Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

CuRL: Curriculum Reinforcement Learning for Goal-Oriented Robot Control

Author:
Harry Uglow

Supervisor:
Dr. Ed Johns

Second Marker:
Dr. Marc Deisenroth

June 17, 2019

Abstract

Deep Reinforcement Learning has risen to prominence over the last few years as a field making strong progress tackling continuous control problems, in particular robotic control which has numerous potential applications in industry. However Deep RL algorithms alone struggle on complex robotic control tasks where obstacles need be avoided in order to complete a task. We present Curriculum Reinforcement Learning (CuRL) as a method to help solve these complex tasks by guided training on a curriculum of simpler tasks. We train in simulation, manipulating a task environment in ways not possible in the real world to create that curriculum, and use domain randomisation in attempt to train pose estimators and end-to-end controllers for sim-to-real transfer. To the best of our knowledge this work represents the first example of reinforcement learning with a curriculum of simpler tasks on robotic control problems.

Acknowledgements

I would like to thank:

- **Dr. Ed Johns** for his advice and support as supervisor. Our discussions helped inform many of the project's key decisions.
- My parents, **Mike and Lyndsey Uglow**, whose love and support has made the last four year's possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of robotics has abundant opportunities to automate or improve upon human ability in all walks of life. The current state-of-the-art in robot learning is fast-moving, with robots able to autonomously complete a widening variety of tasks as new methods are developed. Given the current rate of development and market demand, it is likely that the first widely available home assistant robots will be on sale with a few years.

These robots should be able to perform a vast number of household tasks, such as loading the dishwasher, stacking books onto shelves or laying the table. All such tasks effectively require the robot to move a subject such as a plate to a goal pose while avoiding obstacles in its environment. We call this type of task an obstacle-driven goal-oriented robotic control task. A technique for training robotic equipment to perform such tasks autonomously would have great applications both in home robotics but in wider industry such as for surgical or warehouse robotics. One core task that this project tackles is training a robot to stack a plate into a dishwasher rack.

Deep Reinforcement Learning has risen to prominence over the last few years as a field with large potential for tackling continuous control problems, in particular robotic control in the real world. Deep RL algorithms can learn complex policy functions by using artificial neural networks as approximators. In robotics, a classical approach would be to write a program that explicitly specifies how to approach each section of a task, and respond to situations the robot may find itself in. This approach does not scale well and is useful only in very controlled domains.

Reinforcement Learning (RL) algorithms, on the other hand, do not require the programmer to engineer specific features of the robot's behaviour. Instead the robot receives a "reward" for completing the task, and uses trial-and-error to learn a policy that maximise its reward. In traditional reinforcement learning, all possible states are modelled in a state space and the agent learns what to do in every state. Representing a real world environment like this is infeasible as the state space is far too large.

Deep reinforcement learning overcomes this by using neural networks to approximate a policy function instead of modelling the state space. This makes it well-suited to robotics, and recent research shows how Deep RL approaches have led to strong performance on a range of tasks [5, 6, 7, 8, 9].

One of the core problems with Deep RL is that it requires a significant volume of training data to achieve strong performance; it is very sample inefficient [10, 11].

This is an inconvenience for any application but it can be prohibitively problematic for robotics for the following reasons:

- **Training is time-consuming**
  - The level of computation required makes training lengthy in many areas of machine learning
  - In robotics, environments may need resetting by a human in between episodes. Algorithms which require thousands of training episodes become infeasible.
- **Robots are expensive**
  - Buying many to gather data in parallel is prohibitively expensive
  - In a university environment time using robotic equipment may be charged for
- **Risk of real-world damage**
  - Lots of training could contribute to faster wear
  - An untrained robot could cause damage to itself or other objects during training

Training in simulation offers the potential for many robots to gather training data in parallel. This can both speed up training time and while in a simulation an untrained robot cannot cause any damage. However, training in simulation is not without its own problems. The performance of the robot in simulation is useless if it cannot be transferred to a real environment. Transferring models learned in simulation to reality is far from a solved problem but recent research has shown great progress in this area [12, 5, 6].

In this project we use simulations extensively, not only for faster training and damage mitigation. We also leverage full state data, such as exact obstacle positions, which are available in simulation but not in reality. We use this low-dimensional information to train our policies faster and then transfer them to high-dimensional image-based controllers that can work from images. Another significant reason why simulations are crucial to this project is that in our developed method we transform the environments in ways not possible in reality, manipulating obstacle shape and size to help guide training.

1.1 Objectives

The initial aim for the project can be broken down into three core goals.

1. To improve upon current research by developing a pipeline for training a robot in simulation, transferring trained policies from to reality, and fine-tuning them in the real world.

2. To keep the pipeline general enough to produce policies with good real-world performance on a wide range of tasks achievable with a Sawyer robotic arm [13]

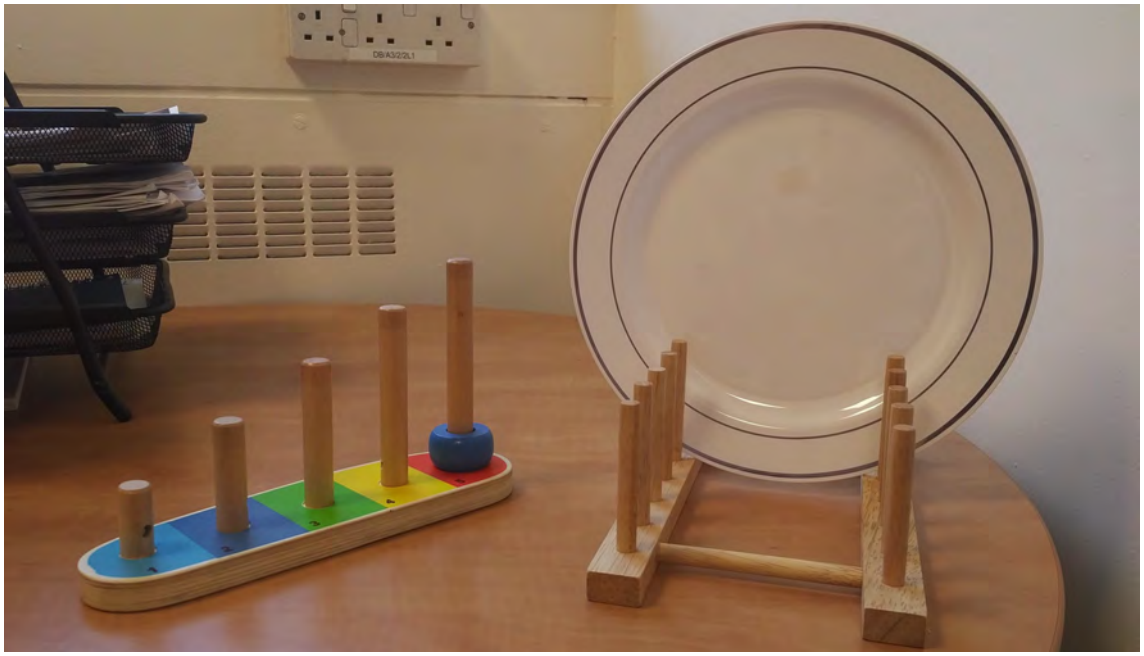

Figure 1.1: Potential tasks we could solve include placing a plate into a dish rack or stacking a bead onto a child bead toy.

3. To minimise the time and effort needed to use the pipeline on a new task.

As the project progressed we found success in a simulated household robotics task: placing a plate in a dish rack. At this point we could have immediately moved to the sim-to-real transfer phase of the project. However, experiments on other tasks showed that the method in its current form was not general enough for use on a wider range of obstacle-driven goal-oriented robotic control tasks. We chose to focus more on our second aim and spent much more time training in simulation to improve our method and develop a novel approach to reinforcement learning for robotic control: CuRL.

Publication Objective

CuRL is a promising new method for robot learning that could be used to be used to push the state-of-the-art for Deep Reinforcement Learning in this area given further research. As such we are targeting the Conference on Robot Learning 2019 for a paper submission. This annual international conference accepts papers focused on the intersection of robotics and machine learning.

The deadline for paper submission is 7th July 2019. To boost our chances of acceptance we will continue working on this project after this report has been submitted, aiming to improve our results where possible and report them in a paper which concisely presents this body of work.

1.2 Contributions

1. **CuRL** - A method for teaching policies to complete goal-oriented robotics tasks that are difficult to solve with state-of-the-art Deep Reinforcement Learn-

ing. Using a curriculum of simpler tasks, we progressively guide a policy towards being able to complete the desired full task. A curriculum of simpler tasks has been used to train an agent in traditional reinforcement learning [14], but to the best of our knowledge this has not been applied to continuous control tasks before.

2. **Set of goal-oriented robotic control task environments** - As CuRL can complete tasks unsolvable by other Deep RL tasks, the current baselines [15] are not complex enough for comparison of curriculum-based approaches. Our Dish Rack and Reach Over Wall tasks could stand as the baselines for comparing success and efficiency of future improvements to CuRL.

3. **Extensions to open-source implementations** - OpenAI's Baselines [15] is a widely-accepted standard implementation of a suite of Deep Reinforcement Learning tasks for testing on a set a baseline tasks. The environment API defined here is commonly used in other implementations. However, it remains quite closely tied to the idea that only one policy network is used to complete a task. In recent papers, Residual Reinforcement Learning [16, 17] proposes to use multiple policies to solve more complex tasks. CuRL was inspired by this concept and also uses multiple policies. We make some small extensions to Baselines to support residual policies, available in a fork.[1]

1.3 Report Structure

We begin with the prerequestites of Deep Reinforcement Learning in Chapter 1.3, working from machine learning in general up to Proximal Policy Optimization, the Deep RL algorithm used in this project. Chapter 2.5.2 gives an overview of the core tasks used in this project which will be referred to throughout. This chapter also contains details on how our environments are implemented.

Chapter 3.3.1 and 4.2 detail our early experiments as we work towards developing CuRL. A full overview of CuRL is given in Chapter 5.3.5, before we detail our attempts at sim-to-real transfer in Chapter 7. An exploration of significant results and why CuRL is necessary is given in our evaluation, Chapter 7.2.3. In Chapter 8.4 we conclude and detail avenues for future research.

[1] https://github.com/harry-uglow/baselines

Chapter 2

Background

2.1 Machine learning (ML)

The branch of computer science and mathematics that develops algorithms which can improve performance on a task with experience. Tom Mitchell [1] defines learning as follows:

> 'A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at task s in T, as measured by P, improves with experience E' (p. 2).

Algorithms in machine learning are often classified into one of three sub-categories:

- **Supervised learning** - A supervised learning algorithm learns with respect to a training set that includes the correct output for each entry in the set. Generally, a supervised learning algorithm would learn to approximate a function that best represents the training data, aiming to give accurate output on unseen examples. Classification is one area where supervised learning algorithms are useful, for example recognising handwritten digits [18].

- **Unsupervised learning** - This area of ML aims to find structure in datasets. They aim to learn something about given datasets based on the data alone, without labels or training examples. Principal component analysis is an example of a mathematical approach to identifying patterns in a dataset.

