose of determining the difficulty of start states, we also use the parameter δ_d to set the relative difficulty of the task parameters $\mathcal{H}$. Since we want to set the difficulty of regular resets independently of the difficulty of demonstration resets, we introduce a second difficulty parameter δ_r and update it analogously to Eq. (8) according to the success rate sr_r.

At every reset, we adapt the task difficulty parameters:

$$\text{for each } h \in \mathcal{H} : h = \begin{cases} \delta_d & \text{if demonstration reset} \\ \delta_r & \text{if regular reset.} \end{cases} \tag{10}$$

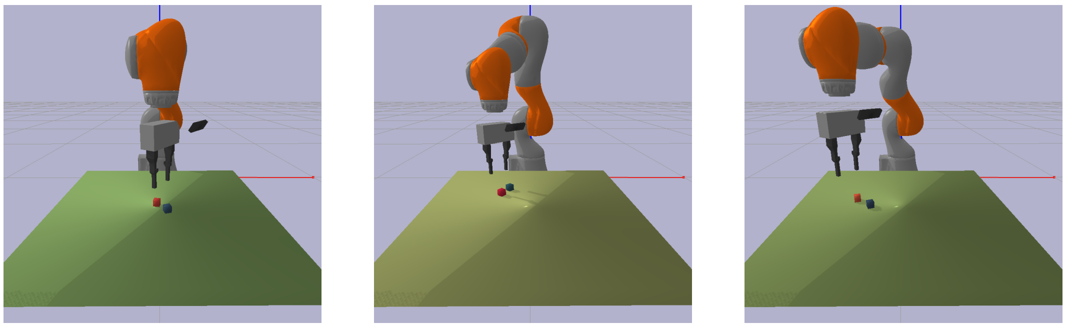
When resetting from demonstrations, some parameters (e.g. the initial gripper height) are already predetermined by the demonstration state and will not be affected by the adaptive difficulty of Eq. (10).

The task parameters $\mathcal{H}$ can be split into two groups. First, there are parameters that directly control the task difficulty, for instance the bounciness of objects or the initial height of the gripper. Changing them has an immediate impact on the complexity, e.g. by increasing the minimum amount of steps that are necessary to solve the task or by making the criteria for task completion harder. Secondly, there are parameters that control the domain randomization and only contribute indirectly to the task difficulty. If we were only focusing on learning a task in simulation, it would not be beneficial to introduce randomness into the environment, but since the goal is to transfer policies to the real world, domain randomization is essential for a successful policy transfer. It helps the policy to learn robust, meaningful features, makes it less prone to overfitting and reduces the need for exact calibration of the robot's dynamics. Previous methods have demonstrated successful applications of visual domain randomization (Tobin et al. (2017)) and dynamics randomization (Peng et al. (2017) to the field of robotic manipulation.

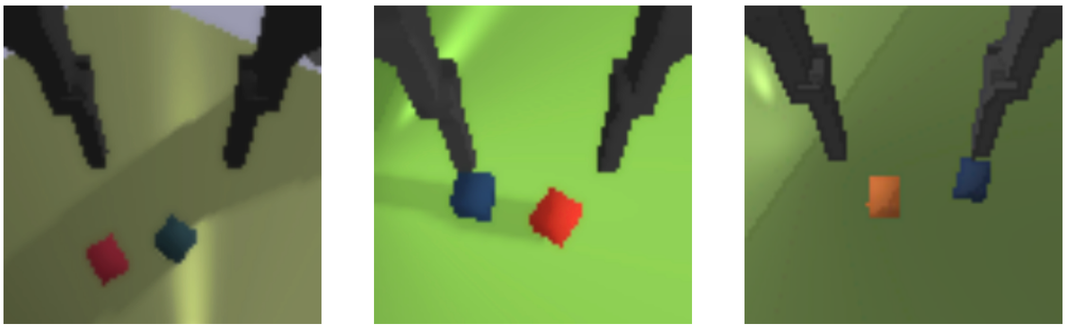
We randomize the visual appearance of scene and objects (e.g. color, size, position) as well as the robot's dynamics. In addition, we further postprocess the camera image with standard image enhancement such as changing hue and saturation or adding blur. The parameters that we use for domain randomization are listed in Table 2.

In contrast to the first group of task parameters, these properties do not directly influence the difficulty of the task. Instead, it is their degree of randomness, i.e. how much a property can vary, that determines the difficulty. Too much domain randomization might prevent the policy from learning the task at all.

Therefore we apply the concept of curriculum learning also to domain adaptation. Our experiments clearly suggest, that it is beneficial to increase the degree of domain randomization during training with the parameters δ_d and δ_r. By doing that, the policy adapts to domain randomization gradually without being overburdened. At the beginning of the training when the policy network still has not learned any features, we set domain randomization to a minimum. The expectation is, that in this easy setting, the policy will start to learn fast. As soon as the success rate rises, our algorithm increases the difficulty parameters, which in turn has a direct effect on the



(a) Scene View (not used for training)



(b) Gripper View

Figure 7: Domain randomization for the stacking task. The images show instances of different initial states. We randomize camera properties (pose and FoV), object properties (position, size, color), scene properties (table position and color, robot pose, light source) and do a randomized image postprocessing). Note that **(a)** only serves as an illustration of different task setups and does not feature the full visual domain randomization. Our approach uses only egocentric camera images from **(b)**. See Table 2 for a complete list of randomized parameters.

range of randomness in the environment.

To our knowledge, we are the first to use curriculum learning for domain adaptation. See Fig. 7 for an illustration of initial states and observations under domain randomization.

5.4 Algorithm

Our algorithm is described in detail in Algorithm 1. It can be used in combination with any on-policy reinforcement learning algorithm, in our experiments we use PPO to update the policy. Apart from the typical PPO hyperparameters, the hyperparameters that have to be chosen for our method are the desired success interval $[\alpha, \beta]$, the difficulty increment ϵ and the width of the window for sampling demonstration states σ.

5.5 Sim-to-Real Transfer

We perform zero-shot policy transfer to the real world, this means that the policy is trained entirely in simulation and run on the real robot without any further fine-tuning or training. When comparing simulation and real world, there is a noticeable domain gap for the visual domain and a discrepancy in the physical behavior of the robot. The environment in simulation only coarsely matches the real task setup and PyBullet's rendering engine is lacking in detail. Depending on the weather and time, the lighting conditions and visual appearance of the real world can change significantly. Due to KUKA's RoboticsAPI being closed source, it is tedious to do a full model identification for the robot's dynamics. Hence, we only roughly calibrated the simulated robot, which is explained in Chapter 4. Besides that, PyBullet is optimized for performance, rather than being a highly realistic physics simulation.
The key factor for a successful policy transfer is domain randomization because it forces the policy to learn domain invariant features. Our approach works on the real robot without any object detection, tracking or position information other than the unprocessed camera image and the proprioceptive state of the robot manipulator itself. The policy is learned end-to-end, with a single neural network that takes raw pixels and the robot state as input and directly outputs actions for the robot controller. We demonstrate the ability of our method in the experiment section by learning a block stacking task, for which we a achieve a success rate of 67%.