- **Reinforcement learning** - Reinforcement learning algorithms aim to teach an agent how to behave in an environment using the concept of rewards. RL algorithms allow agents to learn a policy aiming to maximise the reward it receives when interacting with the environment by trial-and-error. As this is a reinforcement learning project, a larger introduction to the area is given in Section 2.3.

2.2 Artificial Neural Networks (ANNs)

The ANN is a data structure central to many ML algorithms. It is used to represent an approximation of a complex function that an algorithm is trying to learn. During the "learning" phase the network is updated to improve that approximation. There

are different types of neural networks but for simplicity and to keep the discussion relevant to this project we will focus only on feed-forward networks.

ANNs are vaguely inspired by the biological neural networks that make up our brains [19]. The brain contains billions of neurons, with many connections to other neurons. When the neuron is stimulated by information received on these connections, the neuron will activate and send an electrical signal to other neurons it is connected to. The artificial neuron is designed with this concept in mind. An artificial neuron has incoming connections and computes a weighted sum of all its inputs. This weighted sum then serves as the input to an activation function, which computes the output for this neuron that is then sent to all other neurons.

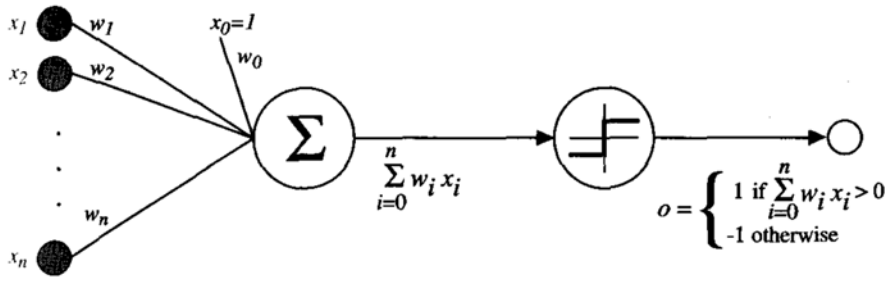

Figure 2.1: Diagram showing a perceptron with a step activation function [1, p. 87]

In an ANN, many artificial neurons are connected in layers. Each neuron is connected to neurons in the layer before it that make up its input. It is then connected to neurons in the layer after it, and its output acts as one of the inputs for these neurons. In a feed-forward network, all the neurons and connections can be represented in an acyclic graph. All connections flow from the networks inputs to the outputs. All the connections have an associated weight and each neuron has a bias. It is these parameters that are updated during training to improve the representation of the underlying "true" function.

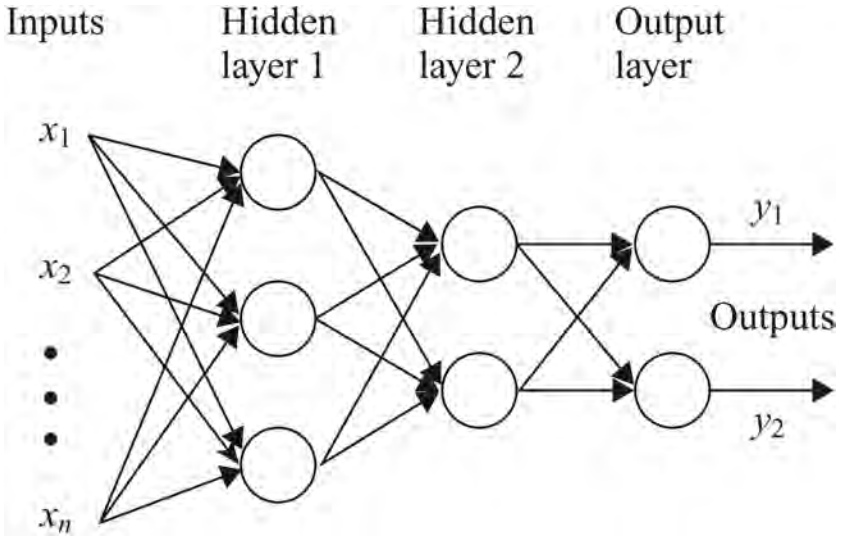

Figure 2.2: A simple ANN architecture with n inputs and 2 outputs [2, p. 205]

A common way to update the nework is with Stochastic Gradient Descent (SGD). Under this method, at each iteration all the training examples are passed through

the network. The aggregate error in the network's actual output relative to the expected output over all examples is calculated using a loss function. Common loss functions include the mean-squared error and cross-entropy. The gradient of the loss function informs us how much we should update each weight. This gradient, along with a learning rate parameter, dictate how much the weights are then updated. This process is known as backpropagation [1].

2.2.1 Strengths of ANNs

Though training may take some time, an advantage of neural networks is that once they are trained they are relatively quick to use. Another is that they allow us to represent approximations of functions that would otherwise be infeasible or impossible for humans to work out by hand. For example, suppose we are trying to write some software that can identify whether an image is of a dog or a cat. One approach might attempt to identify all the features that distinguish images of the two animals by hand. From this we could write a function that looks for all these features and decides what animal is in the image. If such a program achieved good accuracy with this approach it is likely to run quite slowly. It may need to perform many image transformations to make its decision.

Alternatively, we could train a neural network to identify whether an image is a dog or a cat based on a set of training data. Both approaches attempt to find a method that comes close to some "true" function which encapsulates the difference between images of dogs and cats. However, the ANN approach is likely to achieve better performance with less effort, and the leaderboard of a 2013 data science competition would show exactly that [20]. This is why ANNs are currently very commonly used in a wide-range of problems in machine learning.

2.3 Reinforcement Learning

Rooted in behavioural psychology, reinforcement learning is concerned with teaching some actor or agent how to behave. Under reinforcement learning algorithms the agent only needs to understand the concept of rewards for a given task, its behaviour is then learned. Trial-and-error is central to reinforcement learning algorithms. Through repeated interaction with a task an agent can learn the actions which led to strong rewards, and those which led to penalties. It can then use this learning to make better decisions about what actions to take on the next attempt. There are three central components to reinforcement learning algorithms:

- **State** - a representation of the environment at time t
- **Action** - an action is a single behavioural step. For example where the task is to learn to play a video game, actions could be "move left", "move right" and "jump".
- **Reward** - The consequence of performing an action a in state s at time t. In the video game example, this would be a change in score.

Traditional reinforcement learning algorithms keep track of all possible states in an environment. The agent develops a policy to take the optimal action in each

state. However, the optimal action is not necessarily that with the highest reward. It is the one with the greatest expected **return**. Return is defined to be the discounted sum of all rewards from time t. We use a discount factor $\gamma \in [0, 1]$ to represent the extent to which immediate rewards are favoured over future rewards. This discount factor also ensures that the sum of rewards will converge (where $\gamma < 1$) in scenarios where there could be an infinite number of steps. So, the return at time t can be represented by the equation:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.1}$$

We want to train an agent to act in a way that maximises R_t. To do this we track the optimal return for each state, or the **value** of each state. The Bellman Optimality Equation is one way of formulating the optimal value for a given state:

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^*(s')) \tag{2.2}$$

In words, the value $V(s)$ of a state is the sum over all states, of the probability of moving to each state when taking the selected action, multiplied by the optimal return from that new state. The return is represented as the sum of the immediate reward from that transition and the discounted value of the new state. The values are optimal $V^*(s)$ when the best action is taken and the equation is maximised. In a deterministic environment with full knowledge of that environment, the Bellman Optimality Equations can be formulated and solved for all states. The optimal policy is to take the action that maximises the Bellman Optimality Equation in any given state.

It is rarely the case however, that a complete model of the environment is known. RL methods are often divided into two classes when it comes to how they deal with an unknown model. These are:

- **Model-based** - the algorithm attempts to learn all the state transitions $\mathcal{P}$ and all the rewards $\mathcal{R}$ by experience. A policy is developed that chooses actions based on the developed model of the environment.

- **Model-free** - these methods do not attempt to model rewards or transitions, they only estimate the value of state-action pairs based on experience. A model-free policy chooses actions that previously led to good rewards.

Early RL algorithms such as Dynamic Programming fall victim to what is known as the Curse of Dimensionality, which states that the 'computational requirements grow exponentially with the number of state variables' [21, p. 11], meaning as the complexity of an environment scales significantly more space is required to represent it. Then, there are more states to calculate the optimal value for which takes more computing time. Because of the Curse of Dimensionality scalability is a big concern for RL algorithms. Model-free methods tend to scale better to complex environments, where they often use ANNs to approximate policy functions. As such, we will only focus on model-free approaches as the domain of this project, continuous robotic control, will require methods that can scale to real-world environments.

2.3.1 Ensuring sufficient exploration

Without full knowledge of the environment, agents need to explore it in order to learn how to behave to maximise reward. Consider the example of a man driving to work every day, without ever looking at a map (the model). There are a number of routes he can take and he wants to get to work fastest (maximise his reward). On the first day at work he takes one route that gets him to work in 30 minutes. On the second day, he has a choice. He could exploit his knowledge and take the same route again, expecting it to take around 30 minutes again. Alternatively, he could explore a new route hoping to find a faster one, but with the risk of getting lost and taking longer.

This is the **exploration vs exploitation tradeoff**. An agent will likely receive higher reward by choosing actions that have led to high reward in the past, but it cannot learn about potentially better policies if it does not explore. Clearly, both exploration and exploitation is needed to some extent. Typically, RL methods explore more at the start, and shift to more exploitation as more becomes known about the environment.

2.3.2 On-policy and off-policy methods

A further way of splitting model-free RL algorithms is to define them as either on- or off-policy. An on-policy method is one that learns from experience of the environment gathered by following the current policy. Off-policy methods can improve their current policy with experience from other policies. When it comes to the exploration vs exploitation tradeoff, on-policy algorithms must be careful to ensure they explore enough but off-policy ones can exploit with the policy they aim to improve while using other policies to explore. In this project we currently intend to use an algorithm called Proximal Policy Optimisation. This is an on-policy algorithm.

2.4 Deep Reinforcement Learning

When an ANN has a very complex architecture with a vast number of neurons organised across many layers, it is often referred to as a deep neural network (DNN). Deep learning is the area of machine learning that uses DNNs to tackle complex problems in a wide range of areas including computer vision and natural language processing. The networks found in deep learning can be extroadinarily complex. For example the winner of the ILSVRC 2015 classification task [22] used an extraordinarily large neural network with 152 layers [23]. Deep reinforcement learning extends traditional RL, using deep neural networks to represent the state space instead of a hand-developed model. Deep RL has allowed RL algorithms to scale to continuous state spaces. A DNN can represent a policy function mapping states to actions. Infinitely many states, and infinitely many actions can be encapsulated by this approximate policy function.

2.4.1 Success of Deep RL

In 2013, DeepMind popularised deep RL when they published the first deep learning model to directly learn a control policy with reinforcement learning from "high-

dimensional sensory input", in this case visual input. Mnih et al. showcased their Deep Q-Network (DQN) learning to play several Atari games [24]. Their approach performed better on 6 of 7 games than any previous AI approaches, and was better than the best human at three of them. Since then, deep RL has seen significant growth as a research field. DeepMind continues to innovate in the field. They developed a general RL algorithm successful at chess, Go and more [25] and showed a promising method for sim-to-real transfer [5]. Though DQNs generally don't perform as well in continuous space [26] as they do in the discrete environment of Atari games, other deep RL methods has proved particularly successful in robotics. In 2017, OpenAI introduced "one-shot imitation learning", a method allowing a robot to learn a task after watching a human demonstrate it once [27]. Laskey et al. used deep RL with an algorithm known as DART to train a robot to make a bed, again in 2017 [7]. Even more recently in 2018 Rahmatizadeh et al. proposed a technique for learning multiple robotic tasks with visual input, including picking up a towel to clean a box [8].