Algorithm 1 : Adaptive Curriculum Generation from Demonstrations

```
Algorithm train_policy:
     Input  : Iterations $N$, initial policy $\pi_0$, interval $[\alpha, \beta]$, increment $\epsilon$
     Output : final policy $\pi_N$
1    $sr_d, sr_r, \delta_d, \delta_r \leftarrow 0$;
2    for $i \leftarrow 1$ to $N$ do
3        $rollouts \leftarrow$ generate_rollouts();
4        $\pi_i \leftarrow$ update_policy($rollouts, \pi_{i-1}$);
5        update success rate $sr_d$;
6        update success rate $sr_r$;
         /* update difficulty parameters */
7        $\delta_d^{(i+1)} = \begin{cases} \delta_d^{(i)} + \epsilon, & \text{if } sr_d > \beta \\ \delta_d^{(i)} - \epsilon, & \text{if } sr_d < \alpha; \\ \delta_d^{(i)}, & \text{otherwise} \end{cases}$
8        $\delta_r^{(i+1)} = \begin{cases} \delta_r^{(i)} + \epsilon, & \text{if } sr_r > \beta \\ \delta_r^{(i)} - \epsilon, & \text{if } sr_r < \alpha \\ \delta_r^{(i)}, & \text{otherwise} \end{cases}$
     end

Function generate_rollouts:
     Input  : Batchsize $B$, policy $\pi_i$, success rate $sr_r$, training progress $prog$,
              difficulty $\delta_d$, difficulty $\delta_r$, demonstrations $\mathcal{D}$, episode length $T$,
              window size $\sigma$, env params $\mathcal{H}$
1    $rollouts \leftarrow []$;
2    for $j \leftarrow 0$ to $B$ do
3        sample action $a_j \sim \pi_i$;
4        execute action $a_j$ and receive state $s_{j+1}$, observation $o_j$ and reward $r_j$;
5        $rollouts$.append($[s_{j+1}, o_j, r_j]$);
6        if terminal $s$ then
7            with probability $p = 0.5(sr_r + prog)$ do
                 /* reset from demonstrations */
8                for each parameter $h \in \mathcal{H}$ : set $h \leftarrow \delta_d$;
9                sample demonstration $d \sim D$;
10               sample timestep $t \sim \text{unif}((1 - \delta_d)T - \sigma, (1 - \delta_d)T + \sigma)$;
11               set start state to demonstration: $s_{j+1} \leftarrow d_t$
12           otherwise
                 /* regular reset */
13               for each parameter $h \in \mathcal{H}$ : set $h \leftarrow \delta_r$;
14               sample start state from start state distribution: $s_{j+1} \sim \rho_0$;
     end
     return $rollouts$;
```

6 Experiments

In this chapter, we demonstrate the ability of our approach to solve complex robot manipulation tasks with only a handful of demonstrations. This is exemplarily shown by means of a block stacking task. We compare the performance of our method with various baselines and conduct several ablation studies to assess the relative importance of the individual components of our method. The main questions we are answering in this chapter are:

- How does Adaptive Curriculum Learning from Demonstration compare to existing curriculum learning methods and other baselines?
- How does curriculum learning for domain randomization affect the learning speed and final performance?
- How well do policies learned exclusively in simulation transfer to the real robot?

Before addressing these questions, we describe the task setup in detail.

6.1 Stacking Task

Block stacking is a popular benchmark task for robot manipulation and has been examined by previous work, e.g. in Deisenroth et al. (2011), Popov et al. (2017), Zhu et al. (2018) and Riedmiller et al. (2018). To solve the task, the agent has to stack one block on top of the other one, this involves several important subproblems such as reaching, grasping and placing objects and features complex, contact-rich interactions, that require substantial precision. Since it is a task of long episode

length that requires at least around 100 steps to be solved with our choice of per-step action ranges, it is particularly well suited for curriculum learning.
We use the robot setup described in Chapter 4 and only explain the task specific parts in this section. The task setup consists of a table with two small, square, colored blocks that are randomly placed in the workspace's center.
The stacking task is implicitly parameterized by a large number of parameters, for the purpose of constructing a training curriculum we separate those that have a sufficient contribution on the task difficulty in a set $\mathcal{H}$. See Table 2 for a complete list of the task difficulty parameters.

6.1.1 Goal Definition and Reward Specification

Defining what constitutes a successful stack is not trivial, it involves a trade-off between certainty of stability and measurement time. We consider a stack as successful, if for a period of 10 consecutive timesteps, the blue block is in contact with the red block, but not with the table nor with the gripper and the velocity of the blue block is below a certain threshold after the 10 timesteps. The latter turned out to be an important indicator for the stability of a stack, because it is possible that the upper block falls down, even after being on top of the other one for more than 10 steps. The agent has 150 timesteps to complete the task. The sparse reward function used for the experiments is defined as:

$$r_t = \begin{cases} 1 - \phi, & \text{if stack successful} \\ 0, & \text{otherwise,} \end{cases} \tag{11}$$

where $\phi \in (0, 1)$ is a penalty for the movement of the bottom block during the execution of the task. See the Appendix for a definition of ϕ.

Parameters	**Parameter directly controls task difficulty**	**Parameter controls DR**	**Parameter affects demo reset**	**Parameter affects regular reset**
Initial gripper height	✓			✓
Lateral gripper offset	✓			✓
Distance between blocks	✓			✓
Bounciness of objects (physics parameter)	✓		✓	✓
Num. steps stacked for task success	✓		✓	✓
Min. final block vel. for task success	✓			✓
Table height		✓		✓
Height of the robot base		✓		✓
Initial gripper rotation		✓		✓
Block size		✓		✓
Gripper speed		✓	✓	✓
Position offset of relative Cartesian position control		✓	✓	✓
Camera field of view (FOV)		✓	✓	✓
Camera position		✓	✓	✓
Camera orientation		✓	✓	✓
Block color (varying hue)		✓	✓	✓
Table color		✓	✓	✓
Camera image brightness		✓	✓	✓
Camera image contrast		✓	✓	✓
Camera image hue		✓	✓	✓
Camera image saturation		✓	✓	✓
Camera image sharpness		✓	✓	✓
Camera image blur		✓	✓	✓
Light direction		✓	✓	✓

Table 2: Task parameters. These are the parameters $\mathcal{H}$ that are used to adapt the task difficulty of the stacking task. They can be split in two groups: those that directly control the task difficulty and those that control the domain randomization and only contribute indirectly to the task difficulty. Some parameters are already predetermined by a demonstration reset and only affect regular resets.