2.5 Proximal Policy Optimisation (PPO)

PPO is a policy gradient RL algorithm and was proposed in 2017 by Schulman et al. from OpenAI [28]. Policy gradient methods learn directly from agent experience; they are on-policy methods. After gaining some experience the policy is upgraded once before that experience is discarded and new experience is gathered under the new policy. This of course keeps the space complexity of the algorithms relatively low but it comes at the expense of sample efficiency. It typically takes a vast amount of experience to achieve good performance.

Deep RL algorithms are generally very sensitive to hyperparameters [29]. For example if the learning rate is too large, a policy update could move too far in the update direction and give a very poor policy. In an on-policy environment, the agent may then never recover as it struggles to gain useful training data under this policy. One of the core aims when developing PPO was to make the algorithm easy to tune by reducing this sensitivity to hyperparameters. Another two important objectives were that the algorithm was both easy to implement, and sample efficient.

2.5.1 Understanding the PPO objective function

Intuitively, a policy gradient method attempts to estimate the gradient of the policy and use gradient ascent to find the optimal policy. Therefore, they need an objective function which has a gradent estimator as its differential. The most commonly used objective function is given in Equation 2.3. It is the expectation of the log of the policy's actions multiplied by an estimate of the advantage function. The advantage function encapsulates the relative value of a chosen action given the current state. It is the difference between the actual return, or the discounted sum of rewards, and the expected return under our current estimate of the value function. The advantage function essentially tells us how much better or worse return form an action was than our expected value for that state.

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.3}$$

Using this as our whole objective function seems sensible, but as mentioned above this can lead to detremental policy updates that step too far in one direction. A previously developed algorithm called Trust Region Policy Optimisation (TRPO) [30] adds a constraint to ensure policy updates are not too large. This complicates the policy optimisation problem and makes TRPO both more complex to implement and slower, but the principal of developing a "trust region" and not straying too far from the current policy is present in PPO. Like TRPO, PPO uses what is known as a "surrogate" objective function, containing a probability ratio between the old and new policies instead of the log of the policy function.

$$\hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t] \tag{2.4}$$

To overcome the stated problems with TRPO's constraint, PPO's objective function uses a simpler constraint included directly in the objective function. Setting $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, the main objective is:

$$L^{CLIP} = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.5}$$

As the equation shows, the objective it uses a min function to either give the value of the unconstrained objective function as in 2.4 or the advantage function multiplied by a clipped probability ratio. Figure 2.3 shows the effect of the clip function.

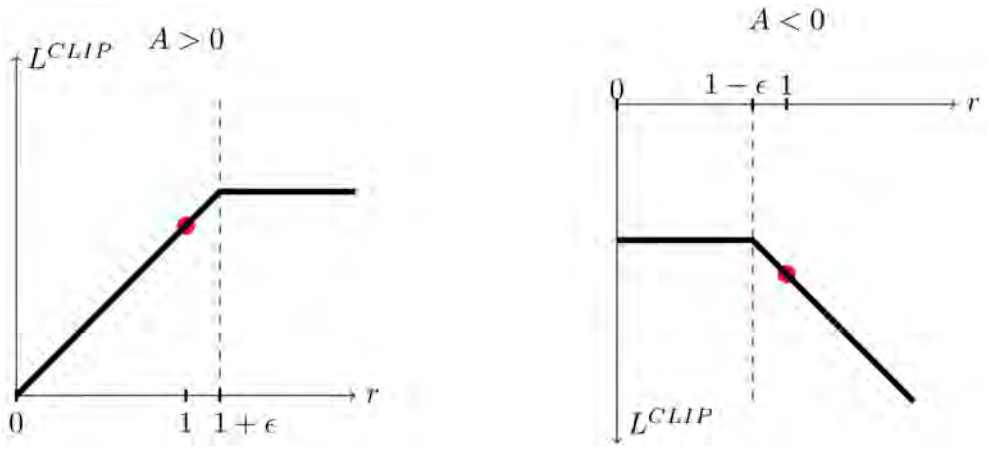

Figure 2.3: Left: Change in L^{CLIP} with r when the advantage function A is positive. At values for r greater than $1+\epsilon$ L^{CLIP} does not change Right: Change in L^{CLIP} with r where the advantage function A is negative.

When the advantage function is positive and the selected action led to a better return than expected, above $r = 1$ the action will become more likely after the policy update. The clip at $1+\epsilon$ ensures that the update is within a size set by the hyperparameter ϵ. We stay within our trust region. When the action was worse than expected, meaning the advantage function was negative, values of $r < 1-\epsilon$ are clipped. This avoids completely destroying the likelihood of selecting an action based on one bad experience.
The advantage function used throughout is an estimate as it contains an estimate of the value function. To measure the accuracy of this estimate a squared error loss

function $L_t^{VF}(\theta)$ is used. Combining this loss term with the clipped objective term and an entropy term to encourage exploration, we obtain the overall PPO objective function.

$$L_t^{PPO}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2.6}$$

2.5.2 Advantages of PPO

This project is concerned with training a robotic arm to perform a variety of tasks. These tasks are continuous control problems, which PPO has been shown to be effective on. Figure 2.4 shows it performing better than several other algorithms previously known to perform well on continuous control problems. The graphs also show that PPO generally performs well in fewer timesteps than the other algorithms. This and the fact that it is relatively simple to implement make it an attractive choice for this project.

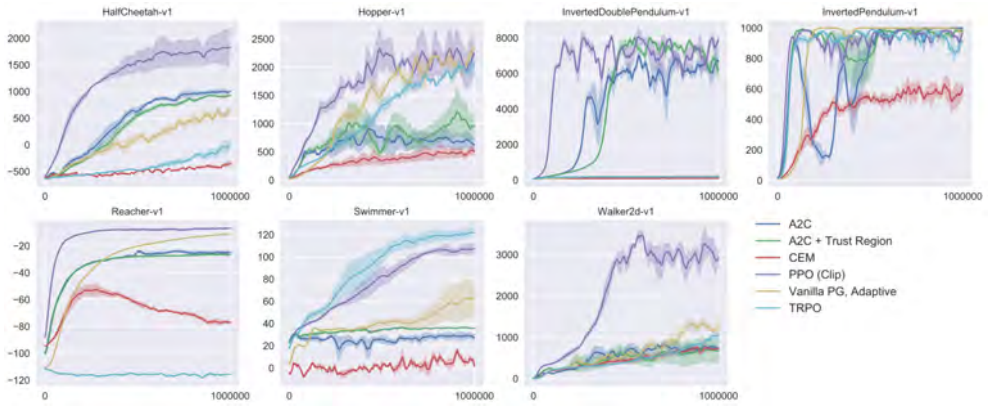

Figure 2.4: PPO compared with other algorithms on several MuJoCo physics engine [3] environments, training for one million timesteps.

Main methods in Figure 2.4

- **Vanilla Policy Gradient** - Policy Gradient uses the sampled reward return to upgrade the policy without a value estimate, and therefore no advantage function. Put simply, this can lead to incorrect updates because the gradient is not based on any comparison to other actions. When all actions lead to negative return, even when the best action was taken and penalty was minimised, policy gradient will see the negative return and think the action should be avoided.

- **A2C** - The Advantage-Actor-Critic uses an ANN to learn both the value function and a policy. The value function is the critic, and the policy function is used by multiple actors in parallel to gather experience. A2C uses Equation 2.3 for its udpates [31]. It does not use a trust region and so can make destructively large updates. However the graph shows adding a trust region does not help too much.

- **CEM** - Avoiding heavy technical detail, the Cross-Entropy Method is a statistical method that represents the policy function as a distribution of the probabilities for selecting actions. After some interaction with the environment, it updates the distribution to better fit the sampled experience [32].

Chapter 3

Environment Specification and Implementation

3.1 3D Simulation Software: V-REP

V-REP is a simulation software which allows users to create and run simulations through its GUI. It comes with models of many real robots preinstalled and ready to be dragged and dropped into a scene. In particular it had an official model of the Sawyer arm provided by Rethink Robotics. It is also possible to create new objects in V-REP from a variety of primitive shapes. This is how we created the dish rack and other models for the project's simulations. Multiple cameras and vision sensors can be used to render and retrieve images of a simulation. Environments can be simulated using a few different physics engines, all available from a simple dropdown. We didn't require very complex physics in the types of tasks we hoped to simulate, so our choice of physics engine was not too important. We chose to use Bullet [33], an engine widely used on other similar projects [34, 35].In general the V-REP GUI provides easy customisation of almost any aspect of a simulation which made it a useful tool for creating simulations for this project.

Beyond the GUI it is of course possible to connect to and control simulations from other programs. V-REP's main scripting language is Lua and an extensive Lua API is provided for programmatically managing parts of a simulation from V-REP. To control a simulation from external programs there is a more limited remote API with bindings available in Python, C/C++, Matlab, Octave and Lua. This has most of the core API we needed but where it was lacking we created Lua functions in a simulation script and called them from Python.

Common Terms

There are some terms used in the report referring to how objects behave in V-REP, which we will quickly define:

- **Vertex** - Vertices are the faces of shapes. In 3D simulations, to keep calculations simple and therefore fast, these are always planar triangles. This includes curved shapes such as spheres, which do not have true mathematical curves in a 3D simulation. Instead they are approximated using triangular vertices.
- **Static/Dynamic objects** - If an object is set to static in V-REP it means its

position is fixed relative to its parent object. A dynamic object is the opposite of this. Its postion is not fixed and it behaves according to the environment's physics engine.

- **Respondable** - Respondable objects can interact with other objects in the scene i.e. other objects can collide with them. If an object is non-respondable, other objects will pass through it.

3.2 Task Specifications

Here we provide details of all significant reinforcement learning tasks created over the course of the project. In their implementation they conform to OpenAI Gym's environment API, implementing the 'Env' class. All environments specified the following:

- **Observation Space** - Observations are the same as "state" defined in 2.3. Observations need not be the full state of the underlying environment, instead they are the part of the state known to the agent. The observation space is the range of valid observations for an environment. For example, in an image-based environments the space's shape is the resolution of the image and the number of channels in an image, and each of these is an integer in the range of valid pixel values eg. $[0 - 255]$. In a full state environment, the observation space is a single vector of values such as joint angles and target position.

- **Action Space** - The action space defines the range of valid actions. This could be a discrete range, such as $open, close$ for a gripper, or a continuous range such as the target velocities for seven joints. In all tasks described below, actions are target velocities for each joint on a robot.

- **Reset method** - This called at the beginning of the episode. The method should return the environment to an initial state and return an observation of that state.

- **Step method** - This method takes an action as input. It should execute that action on the environment, advancing the simulation to a new state and receiving a reward. The reward and an observation of this new state should be returned in the tuple `(observation, reward, done, infos)`. The tuple contains two other values. `done` is a boolean. When `True`, the algorithm should call `reset()`. `infos` is a dictionary that can be used to return additional data where needed.

Many of the tasks in this project shared common reward components. We will define them here to refer to in the task descriptions. The distance component,

$$r_d = -||x_g - x_s||_2 \tag{3.1}$$

where x_g is the target position and x_s is the **subject** position. The subject is either the position of the robot arm's end effector, or where the robot holds an object it

is that object's centroid. This component penalises an agent more the further it is from a target location. The orientation component

$$r_\theta = -\frac{\sum_{i \in \mathbf{axes}} |\theta_{t_i} - \theta_{s_i}|}{max(||x_g - x_s||_2, c)} \tag{3.2}$$

penalises difference in orientation between the subject and a target. θ_t is the target's orientation and θ_o is the subject's orientation. **axes** is the set of orientation axes we are concerned with, and c is a constant. The sum of the orientation difference is divided by the result of max so that incorrect orientations are penalised more the closer the subject gets to the target. By the time the subject is within c of the target, it should be in the correct orientation and so the orientation reward component is at its maximal weighting. The control component,

$$r_v = -\sum_{j \in J} a_j^2 \tag{3.3}$$

where a_j is the velocity for joint j in the set of controllable joints J, penalises large actions. This aims to control the speed at which an agent attempts to move the robotic arm. Later on in the project, this component is removed and actions are scaled to a sensible range instead.

3.2.1 Reacher Tasks

2D Reacher

A three-jointed arm has to reach a point in two-dimensional space. Observations are the 3 joint angles and Cartesian coordinates of the target point. The reward function is

$$r = \lambda_1 r_d + \lambda_2 r_v \tag{3.4}$$

where λ values are hyperparameters used to weight each reward component. As this task was intended to aid familiarisation with the field of robot learning and to sanity check early implementations, it was kept fairly simple. Joints were reset to the same positions every episode and in first attempts so was the target.

Sawyer Reacher

In this task the agent must control a seven-jointed Sawyer robot arm to reach a target cube. Observations are the joint angles and the position of the cube in 3 dimensions. The reward function is the 3D, seven joint equivalent of 2D Reacher's reward function, containing a distance and control component. The initial joint angles are kept the same every episode but the target cube's position is randomised.

Reach Over Wall

We created this environment as a simple extension to Sawyer Reacher, introducing an obstacle for the robot to avoid. The environment allowed us to explore different strategies for learning to solve more complex tasks, such as using waypoints to guide learning. It was not intended to be used for sim-to-real transfer. Observations and

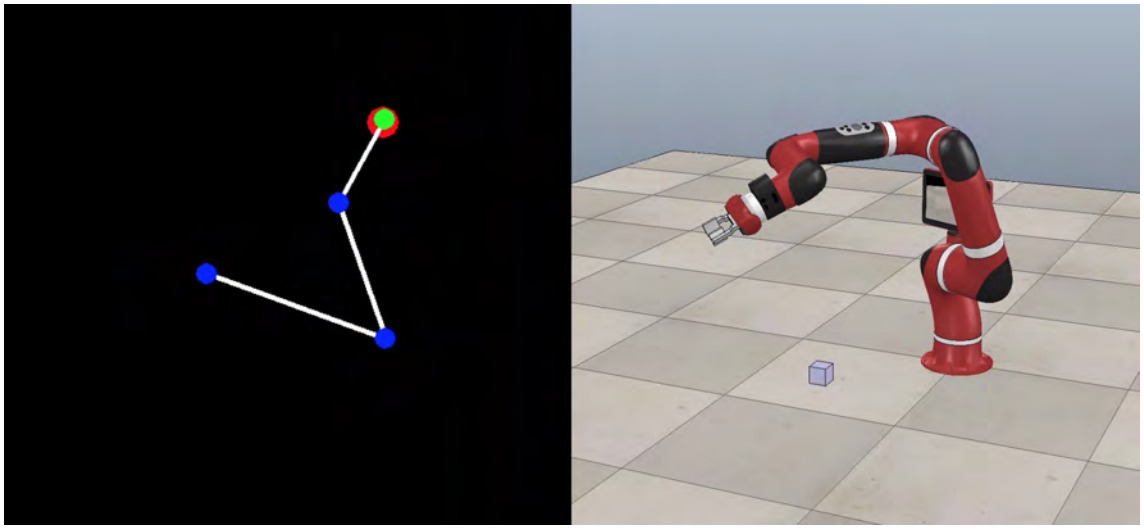

Figure 3.1: Basic reacher tasks used in this project, in two and three dimensions.

actions are the same as above. The wall object stood 300 mm high and was 19 mm thick. It was dynamic and respondable, meaning the robot could knock it over. Its position was not varied, but target cube position and initial joint angles were.

In different attempts at solving the task we used two different dense reward functions. The simplest is

$$r_1 = \lambda_1 r_d + \lambda_2 r_v + \lambda_3 r_w \tag{3.5}$$

where $r_w = -||\theta_w||_1$ is the orientation vector of the wall relative to its upright starting orientation. This component penalises an agent for moving or toppling the wall. A slightly more complex function using waypoints is

$$r_2 = \lambda_1(||x_g - x_w||_2 + ||x_w - x_s||_2) + \lambda_2 r_v + \lambda_3(-||\theta_w||_1) \text{ OR } r_2 = r_1 \tag{3.6}$$

where x_w is the position of a waypoint above the wall. This reward function uses the total distance from the target to the waypoint and from the waypoint to the Sawyer's tip as its distance component for the first phase of the task. After the waypoint is passed r_d is used as normal. Different conditions for switching from the first distance component to the second were used and are discussed later. The unusual reward function aims to incorporate a waypoint into a dense reward function, encouraging the robot to travel to the target via the waypoint. However, we concede this can lead to mathematical incontinuities. If we move from distance via waypoint to straight distance to target when the subject is not exactly at the waypoint there will be a jump in the reward.

3.2.2 Dish Rack Tasks

Putting a plate in a dish rack is the first task we intended to transfer to the real robot. As such, we tried to create a simulation as close to the real world environment as possible. We measured the real wooden dish rack and recreated it in simulation. It's base was made of two $(260 \times 24.5 \times 14)$ mm cuboids held parallel to each other by two cylinders with length 94.5 mm and diameter of 8.5 mm. To create the prongs,

each cuboid had 7 equally spaced cylinders with diameter 8.5 mm and length 71 mm.

We tested two different types of observation on this task, both of which contain the joint angles as normal and the rotation of the rack about the x axis. The first observation gives the absolute x, y coordinates of the rack. The second is a relative observation. This gives a 3-dimensional vector from the plate's centre to the target. When using an image based controller, observations are instead 128×128 RGB images. When resetting the environment, the joint angles are positioned at random using a normal distribution centred at a default position. The rack is positioned uniformly at random within a 150×200 mm rectangle, and rotated about the x axis uniformly at random up to a maximum of 0.25 radians in either direction.

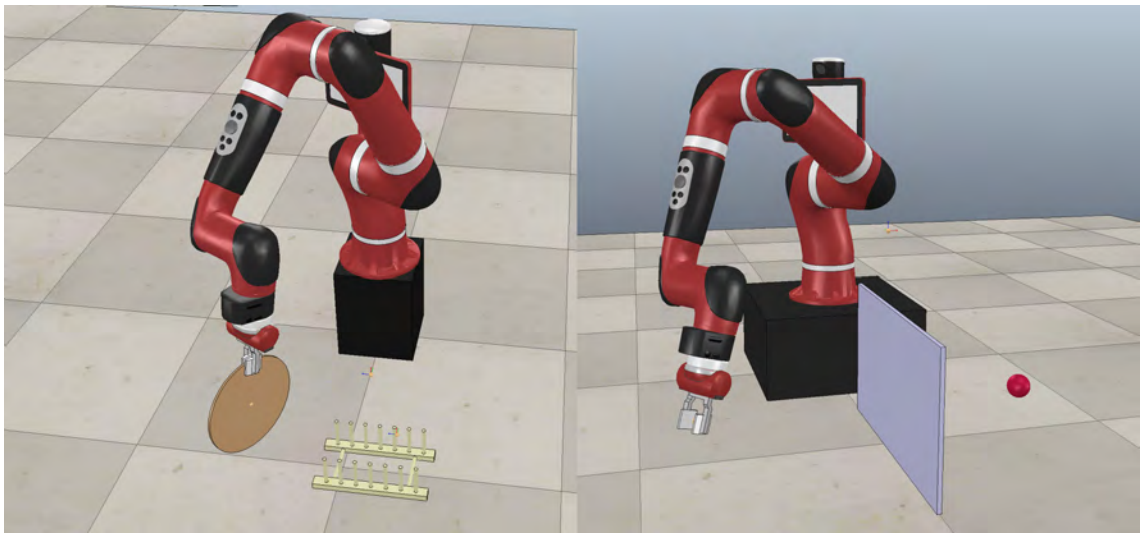

Figure 3.2: Left: Dish Rack scene created in V-REP. We assume the rack cannot be moved so it is made static. Right: Reach Over Wall. The red sphere is static and non-respondable and is used to mark the current target location.

Dense and sparse rewards

We initially train with dense rewards, later using residual learning to train a residual policy in a sparse reward setting. The dense reward function is as follows:

$$r = \lambda_1 r_d + \lambda_2 r_v + \lambda_3 r_\theta \tag{3.7}$$

with reward components as defined above and λ coefficients as hyperparameters. Early versions of this reward function also include a collision penalty, a fixed value deducted from the reward in any step where the plate and the rack are touching. We later remove this penalty and instead clip the actions to a safe maximum speed.

The sparse reward function gives a fixed positive reward in any step where the plate is correctly oriented in the target slot of the rack. We define this as when the plate's centre is within 1.5 cm of a point in the middle of the desired slot, and within 0.1 radians relative to the target point's alpha and beta orientation axes. We are not concerned with the gamma axis as this is the rotation of the plate's circular face, which does not matter. No other rewards or penalties are given.