6.1.2 Environment Observations

The observations that the policy receives are a combination of the camera image and a state vector. The image observation I_t is an RGB image of 84×84 pixels from a camera mounted on the robot flange. Additionally, we use a proprioceptive state vector that consists of:

- $z_{gripper}$, the gripper height above the table
- $\theta_{gripper}$, the angle that specifies the rotation of the gripper
- w_{finger}, the opening width of the gripper fingers
- t_e, the remaining timesteps of the episode.

We normalize the state vector to the range $[0, 1]$ Both observation modalities together form the full observation:

$$o_t = \{I_t, [z_{gripper}, \theta_{gripper}, w_{finger}, t_e]\}. \tag{12}$$

Consistent with results of Kalashnikov et al. (2018), our experiments show that it is essential for the performance of the policy to include the propriopceptive features. For a discussion, see Section 6.3.4.

6.1.3 Action Space

We control our robot with Cartesian position control using inverse kinematics.
An action a_t is a 5 dimensional continuous vector

$$a_t = [\Delta x, \Delta y, \Delta z, \Delta\theta, a_{gripper}]. \tag{13}$$

The first three entries are relative position changes for the Cartesian controller, defined with respect to the camera frame. The fourth entry corresponds to the yaw rotation of the gripper and the last entry is the gripper action. Action outputs are clipped to the range $[-1, 1]$. Since actions are continuous, but the gripper requires discrete actions, $a_{gripper}$ is discretized to $\{-1, 1\}$.

6.2 Training Details

6.2.1 RL Algorithm

The policy network is trained using Proximal Policy Optimization (PPO). We use a *PyTorch* (Paszke et al. (2017)) implementation of the algorithm (Kostrikov (2018)). However, our approach can be used with an any on-policy reinforcement learning algorithm. The training runs 8 environments in parallel for a total number of 10^7 timesteps, this takes approximately 11 hours on a system with one GPU and 16 cores. With a batch size of 512 per environment, this equals 2440 PPO updates. Every 20 update steps, we evaluate the performance of the policy on the maximum task difficulty in order to obtain comparable results between different methods. For having more reliable results, the final evaluation is done with 100 episodes.
The default parameters for our algorithm are $\epsilon = 0.002$ and $[\alpha, \beta] = [0.4, 0.6]$. We use the *Adam* (Kingma and Ba (2015)) optimizer with an initial learning rate of 2.5×10^{-4}, which is linearly decayed in the course of the training. See Table 3 of the Appendix for a complete list of hyperparameters that are used for the experiments.

6.2.2 Network Architecture

The neural network consists of two separate input streams for the different observation modalities. The image observations of size $84 \times 84 \times 3$ are passed through a convolutional neural network (CNN) of 3 layers with rectified linear unit nonlinearities (RELU), followed by a fully-connected layer with 512 units. The CNN uses the network architecture of Mnih et al. (2015). The first convolutional layer has 32 filters of size 8×8 with a stride of 4, followed by a layer of 64 filters of size 4×4 with stride 2 and a last layer with 32 filters of size 3×3 with stride 1.
The state observations are processed by a multilayer perceptron (MLP) with 2 layers

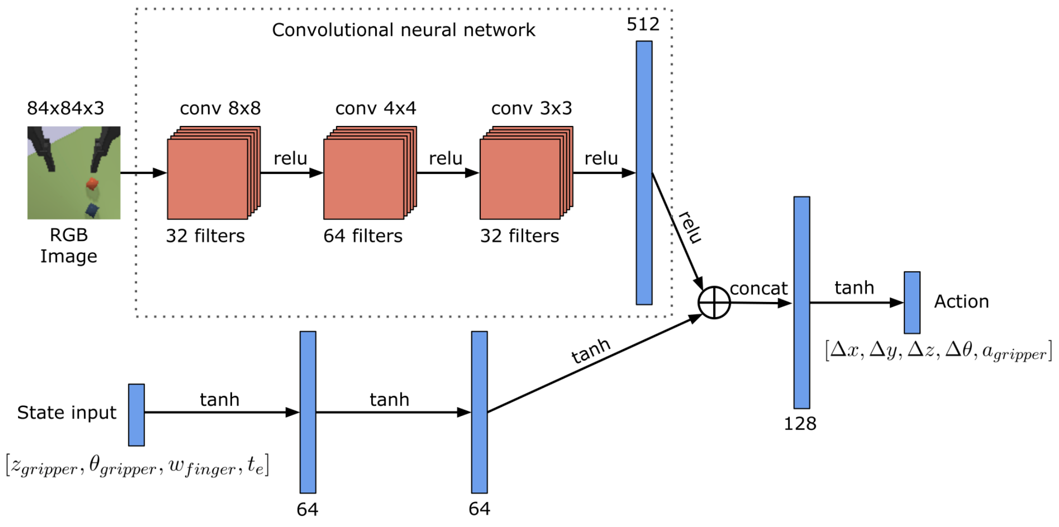

Convolutional neural network
84x84x3
RGB Image
conv 8x8
32 filters
relu
conv 4x4
64 filters
relu
conv 3x3
32 filters
relu
512
relu
concat
tanh
Action
$[\Delta x, \Delta y, \Delta z, \Delta\theta, a_{gripper}]$
128
State input
$[z_{gripper}, \theta_{gripper}, w_{finger}, t_e]$
tanh
64
tanh
64
tanh

Figure 8: Architecture of the policy network. The policy takes camera and state observations as input and outputs actions for the robot controller. Convolutional layers are shown in red, fully-connected layers in blue. The value function has the same architecture apart from having a single output for the value. Policy and value function share the weights of the CNN (dotted box).

of 64 units each and tanh nonlinearities.

Then, both input streams are concatenated and passed through another MLP with 128 layers and tanh nonlinearities, followed by an output layer.The policy output is the mean and standard deviation of a diagonal Gaussian distribution over the 5-dimensional continuous actions. The value function outputs a scalar value. Policy and value function have the same architecture and share weights for the CNN part of the network, the rest is separate for both. See Fig. 8 for an illustration of the network architecture.

6.2.3 Demonstrations

We recorded a set of 10 task demonstrations with a 3D mouse and stored the PyBullet simulator state for every timestep, this allows us to reset the environment to an arbitrary step of the demonstration dataset. At each reset, the parameters defining the domain randomization can be resampled. We gave the teacher 200 timesteps to finish an episode, but only use the first 150 steps, because we do not want to reset from an already solved state. For the behavior cloning baseline, we gathered 100 demonstrations with states and actions.

6.3 Experiments in Simulation

All experiments were done on the stacking task in simulation with the same hyper-parameter setup and the same network architecture, unless stated otherwise. Every experiment was run five times with different random seeds to account for the large variance of RL. The plots show the mean and standard deviation of the evaluation episodes across all seeds, which were always run on the highest (i.e. final) task difficulty. Fig. 9 shows how the training success rates and the difficulty parameters correspond to each other during training.