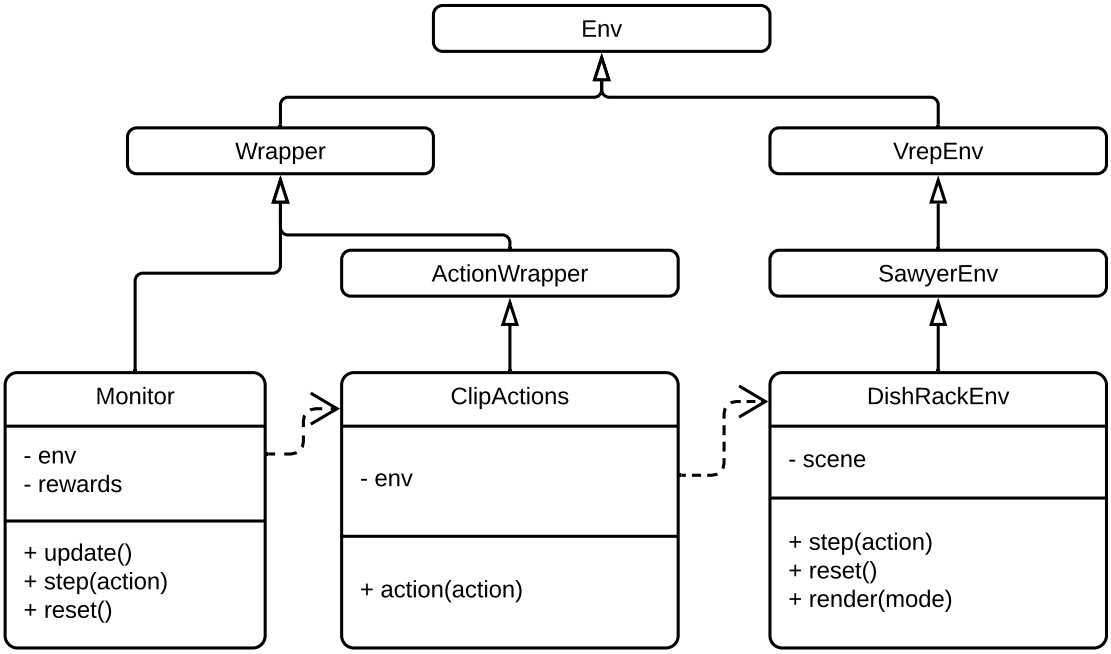

Figure 3.3: Architecture Diagram showing wrappers around a base reinforcement learning environment. Only significant fields and methods shown.

3.3 Environment structure

OpenAI Gym's environment API includes not only the base `Env` class but also a `Wrapper` class. Subclasses of wrappers are used to composably transform a base environment in a particular way, such as converting observations to Tensors. We make extensive use of these wrappers. OpenAI Baselines introduces the `VecEnv` class, for vectorised environments. Subclasses of `VecEnv` allow multiple instances of an environment to be run asynchronously in parallel. It also introduces a `VecEnvWrapper`, for transforming vectorised environments in a modular fashion.

We make extensive use of `Wrappers` and `VecEnvWrappers` in our implementation. Figure 3.3 shows the structure the Dish Rack environment and its wrappers before vectorisation as an example. `DishRackEnv` inherits `SawyerEnv`, encapsulating what is common to all environments with a Sawyer robot. This in turn inherits `VrepEnv`, which handles communication with V-REP. `SubprocVecEnv` manages parallel environments by creating each environment in its own subprocess. Figure 3.4 shows an example of how this vectorised environment might be wrapped for use in residual reinforcement learning.

3.3.1 Explanation of Significant Wrappers

Pre-existing wrappers

These wrappers are already implemented in either Gym, Baselines or the Ikostrikov RL suite.

- `Monitor` - Saves rewards and episode returns to log files for graphing.

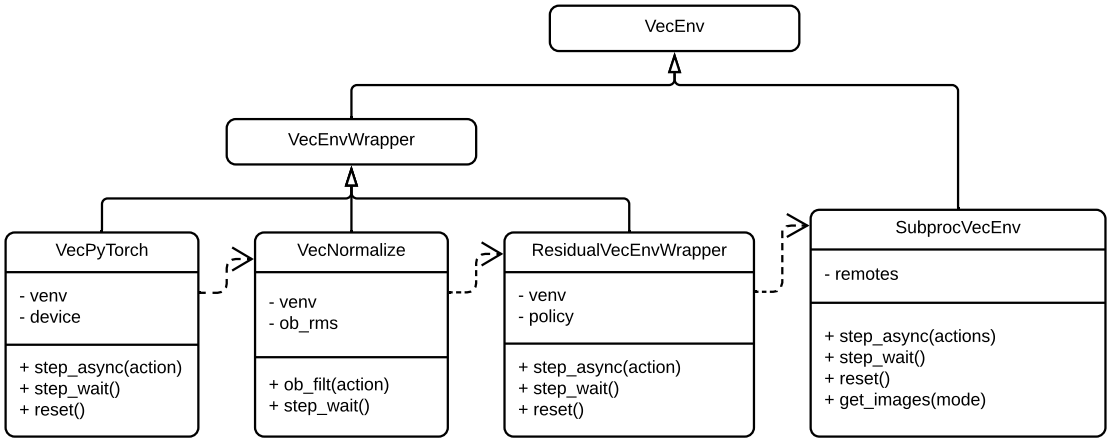

Figure 3.4: Architecture Diagram showing vectorised environment and surrounding wrappers. Only significant fields and methods shown.

- `SubprocVecEnv` - Manages parallel environments, giving each their own subprocess and communicating with them as the parent. `step_async` passes each action in an array of actions onto its respective child environment, before calling `step_wait` to wait for returning observations and rewards.

- `VecNormalize` - Keeps a running mean and standard deviation of observations in `ob_rms`, using this to normalise each new observation as it arrives with the `ob_filt` method.

- `VecPyTorch` - Converts array of observations to Tensors. This makes them usable as input to a Pytorch policy network.

Contributed wrappers

OpenAI's environment structure heavily assumes that only one policy network operates on the given task. To use residual reinforcement learning in this project we had to make considerable changes. We made our changes in the form of wrappers to try and make our work reusable by keeping to convention as much as possible. Here we will give a quick overview of new wrappers.

- `ClipActions` - In spite of environments defining an action space, Baselines does not currently respect those action ranges. We add a simple clip wrapper to keep actions within the specified range. `ClipActions` extends the pre-existing `ActionWrapper`. This means it implements the `action` method, called at every step to transform the action before it is sent to the base environment.

- `ResidualVecEnvWrapper` - Keeps track of most recent observation (normalising it where appropriate). In `step_async`, uses that observation as input to the initial policy. Output from this policy is added to the given residual action to obtain the whole action. This is then passed onto its wrapped environment to be executed.

- `ImageObsVecEnvWrapper` - Wraps a full-state environment so it can instead be used as one with visual observations. Uses `get_images` method to call `render` on an underlying object. Passes the returned images on as observations instead of the low-dimensional state vector.
- `PoseEstimatorVecEnvWrapper` - Used to rollout a policy with a pose estimator. Keeps track of most recent image observation. Inputs this to a pose estimator network and replaces the observations of lower `ResidualVecEnvWrapper`s with estimates.
- `E2EVecEnvWrapper` - Similar to `ImageObsVecEnvWrapper`. Returns a tuple of partial state and images as observations for training an end-to-end controller.

Chapter 4

Early Work

We began towards the end of the first term, working in small steps towards training a robot in 3D simulation. The starting point, provided by Dr. Johns, was a 2D simulation of an arm with three joints moving to a point. The program works out how to move the arm to the point using inverse kinematics. These are calculations based on perfect information of the environment, it is not a method usable for complex tasks based on image data of the real world. My first step was to train a neural network to output joint velocities and use these to move the arm to the target instead. Starting with a small example where the network architecture need not be complex was a useful exercise for familiarisation with PyTorch.

We trained an ANN with fixed starting joint angles and a fixed target point. The network received as input the three joint angles and the x, y coordinates of the target, all normalised into the interval $[0, 1]$. As output it returns a velocity for each of the joints, indicating how they should rotate at that timestep. Though not reinforcement learning as there is no concept of rewards, the 2D example served as an opportunity to explore methods of gathering training data which may be useful later.

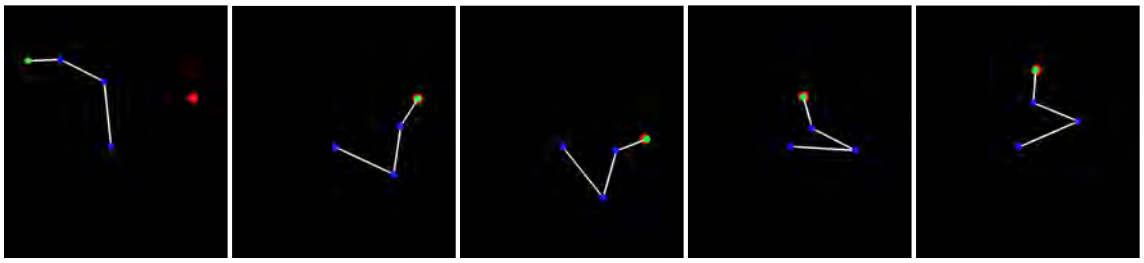

Figure 4.1: Leftmost: Starting configuration. Moving right: The arm in four different successful poses. The red circle marks the target.

We used an approach known as DAgger, which collects training data using the current policy at each iteration but updates the policy using all training data collected under any previous policy [36]. This meant that at each step when testing the neural network we used the inverse kinematics to determine what the network should have output. This information was then added to the training set and after an episode the network is trained some more. Before the first run, the network is initialised with a demonstration. The demonstration gives a set of points along a straight path to the target with the expected joint velocities. This initialisation

ensures training starts along the right lines to keep the data DAgger generates relevant. Eventually the training set is large enough and the network has been trained enough that the robotic arm will consistently make it to the target.

Following success in that, we made the problem more realistic. The next aim was to train the network to move the arm to any randomly selected target in a small range. We kept the initial joint angles fixed as these should be controllable even in a real world environment, and the target being in an expected range also made sense. Even in the real world it's unlikely that the task would have the target just anywhere in the environment. We gathered training data in two different ways to compare their performance. The first is to use DAgger as before and the second is to continue adding demonstration paths like the one used to initialise the DAgger approach. We called this the "ground truth" method. The data for this method could be generated beforehand into a huge supervised learning training dataset. However, we add a new path each episode so that the training set sizes remain the same and the tests are fair. Even though this is supervised learning DAgger is similar to an on-policy algorithm in that can learn from its own policy's mistakes as it goes. On the other hand with the ground truth method, the network just learns what we would ideally like it to do where no mistakes are made.

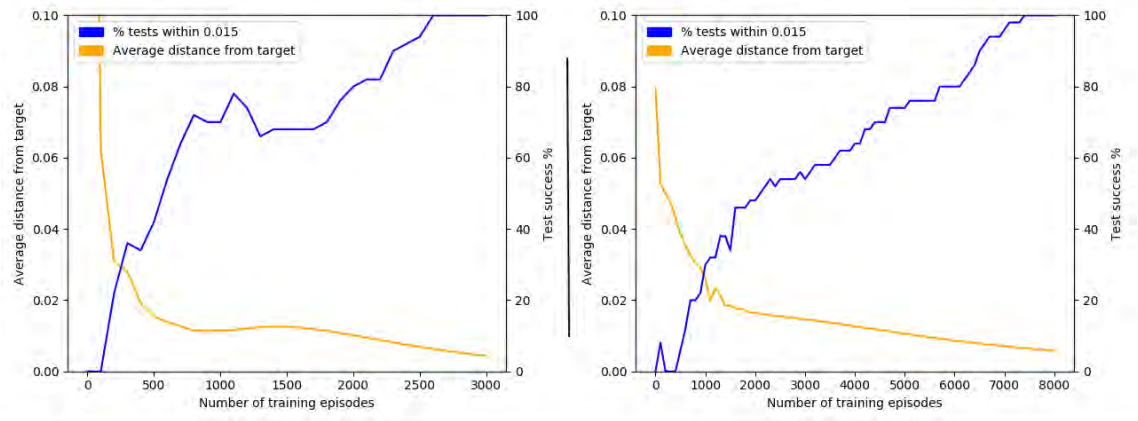

Figure 4.2: The network is trained for 10 epochs in between every episode. Performance on the 50 point test set is measured every 100 episodes. Left: Results for DAgger. Right: Results for ground truths.