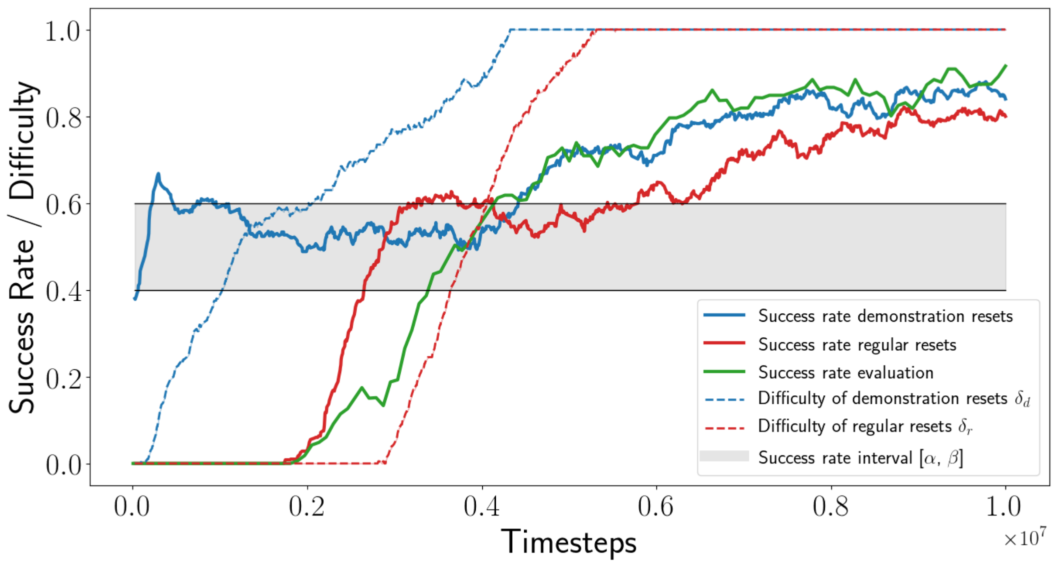


Figure 9: This plot illustrates the increase in difficulties (δ) during training depending on the success rates. The difficulty of demonstration resets δ_d is increased, if the success rate of episodes with demonstration resets exceeds the defined threshold (gray area). The same happens for regular resets (δ_r). The experiment uses a default interval of $[\alpha, \beta] = [0.4, 0.6]$. Our algorithm adapts the δ parameters such that the success rates of demonstration resets and regular resets stay within that interval, this leads to a faster learning speed.

6.3.1 Baseline Experiments

In this experiment we compare the performance of our method with standard RL baselines without curriculum learning.

Ours (ACGD): Full method, adaptive curriculum learning from demonstrations with adaptive task difficulty.

RL sparse: RL without demonstrations or adaptive task difficulty.

RL shaped: RL without demonstrations or adaptive task difficulty, but with dense rewards. The reward function consists of the euclidean distance between gripper and the blue block and the euclidean distance between the blue block and a point above the red block. Additionally, there is an intermediate sparse reward given, when the blue block is successfully lifted off the ground and a final sparse reward, when the task is completed.

BC init + RL sparse rewards: The policy is pretrained with behavior cloning and used as an initialization for RL with sparse rewards.

Results

Fig. 10 shows the results of the experiment. Our method was the only one that was able to successfully learn the task. It achieves an average final success rate of 91% and an average final reward of 76%. The rewards are lower than the success rate, because the policies do not learn to always avoid touching the red block, which causes a small reward penalty. Interestingly, the training with multiple random seeds led to the emergence of distinctive behavior. Some policies keep the gripper's rotation to a minimum and rather move it backwards and sideways, while others first rotate the gripper and move it forward afterwards. This suggests, that the bias introduced by human demonstrations does not prevent the policies from learning individual behavior.

Without a curriculum or a shaped reward function to guide exploration, *RL sparse* is not able to learn any meaningful actions. The shaped reward function helps the policy to get closer to the goal, it successfully manages to grasp the blue block and

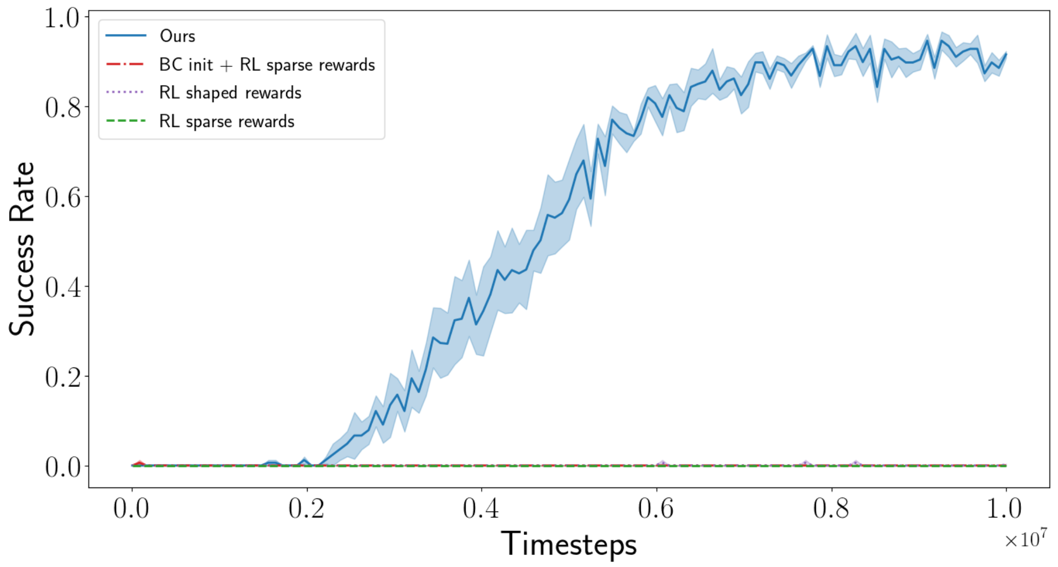


Figure 10: Baseline experiment. We compare our approach with standard RL baselines without curriculum learning on the block stacking task in simulation. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds. Out method is the only one to successfully learn the complete task.

brings it near the red block, but it fails to learn to stack. The policy initialized with BC shows some promising behavior and even succeeds in some cases to lift the block. However, RL is not able to improve the pretrained policy and the lack of any reward signals leads to catastrophic forgetting of the previously acquired skills.

6.3.2 Curriculum Experiment

In this experiment we are evaluating different ways to build a curriculum of start states.

Ours: See baselines experiment.

Linear w/ task difficulty params and regular resets: Linear reverse curriculum learning from demonstrations. The difficulty of start states and the task parameters increases proportional to the training time. The probability of regular resets also increases linearly.