The results of the tests are shown in Figure 4.2. A test was deemed to be succesful where the end of the arm was within 0.015 distance from the target. Visually this meant the end of the arm is entirely over the target. There were 50 test targets, generated with a seeded random generator so the targets were the same across all tests. DAgger takes 2,600 training episodes to make all 50 targets, a significant speedup on the 7,300 episodes taken by the ground truth method. While DAgger requires fewer samples, even on this simple example in only 2 dimensions we can see it requires a significant volume of training data before the policies are 100% accurate.

Moving forward I will next be using PPO to learn this simple task. In the following sections this simple example of training an arm to move to a target will be referred to as the "arm to point" task.

4.1 2D Reacher with Proximal Policy Optimisation

As we seek to develop a pipeline capable of learning many tasks, it did not make sense to test and compare performance on this early task when selecting our reinforcement learning algorithm. Ideally the RL algorithm we selected would work out-of-the-box without fine-tuning hyperparameters. Proximal Policy Optimisation (PPO) is a recent algorithm that has shown itself to be more robust to hyperparameters and more sample efficient (shown in Figure 2.4) than other policy gradient methods. For these reasons chose to use PPO as the project's core RL algorithm.

I decided to use Ilya Kostrikov's implementation [37] as a starting point. It is a Pytorch implementation of Proximal Policy Optimisation for use in deep reinforcement learning. It is inspired by OpenAI's popular "Baselines" [15] implementation, but I chose to use this one instead for the ease of use of PyTorch compared with Tensorflow, and because the smaller codebase would make for easier modifications, where necessary. Like Baselines, Kostrikov's implementation assumes environments conform to the interface in OpenAI's Gym toolkit [38].

To train on the 2D Reacher task I rewrote it as a Gym environment, as described in Section 3.2.1. We trained a policy to reach a target positioned randomly in a square with sides of length 0.3. We let the policy train for 100,000 timesteps and it converged easily within that. Over 50 test episodes, the trained policy positions the arm within the square in every one.

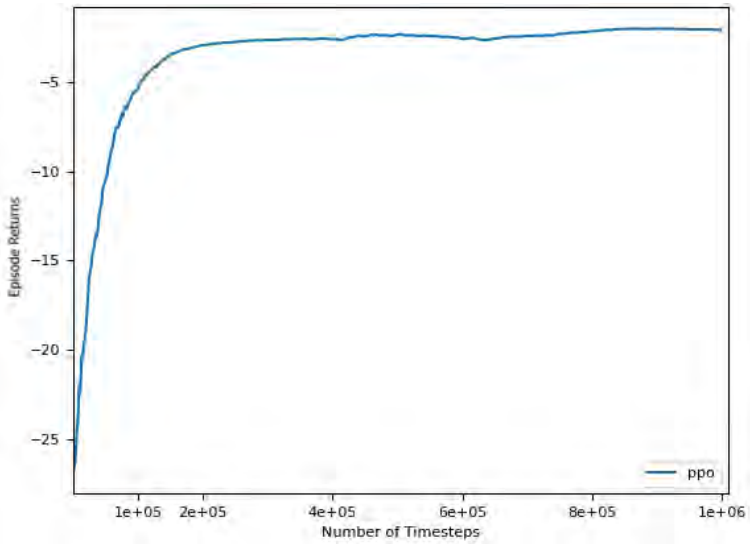

Figure 4.3: Returns over time as training continues on the 2D Reacher task.

The training graph is shown in Figure 4.3. It is difficult to draw a direct comparison between PPO and DAgger or ground truth as the reward function is not just the distance from the arm to the target. The reward function also does not show us how often the policy completes the task (under completion conditions in Figure 4.2). However, after the 100,000 timesteps the policy is 100% successful, and it appears to converge within 20,000 timesteps. This translates to within 625 episodes as the episodes were 32 steps long.

Though this is a rough and inaccurate comparison, it appears that PPO is much more sample efficient than supervised learning methods. A scientific comparison of PPO against these methods is not really relevant to the project's aims, so we do not

waste time by providing one here. Nonetheless it is useful to make this comparison to solidify our choice in using RL with PPO.

4.2 Moving to 3D

Now that the Pytorch implementation of PPO had been proven to work on the simple 2D reacher task, we wanted to start using it in 3D simulations of a real world robot and needed a tool to do so. We chose to use V-Rep[39] as James et al. [4] used it in their research on Sim-to-Real transfer. This meant Dr. Johns had experience with it and could even provide sample code for connecting to V-REP scenes from Python.

To make sure all the code written for connecting to V-REP and controlling an environment was correct we kept the first reinforcement learning task simple by simply recreating our Reacher task, as described in Section 3.2.1. We created a scene with the Sawyer arm at the center and a target marked by a cube. Our scene also had an attached custom script with some minor extensions to the API that helped with resetting the scene between episodes.

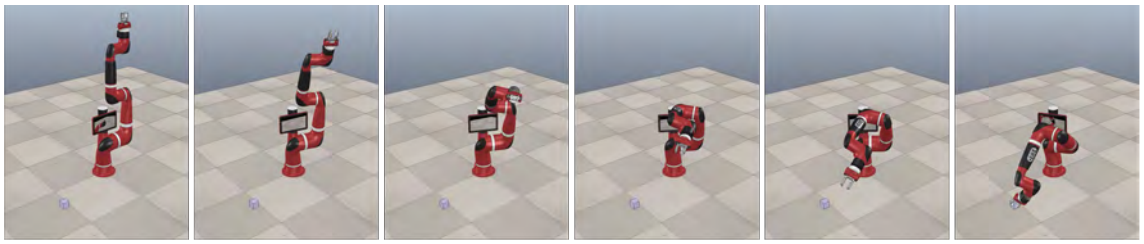

Figure 4.4: Executing the trained Sawyer Reacher policy. The robot travels towards the cube and stops

After sanity-checking the implementation with a static target, we trained a policy where the target is placed uniformly at random on the ground in a square with 0.6m sides. Within 600,000 training timesteps the policy converged. Training was successful with the robot reaching the cube's location (and staying there) in all of 50 test episodes. A sample episode is displayed in Figure 4.4.

Chapter 5

Developing Curriculum Learning

Now that we could create and successfully train agents on V-REP environments, it was time to start developing our method for solving complex tasks, for which training solely with PPO from scratch may not be enough. Ideally, we would like a policy to train with minimal human guidance. When we guide policy training based on some prior knowledge of the environment or assumption about how a successful policy would behave we risk prohibiting the robot from learning the most efficient and reliable method for completing a task.

Sparse reward functions capture this ideal. When an agent is only rewarded for completing the task, it has no constraints on how it completes it. However it is unrealistic to train from scratch with sparse reward functions. Let's take the dish rack task as an example. We have to control a robot with seven joints to reach a specific area in an acceptable orientation, navigating around the dish rack to do so. Moving the robot randomly is highly unlikely to ever place the plate in the rack. Therefore, training an initially random policy, that will only receive positive reward on completion of the task, would require unfeasibly many episodes of interaction with the environment before it could hope to consistently complete this task.

Dense reward functions offer a viable alternative to the limitations sparse rewards have with complex tasks in large state spaces. In a dense reward function instead of rewarding success we instead penalise distance from completing the task. The further an agent is from completing the task, the more penalty it receives. This way even in episodes where the agent does not complete a task, it received greater or lesser penalties at every step which can guide it progressively closer to task completion.

The problem with dense rewards is that a function's author has to mathematically define closeness to task completion, which uses their prior knowledge to affect how the agent learns. Agents trained on dense reward functions are also more likely to converge to local minima which do not complete a task as intended. The RL algorithm may converge to a policy that is close to completion and is thus not heavily penalised, while missing an optimal policy that was harder to find. We come up against this problem and discuss it in greater detail later in Section 5.3.3.

5.1 Residual Reinforcement Learning

In December 2018, two separate papers were published presenting residual reinforcement learning [16, 17]. This method is presented as a means to help an agent learn

mechanics which are difficult for humans to model, by learning them separately to those which we can model. In Johannink et al.'s work, a robot must place a subject block in between two others without knocking the other blocks over.

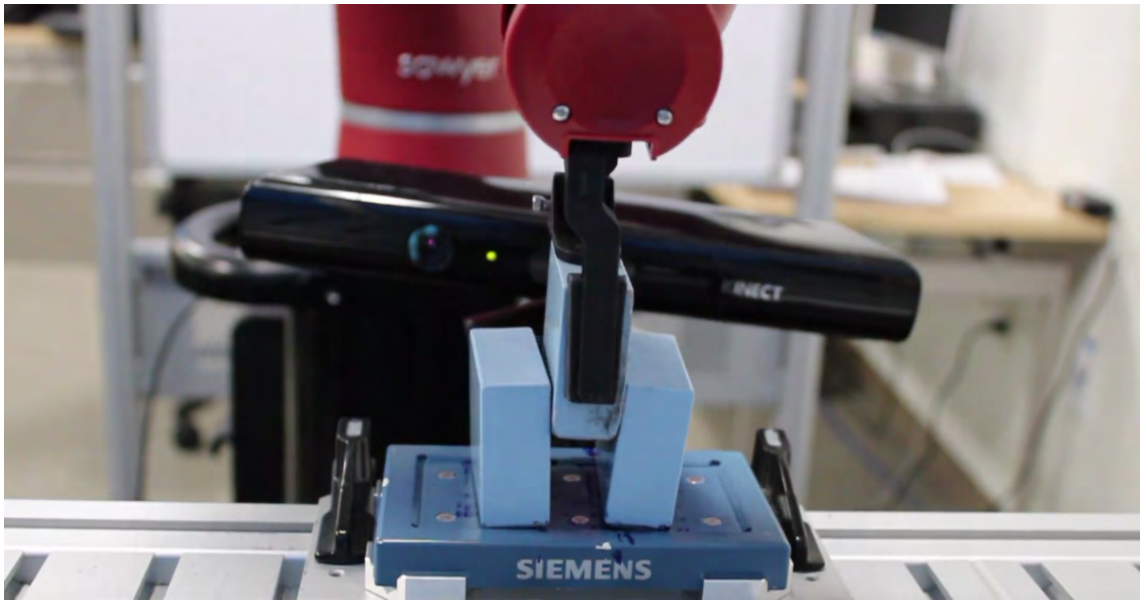

Figure 5.1: A robot attempts to place a block between two others by carefully nudging the other blocks out of the way.

It is easy to model how to move the subject to the target location in Cartesian space, but it is hard to hand-engineer a policy that can carefully nudge the blocks out of the way as it slides a block between them. A hand-engineered controller for moving the robot to the target is used as a fixed initial policy, and the residual policy uses reinforcement learning to learn to carefully interact with the other blocks. Output from the residual policy is added to the initial actiona given by the initial policy to obtain the whole action.

In this project we do not use residual RL with hand-engineered controllers in this way. The tasks we are trying to complete do not involve complex physical interactions. The potential was saw in residual RL was not its utility in learning mechanics that are difficult to model, but rather the more general concept of using multiple policies to achieve a single task. We could use this concept to train tasks that would be otherwise unachievable with RL from scratch.