Linear w/ regular resets: Like the former, but without changing the difficulty of

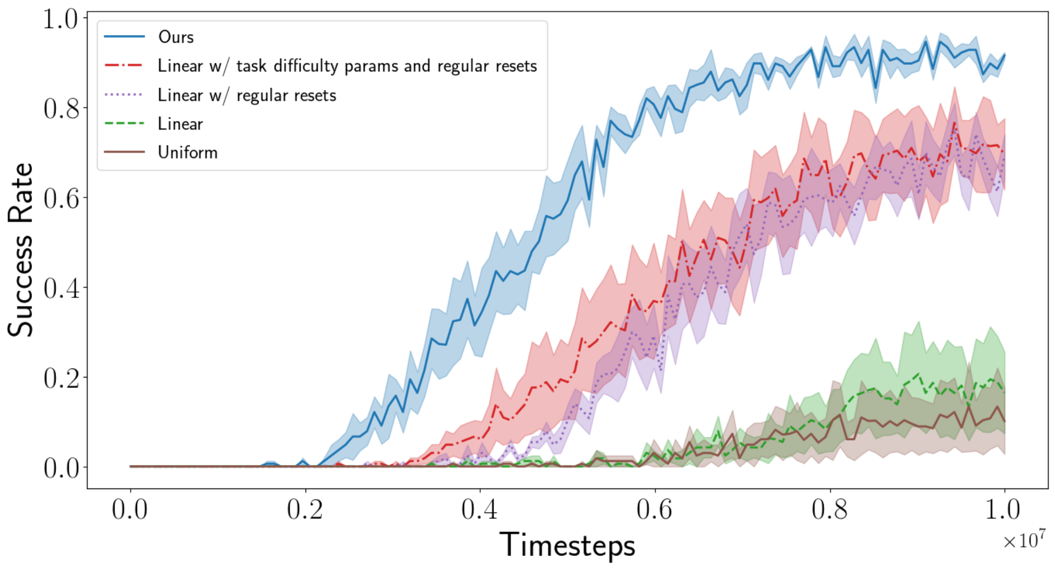


Figure 11: Curriculum experiment. We compare our curriculum learning approach with other curriculum learning methods. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds.

the task parameters, which are kept at the most difficult level for the whole training. Similar to (Fan et al. (2018)).

Linear: Linear reverse curriculum, without changing the difficulty of the task parameters and without regular resets.

Uniform: Uniform curriculum. For the whole training, states are sampled uniformly from the demonstration data, no regular resets.

Results

Fig. 11 shows the results of the experiment. Our methods clearly outperforms the other forms of curriculum learning. It starts to learn faster and achieves a better final performance, with a success rate being more than 20% higher than that of the linear curriculum. It also shows less variance across the seeds, which indicates that the adaptive curriculum improves the stability of the learning.

Linear w/ task difficulty params and regular resets and *Linear w/ regular resets* show similar performance, that is much higher than the linear curriculum without regular resets. This shows how important it is not to learn exclusively from demonstrations, especially if the amount of demonstration data is limited. Uniform curriculum and the simple linear curriculum both perform very poorly, achieving less than 20% success rate on average.

6.3.3 Task Difficulty Experiment

Here, we evaluate if adaptively changing the parameters $\mathcal{H}$ for the task difficulty and domain randomization improves the training speed and performance.

Adaptive task difficulty params (Ours): Full model.

Adaptive task difficulty params, shared δ: Like the former, but a shared difficulty parameter δ for demonstration resets and regular resets (corresponds to using $\delta_r = \delta_d$).

Constant task difficulty params: Adaptive curriculum from demonstrations, but without changing the difficulty of the task parameters $\mathcal{H}$. All parameters were set to the maximum difficulty for the complete training.

Results

Fig. 12 shows the results of the experiment. The full model learns faster and has a higher final success rate. *Constant task difficulty params* shows a particularly high variance, some random seeds performed almost as good as the full model, while others do not even achieve 50% success rate.

6.3.4 Observation Experiment

In this experiment, we investigate the impact of different input configurations on the performance.

Img + state: Default network of our method, see Fig. 8.

Img only: Learning purely from the camera images. We reduced the network architecture to the CNN part of the default network (dotted box of Fig. 8).

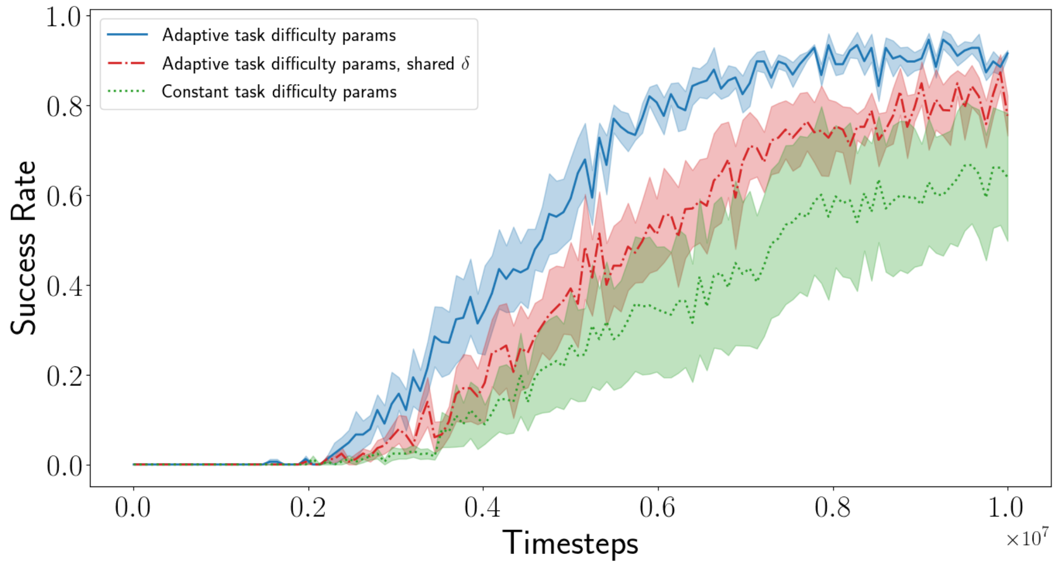


Figure 12: Task difficulty experiment. Here we evaluate the influence of setting the parameters $\mathcal{H}$ adaptively with the δ parameters. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds.

Results

Fig. 13 shows the results of the experiment. Both configurations start to learn fast, but the full model clearly outperforms the *Img only* network. We assume, that this is because the state vector contains valuable information that is not or only insufficiently contained in the images. The height of the blue block above the red block before being dropped is probably the most decisive factor for a successful stack. Due to the perspective of the wrist-mounted camera, the blue block often partially occludes the red block, which makes it difficult to precisely infer the height. The gripper can only rotate approximately 175° to both sides before reaching its limits, if images are the only input there is no possibility to know if a limit is reached soon. Including the remaining episode steps can be important information for the value function. If for instance the block was not successfully stacked on the first trial, the policy will attempt to regrasp the block, but since there is significantly less time available, the value function of *Img only* is likely to overestimate the value of this state.

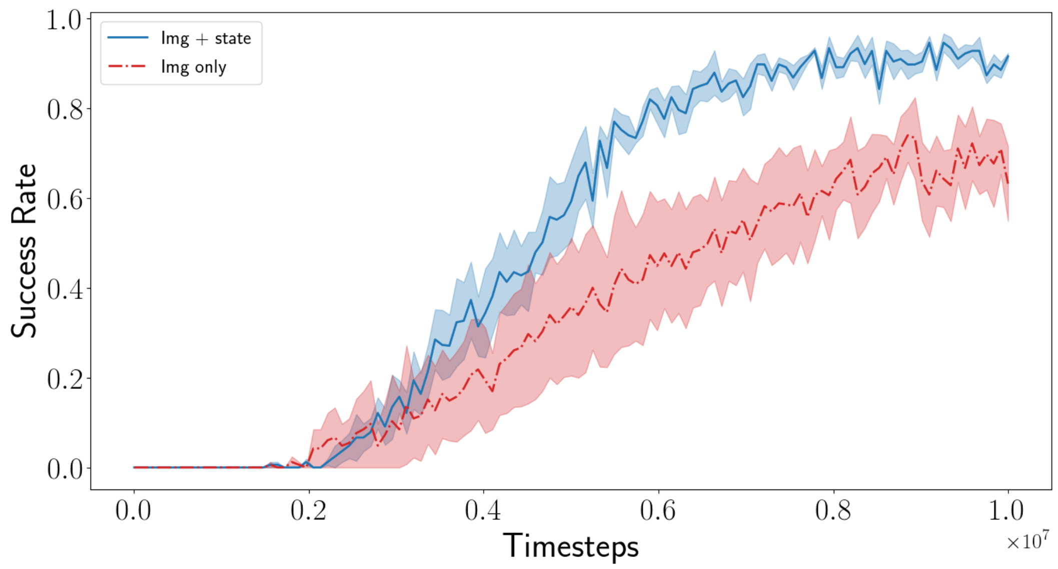


Figure 13: Observation experiment. We evaluate how the performance depends on the input modality. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds.

6.4 Real World Experiment

We demonstrate the capabilities of our approach on a real world block stacking task using the robot setup described in Chapter 4. We train a policy in simulation with Adaptive Curriculum Generation from Demonstrations and domain randomization and run it on the real KUKA iiwa robot without any training in the real world (zero-shot transfer). To evaluate the final performance, we run the policy 30 times on the real robot. We set a constant episode length of 300 timesteps (15 seconds). On each reset, we randomly place the blocks in different configurations underneath the gripper, similar to the task in simulation.

The robot manages to successfully stack blocks within the given time in 67% of the cases and it achieves to at least grasp the blue block in 93% of the attempts. Fig. 15 illustrates a successful task execution. In cases when the policy was not able to stack the block the first time, the policy still attempted a second trial and sometimes succeeded, if there was enough time left. Fig. 14 plots the success rate of the task in dependency of the episode length.

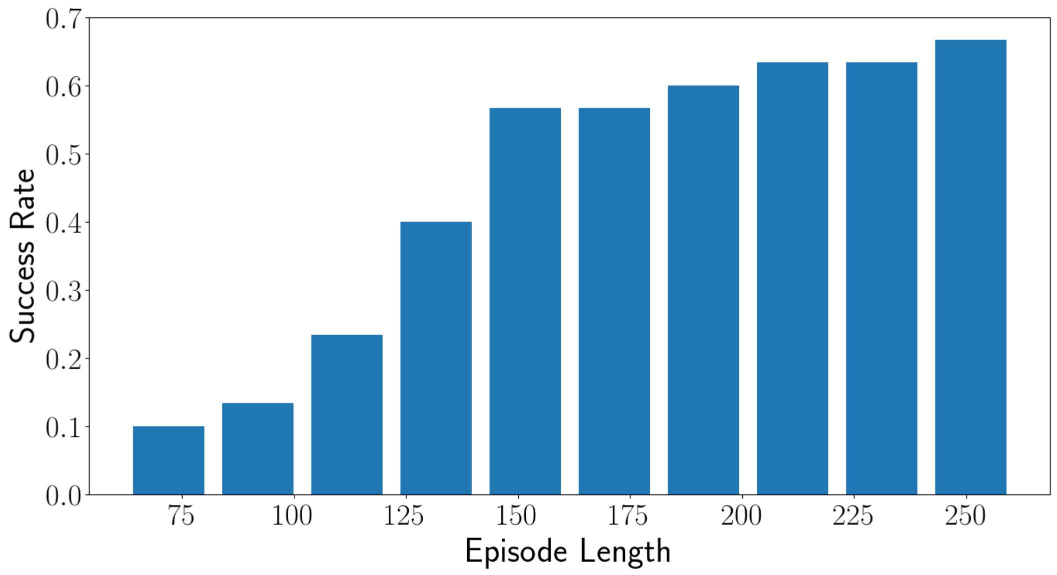


Figure 14: This figure plots the success rate of the real world stacking task in dependency of the episode length. The robot has 300 timesteps to complete the task. Out of 30 trials it managed to stack successfully 20 times.

It is difficult to compare the performance with previous approaches, since there is no clear benchmark task which allows for a fair comparison. Even for block stacking, every approach uses a unique setup with a different robot, a different control interface and the material and size of the objects heavily influence the task difficulty. Zhu et al. (2018), who also do zero-shot sim-to-real transfer of a block stacking task, is the work that comes closest to our approach. However, they use large deformable foam blocks for stacking, these are easier to grasp because they are made of foam and easier to stack because they are larger. They report a success rate of 35% for stacking on the real robot, which is significantly lower than ours.