In early experiments we tried to train initial policies in a short amount of time with supervised learning. These aimed to teach the robot to travel in the roughly correct direction, before residual policies could be trained to complete the task properly. Later we see how using multiple policies allows us to leverage both dense and sparse reward functions to train an agent with sparse rewards in a sensible amount of time.

5.2 Initial Policies with Supervised Learning

Our first approach was to use DAgger to quickly train an initial policy with supervised learning. The idea was that DAgger could learn to reach a static target with the help of waypoints, and a residual policy could use this to learn to reach

a target positioned at random in a small range. We modified our algorithm such that before reinforcement learning began, some initial episodes were run to train the initial policy.

Where we used DAgger before, the policy was initially trained on a single demonstration path. On 3D simulated tasks where we aim to eventually transfer a policy to reality, the initial joint angles are varied. Because of this and to keep initial policy training short, we use multiple demonstration paths to initialise the network before using DAgger to improve the policy with data gathered from real experience. After each DAgger episode the network is trained for a fixed number of epochs using all the training data gathered so far. At each episode we measure the policy network's mean squared error on a validation set. Training continues until validation set performance fails to improve for a number of consecutive episodes.

We tested the approach on Dish Rack and on Reach Over Wall. At each step, inverse kinematics (IK) are used to work out velocities the arm should be taking. Prior to crossing the wall on the Reach Over Wall task, a waypoint above the wall is used as the IK target. Afterwards, the cube is the target. Unfortunately, this approach was not too successful on either of the tasks. We varied hyperparameters such as number of demonstration paths, training epochs per episode and number of episodes without improvement before early stopping. We also attempted to use only noisy demonstration paths without DAgger.

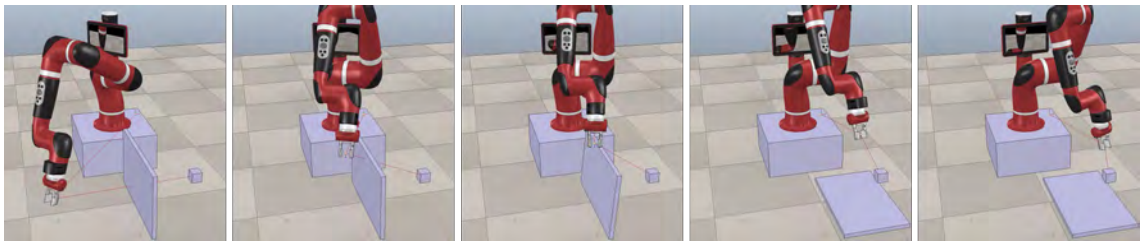

Figure 5.2: Moving the robot under the learned initial policy. The robot hits the wall and stops before the target.

The best initial policies trained from a large number of attempts would move in roughly the right direction, but still have significant errors such as failing to reach the target and missing the waypoint by a wide margin. On Reach Over Wall, this meant that the initial policy hits the top of the wall and then very slowly moves toward the cube. Using this for residual RL, residual policies learn to move close to the wall but not to cross it and reach the target when the penalty for toppling the wall is higher. When the penalty is lower, residual policies instead push right through the wall to the target. The residual policies did not complete the tasks successfully when the target was kept static, let alone with random targets.

We came to the conclusion that our approach was flawed. Trying to quickly train an initial policy on data gathered in just a few episodes was always going to produce imperfect policies that left a lot for residual policies to improve on. From this conclusion we moved to instead train the initial policy with reinforcement learning. Though this would increase training time, RL could help to improve the performance and utility of our initial policy.

5.3 Initial Policies with Reinforcement Learning

5.3.1 Dense Reward Function with Waypoints

We aimed to adapt the concept of a demonstration path to a reward-based RL setting. This meant using a waypoint in some way to encourage the robot to travel via the waypoint on its way to the target. The reward function we came up with is shown in equation 3.2.1. Like supervised learning, RL with waypoints also proved relatively unsuccessful. In fact the policies learnt over a significantly longer span of time were typically worse than those learnt quickly with supervised learning.

When playing back the trained policies, we saw that the robot would always reach the waypoint but rarely reach the target afterwards, it would usually drift off continuing roughly along the same path it had followed to reach the waypoint in the first place. The robot would not change direction to move down behind the wall towards the target in ReachOverWall, nor would it lower the plate into the slot in Dish Rack.

The failure is likely to be because the agent never gains enough experience after reaching the waypoint. PPO is an on-policy algorithm, meaning it only uses experience gained since the last policy update. After a policy update it throws away all prior experience and uses the new policy with some added noise to gain experience. Because of this, even when a policy can consistently make it to the waypoint, the agent is unlikely to reach the waypoint in every episode when exploring around that policy. Over the entire course of training, the agent never gains enough useful experience to learn to reach the waypoint and then the target.

5.3.2 Non-respondable environments

Waypoints had not proved as useful as we had hoped, so if we wanted to use RL to train our initial policies we needed to find something simpler that could be learned by RL from scratch. From our early work, we knew RL with PPO could easily learn to complete a reacher task with a Sawyer robot. As the goal in these more complex tasks was to reach a point navigating around an obstacle, we wondered if it would be possible to reduce the complex tasks to a reacher to train an initial policy. We would then add the obstacle, hoping that a residual policy would be able to learn around the newly introduced obstacle.

We once again tested the approach on Dish Rack and Reach Over Wall. Both the initial policy and the residual policy were trained with the same dense reward function (without waypoints) as described in equations 3.2.1 and 3.2.2 respectively. When training the initial policy we made the obstacle non-respondable in the simulation. This meant we could see it if viewing the environment but the robot could not collide with it. For the residual policy the environment is returned to normal. Interestingly we found this worked on Dish Rack, but not on Reach Over Wall. We will first explore the success on the former before moving to explain why this was unsuccessful for the latter task.

5.3.3 Success on Dish Rack

We trained using dense rewards as in Equation 3.2.2 with an added collision penalty, and show the average returns over time in Figure 5.3. The initial policy is trained

as a reacher policy where the rack is non-respondable in 500,000 timesteps. The residual policy begins training receiving heavy collision penalties as the rack is now in its way. Guided by the general direction of the initial policy and the distance component of the reward function, it eventually learns to complete the task without touching the rack in 96% of 50 test episodes.

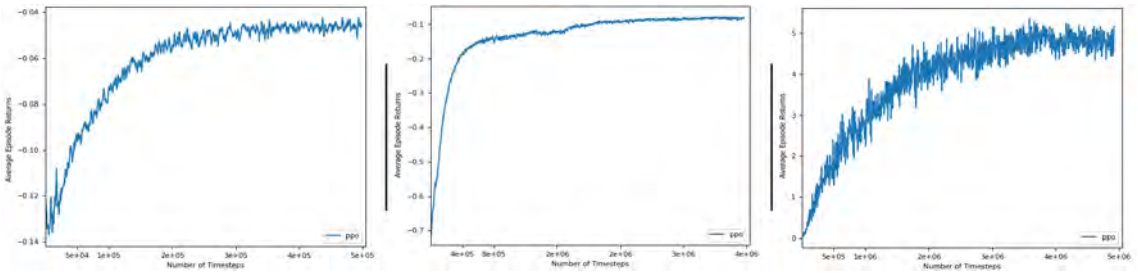

Figure 5.3: Left: Training the initial reacher policy with a non-respondable dish rack. Middle: Using the initial policy to train a residual policy to complete the task with a respondable rack. Right: A final policy trained with sparse rewards

Following this we use these two policies as the initial policy for another residual policy, trained instead with sparse rewards. Initially, exploring around the policy yields low returns but eventually, within 5 million timesteps the policy has converged. The returned policy is also 96% successful. Note how training graphs for sparse reward functions are not nearly as smooth as for dense rewards. This is because small changes in policy lead to much larger changes in rewards when they are the difference between making it to the target point or not.

Action Scaling

Up until now, we executed actions output by the policy without any modifications. We used a control penalty in our dense reward function to encourage the arm to make smaller movements. However, the sparse reward function does not feature this penalty. As such, in the sparse policy trained here, the most noticeable change over its initial policy is that it reaches the target much faster.

Our reward function gave a score of 0.1 for each successful step. In the right hand graph of Figure 5.3 we see that the overall return ends at around 5. The episode length is only 64 steps, each representing 0.05 seconds in the environment. The robot only takes around 14 steps, or 0.7 seconds to move to the rack and finish lowering the plate into it.

This is much too fast, but our thinking was that we could let this happen in simulation to keep episodes shorter and scale the actions when moving to reality. Out of curiosity, we ran the policy with both scaled and clipped actions to see what would happen. The policy is 0% successful in both cases. The likely reason for this is that because the sparse policy takes such large steps, when moving smaller distances the agent ends up in unknown states.

It was clear that the size of actions needed to be limited to ones that are sensible in the real world during training. This could not be done after. So in all subsequent experiments, we used a wrapper to transform the actions to within the environment's specified action range. We tried both a *tanh* scaling function and simple clipping.

Clipping works best as non-linear scaling has an adverse effect on a residual policy's ability to explore evenly around the initial policy during the early stages of training.

5.3.4 Failure on Reach Over Wall

For Reach Over Wall, we did not get as far as training with a sparse reward function. When using an initial policy trained with a non-respondable wall, the initial policy failed to learn to climb over the wall. Instead, where the weight parameter λ_3 for r_w in equation 3.2.1 is small, the robot barges through the wall to the target. When that parameter is larger, the learned policy moves very close the wall before coming to a stop. It does not learn to navigate around it.

We hypothesised that the reason for the success in one task and the failure in another was due to the difference in the level of deviation from the initial policy required to complete the task. Navigating around the dish rack with a plate requires a smaller deviation from a straight line path to the target position than rising over the high wall.

To test this hypothesis we created an alternative (and unrealistic) version of the dish rack task, shown in Figure 5.4 where the prongs were three times as high. This made no changes to the non-respondable environment so we used the same initial policy from earlier. A residual policy was trained on top of this with the same dense reward function aiming to lower the plate into the rack. Sure enough, the policy does not learn to put the plate in the rack. It instead learns to stop in front of the dish rack at the same height as the target, minimising the distance between itself and the target as much as possible without putting the plate in the dish rack.