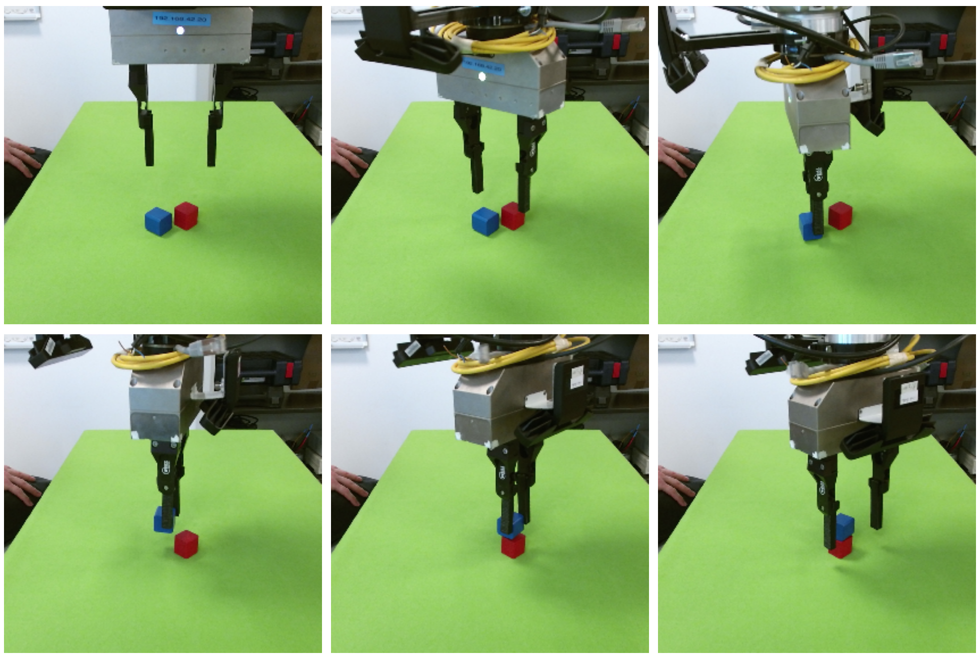


Figure 15: Example of real stacking. The photos show snapshots of the process (top left to bottom right). The results are achieved with zero-shot sim-to-real transfer of a policy on the real KUKA iiwa robot.

7 Conclusion

Reinforcement learning is a powerful, general tool for learning policies that can solve complex robot manipulation tasks. However, when training from sparse rewards, RL suffers from low sample efficiency, because there is no reward signal to guide the agent's exploration. Bootstrapping exploration by the use of demonstrations has been proposed by several approaches in the past and showed promising results.

In this thesis we studied how a curriculum of initial states sampled from demonstration data can be constructed for fast learning speed. We presented Adaptive Curriculum Generation from Demonstrations, an effective curriculum schedule that can be used in combination with an arbitrary on-policy RL algorithm. It automatically generates a reverse curriculum by gradually scheduling increasingly more difficult start states and adaptively setting the difficulty of task parameters, depending on the current performance of the policy. To the best of our knowledge, it is the first time that domain and dynamics randomization for sim-to-real transfer have been learned with curriculum learning.

Our experiments show, that our approach is superior to conventional curriculum learning methods in terms of learning speed and final performance. We demonstrate successful zero-shot sim-to-real transfer of a policy for block stacking. The policy manages to successfully stack blocks with the real KUKA robot in 67% of the cases and has a success rate of 93% for grasping. In addition, we provide a easy-to-use OpenAI Gym Python environment for training policies in simulation and controlling the physical robot.

7.1 Future Work

In future work, in order to further emphasize the relevance, we want to test our approach on an additional set of more difficult tasks, which can not be solved with ordinary curriculum learning. While block stacking has many characteristics that make it a suitable RL benchmark task, a new task could potentially also be closer to realistic industrial applications, for instance one could teach the robot to autonomously screw bolts.

There are also certain aspects of our approach that leave room for improvement. Currently, every task parameter $h \in \mathcal{H}$ is updated homogeneously according to one difficulty parameter δ, but probably there are situations where it would be more beneficial to further increase the difficulty of some subset of parameters, while leaving others fixed. We would like to investigate the possibility to independently set the difficulty of the individual parameters. The *Goal GAN* of Held et al. (2017) seems to be a promising method for automatically scheduling goals at the appropriate level of difficulty for the current policy, which could potentially be used to generate values for the task difficulty parameters as well.

8 Appendix

8.1 Additional Experiments

The experiments were conducted in the same setting as the simulation experiments of Section 6. We investigate the impact of different values for the success rate interval (Fig. 16) and the difficulty increment (Fig. 17).

8.2 Hyperparameters

Table 3 shows a list of hyperparameters that were used in the experiments.

8.3 Reward Functions

The sparse reward function consists of a penalty ϕ for the movement of the bottom block during the execution of the task. It is defined as:

$$\phi = \tanh(\cos^{-1}(2\langle \mathbf{q}_{start}, \mathbf{q}_{end}\rangle^2 - 1) + 10 * \|\mathbf{pos}_{start} - \mathbf{pos}_{end}\|), \tag{14}$$

where **pos** and **q** denote position and orientation (expressed as quaternion) of the bottom block.

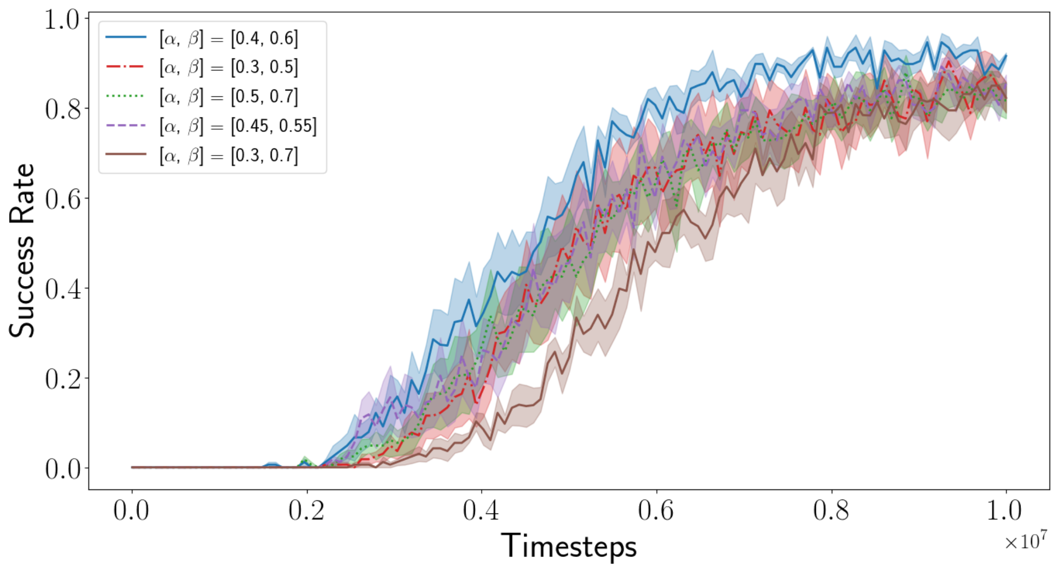


Figure 16: Success Rate Interval Experiment. In this experiment, we evaluate the influence of difference values for $[\alpha, \beta]$. The interval of $[0.4, 0.6]$ corresponds to our default interval and shows the best performance, but also the other runs achieve good scores. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds.

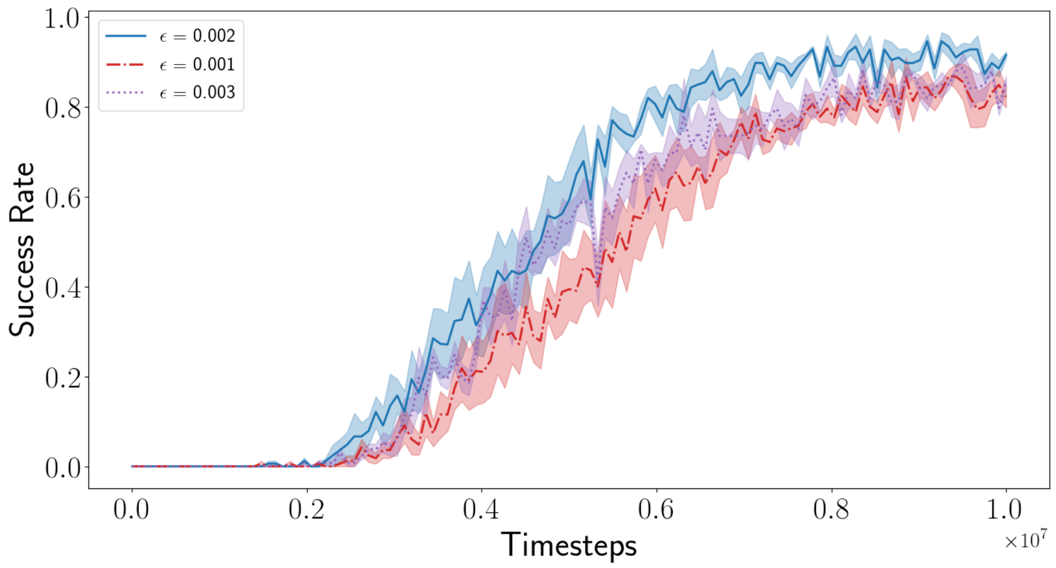


Figure 17: Difficulty Increment Experiment. Here, we are comparing how different values for ϵ influence the training speed and final performance. All three values show similar training curves with our default value of $\epsilon = 0.002$ being slightly better than the rest. The plot shows averages and standard deviations of the evaluation success rate over five runs with different random seeds.

Hyperparameter	**Value**
Adam learning rate	2.5×10^{-4}
Adam ϵ	1×10^{-5}
Discount γ	0.99
GAE τ	0.95
Entropy coefficient	0.01
Value loss coefficient	0.5
Max grad. norm	0.5
Number of actors	8
Minibatch size	8×512
Num. epochs	4
Clip param.	0.1
Training steps	10^7
Interval $[\alpha, \beta]$	$[0.4, 0.6]$
Increment ϵ	0.002
Number of demonstrations	10

Table 3: Default hyperparameters that were used in the experiments unless stated otherwise.

Acknowledgement

First of all, I would like to thank Prof. Burgard for giving me the opportunity to work with a real robot and for giving me the freedom to write on a topic of my choice. I would also like to thank Prof. Brox for being the second examiner of this thesis.
Andy and Max, thank you for being my supervisors, I could not have wished for better support. You dedicated a lot of your time and patience for guiding me throughout the thesis and I always felt welcome to ask for advice.
I would like to extend my thanks to Artemij, Jessica and all the other PhD students at AIS and LMB who helped me with their expertise in reinforcement learning and other topics.
Especially I want to thank Daniel for sharing his knowledge of the KUKA robot with me, saving me hours of work.
Thanks to Claas, Kshitij, Eric and the rest of the students in the AIS pool for making this period as pleasant as possible, complaining with me about the canteen food and giving me back massages after eight hours of work. Otherwise I would have never known that cars in India obey different laws of physics.
Also thanks to my parents and friends for motivation and distraction, this accomplishment would not have been possible without them.
Last but not least, I would like to thank the robot who was my partner for six months. It was not always easy between us but in the end I think I can say, we were a great team and I will miss you.

Bibliography

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

Ivaylo Popov, Nicolas Heess, Timothy P. Lillicrap, Roland Hafner, Gabriel Barth-Maron, Matej Vecerik, Thomas Lampe, Yuval Tassa, Tom Erez, and Martin A. Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *CoRR*, abs/1704.03073, 2017.

Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. *CoRR*, abs/1709.10089, 2017.

Yuke Zhu, Ziyu Wang, Josh Merel, Andrei A. Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, and Nicolas Heess. Reinforcement and imitation learning for diverse visuomotor skills. *CoRR*, abs/1802.09564, 2018.

Linxi Fan, Yuke Zhu, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto, editors, *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 767–782. PMLR, 29–31 Oct 2018. URL `http://proceedings.mlr.press/v87/fan18a.html`.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

J. Peters and S. Schaal. Policy gradient methods for robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Beijing, China, 2006.

Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. In *Robotics: Science and Systems VII, University of Southern California, Los Angeles, CA, USA, June 27-30, 2011*, 2011. doi: 10.15607/RSS.2011.VII.008. URL `http://www.roboticsproceedings.org/rss07/p08.html`.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.

Shixiang Gu*, Ethan Holly*, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Proceedings 2017 IEEE International Conference on Robotics and Automation (ICRA)*, Piscataway, NJ, USA, May 2017. IEEE. *equal contribution.

Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. *CoRR*, abs/1709.07857, 2017.

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.

Martin A. Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch. *CoRR*, abs/1802.10567, 2018.

Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. *CoRR*, abs/1710.06542, 2017.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. *CoRR*, abs/1710.06537, 2017.

Andrei A. Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *CoRR*, abs/1610.04286, 2016. URL `http://arxiv.org/abs/1610.04286`.

Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. *CoRR*, abs/1812.07252, 2018.

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.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

Himanshu Sahni, Toby Buckley, Pieter Abbeel, and Ilya Kuzovkin. Visual hindsight experience replay. *CoRR*, abs/1901.11529, 2019.

Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *CoRR*, abs/1709.10087, 2017.

Carlos Florensa, David Held, Markus Wulfmeier, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *CoRR*, abs/1707.05300, 2017.

David Held, Xinyang Geng, Carlos Florensa, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. *CoRR*, abs/1705.06366, 2017.

Stephen James and Edward Johns. 3d simulation for robot arm control with deep q-learning. *CoRR*, abs/1609.03759, 2016.

Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *CoRR*, abs/1703.06907, 2017.

Stephen James, Andrew J. Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. *CoRR*, abs/1707.02267, 2017.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2016.

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

Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Trans. Graph.*, 37(4):143:1–143:14, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201311. URL `http://doi.acm.org/10.1145/3197517.3201311`.

Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3634-5. doi: 10.1145/2776880.2792704. URL `http://doi.acm.org/10.1145/2776880.2792704`.

Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, Dec 1997. ISSN 1573-2916. doi: 10.1023/A:1008202821328. URL `https://doi.org/10.1023/A:1008202821328`.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

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*, 518:529 EP –, Feb 2015.