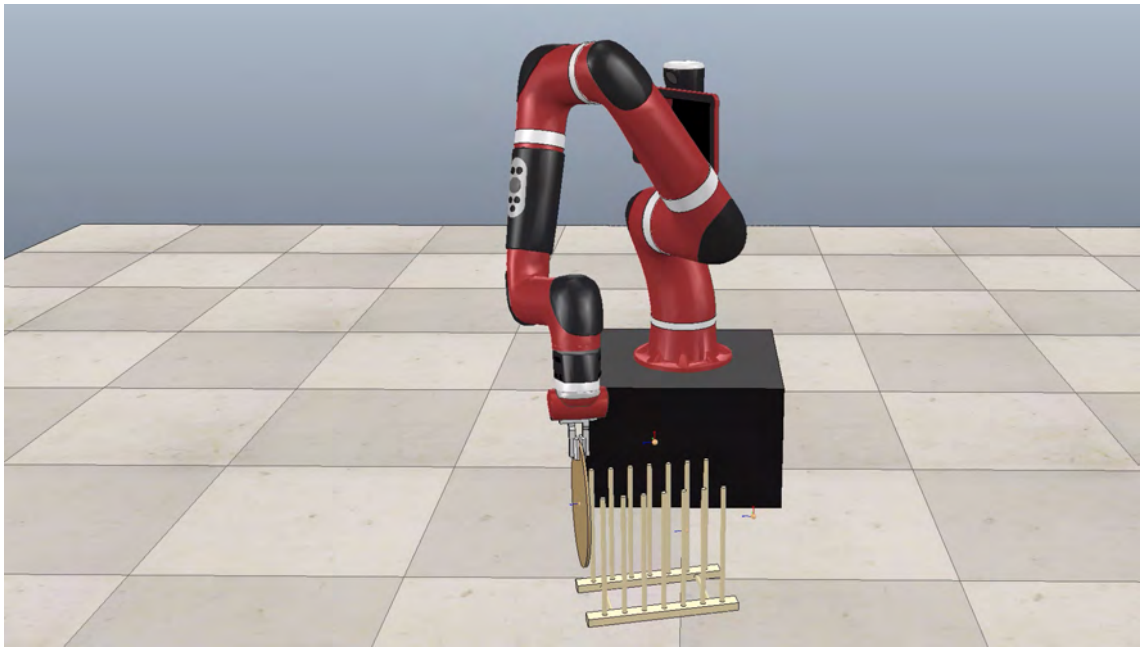

Figure 5.4: A surreal version of the dish rack task where the prongs that must be navigated around are three times as high compared to the regular task.

Limitation of dense reward functions

What is important to note about this last result is that it is not necessarily an example of the policy getting stuck in a local minima due to insufficient exploration. Let's compare two possible policies side-by-side. The first is the one we have trained where the plate moves in a straight line toward the target, stopping in front of the rack. The second is the policy we were hoping to train, where the plate moves over the rack and into the correct slot. We will examine the distance component of the reward function and assume the control, orientation and collision components to be the same. As we are using a dense reward function, while this second policy moves over the rack at the beginning of an episode it is receiving worse rewards each step than the policy which is moving directly towards the target. Only when the second policy begins lowering the plate into the rack does it become closer to the target than the plate being moved under the first policy. This plate has stopped in front of the rack at this point and receives the same reward every step. When the plate under the second policy comes to a stop it too will receive the same reward every step.

So which policy receives the maximum return over the course of an episode? That is determined by the length of the episode. If the episode is not long enough, the policy we would like to train will not spend long enough at the target to gain an overall higher return than the policy which was trained. The deviation at the start of the episode does not lead to a larger return and is therefore not worth it. The first policy is in fact optimal. For the purposes of this report we will refer back to this as the Undesired Optimal Policy Problem.

This problem can be solved for any task by simply increasing the length of an episode, but this is not a practical solution. In environments where reaching the target requires significant deviation, and where a straight path can bring the subject very close to the target, the episode would have to be very long for the optimal policy to be the one we hope to train. With very long episodes training time would be significantly increased. There is also no guarantee that these would stop the policies from getting stuck in a local minimum, converging to the first policy we were trying to avoid.

5.3.5 Summary

So far we have found it is possible to train residual policies to avoid obstacles, albeit with the constraint that those obstacles are not too large. This means our approach in its current state is not general enough to solve all similar tasks. But what if instead of moving directly from no obstacle to the full task, we progressively increase the size of the obstacle? This is the idea that inspired Curriculum Reinforcement Learning, this project's main contribution.

Chapter 6

CuRL: Curriculum Reinforcement Learning

In this chapter we will discuss the concept and implementation of CuRL in detail. The algorithm aims to provide an efficient method of training a policy that can complete general obstacle interaction tasks. To do so, we reduce the task to a reacher with no obstacle. We train an agent to complete this reacher task. Then we continue training, progressively adding parts of the obstacle until the agent can complete the task with the whole obstacle. However, to avoid eventually encountering the problem in 5.3.4 we must first redefine how we reward our agent.

6.1 CuRL Reward Structure

Switching to sparse rewards early

In Section 5.3.1 we trained a dense reacher policy, followed by a dense residual policy on the full task. We then train a final residual policy with sparse rewards. However, if we used a dense reward function as we increased the respondability of our environment, we could eventually encounter the undesired optimal policy problem. Completing the task many not be optimal under our reward function.

To work around this, in CuRL we use dense rewards only to train the initial reacher policy. We train a residual policy to complete the reacher task with sparse rewards. We use that same reward function for every policy afterwards, as the obstacles in a task grow. The drawback of this is that changes to the environment must be small. The agent should be able to reach the target in some episodes when exploring a new environment using a policy trained in a previous environment with smaller obstacles. If the change is too large and it cannot regularly reach the target, it will have nothing to learn from.

Aside from its suitability for obstacle interaction tasks, the other advantage to using sparse rewards earlier is that it makes reward functions much easier to create. To define the dense reward function we only need to decide which positional and rotational axes we are concerned with. The reward function is then a dense reacher penalising the difference between the subject's current pose and the target pose. The sparse reward function is a positive reward for achieving the target pose to within a chosen margin of error.

This is not to say tasks that require more complex reward definitions could not be trained with curriculum learning. The sparse reward function could consist of

penalties for colliding with obstacles or making unnecessary movements. When doing so, more care will need to be taken when weighting the reward function to ensure the policy does not avoid the target altogether.

6.2 Respondability Sphere

To control how the obstacle grows we introduce the concept of a "respondability sphere". It allows us to grow obstacles in a similar and uniform fashion for all tasks, even though the shape and size of obstacles will vary greatly between tasks. The sphere helps define which parts of an object are respondable in each stage of training. Everything inside it is respondable and everything outside is non-respondable. This excludes the robot, which is always respondable.

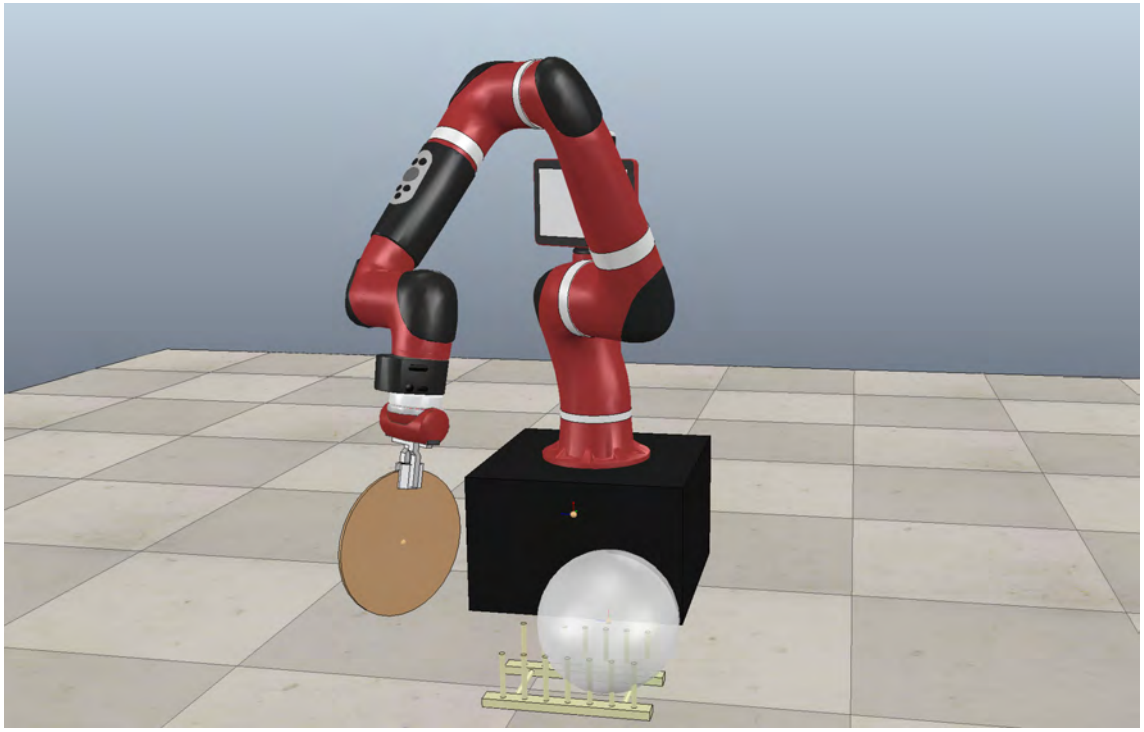

Figure 6.1: A sphere positioned for reference at the target. Under a respondability sphere of the same radius, the part of the rack inside the sphere will be respondable. The part outside will not be.

Starting with a single sphere

Initially the idea was conceived as a single sphere, centred at the target position with its radius growing during training. A sphere was chosen as it meant that everything within a distance equal to the radius would be respondable. Parts of obstacles are added uniformly relative to their distance to the target.

The respondability sphere should start small and grow by a user defined `step_size` between training runs. Steps should be small enough that the agent can still regularly reach the target when exploring under the current policy after the environment's sphere increases in size. This way the agent will still receive some sparse rewards to work from. When setting the step size, the user should also consider

that leaving it too small means there will be more training runs than needed. This will make the overall training duration unnecessarily long.

Two Spheres

In early experiments, we realised that using only a single sphere positioned at the target can be problematic. Learning can fail if the subject of the scene is not the end of the robot, but instead a sizeable held object. In the Dish Rack example, when the prongs closest to the target pose became respondable during training, the 11cm radius plate suddenly needed to be lifted much higher to enter the rack over those prongs. Similar to how training failed on large obstacles, if the subject is too large the agent must learn to deviate further from its current policy than it can in a reasonable length of training.

We overcome this problem by using two respondability spheres. One is centred at the target and the other at the centre of the subject, in this case a plate. In our experiments we give them the same radius and grow them at the same rate.

6.3 Alternative Approaches

6.4 CuRL in Theory

Let's start by looking at ReachOverWall as an example. In our experiments other approaches cannot solve this task. In our experiments other approaches cannot solve this task. We detail this more in our evaluation in Section 8.1.3, but for now let's focus on how CuRL can be used to teach policies something that training from scratch cannot. Figure 6.2 helps to illustrate this.

The circles indicate parts of the wall that would be respondable under different sized respondability spheres centred at the target sphere. The trajectories of matching colours show how the policy should learn to move given the effective respondable shape as an obstacle. The idea is that an initial policy represented by the red trajectory can be transformed into a final policy represented by the green trajectory over a series of training runs.

The general approach is detailed in Algorithm 1. We have a simulated environment of the task and an initial reacher policy trained with dense rewards. We use the `update` function to set the environment's respondability sphere's radius to 0. This way no obstacles are respondable. A residual policy is initialised using sparse rewards. After the initial training run is complete, the respondability sphere's radius is updated to a starting size. The next run trains the same residual policy network on the updated environment. The respondability radius is increased by `step_size` for each subsequent training run until `radius >= end_radius`. The final run trains the residual policy network on the fully-respondable environment. The two final policies, initial and residual, can now be used to complete the full task.

Pruning unnecessary training

The use of a `start_radius` and `end_radius` may not be that intuitive. In it's purest form curriculum learning would start with the respondability sphere radius at 0 and increase by `step_size` until every object in an environment is fully respondable.

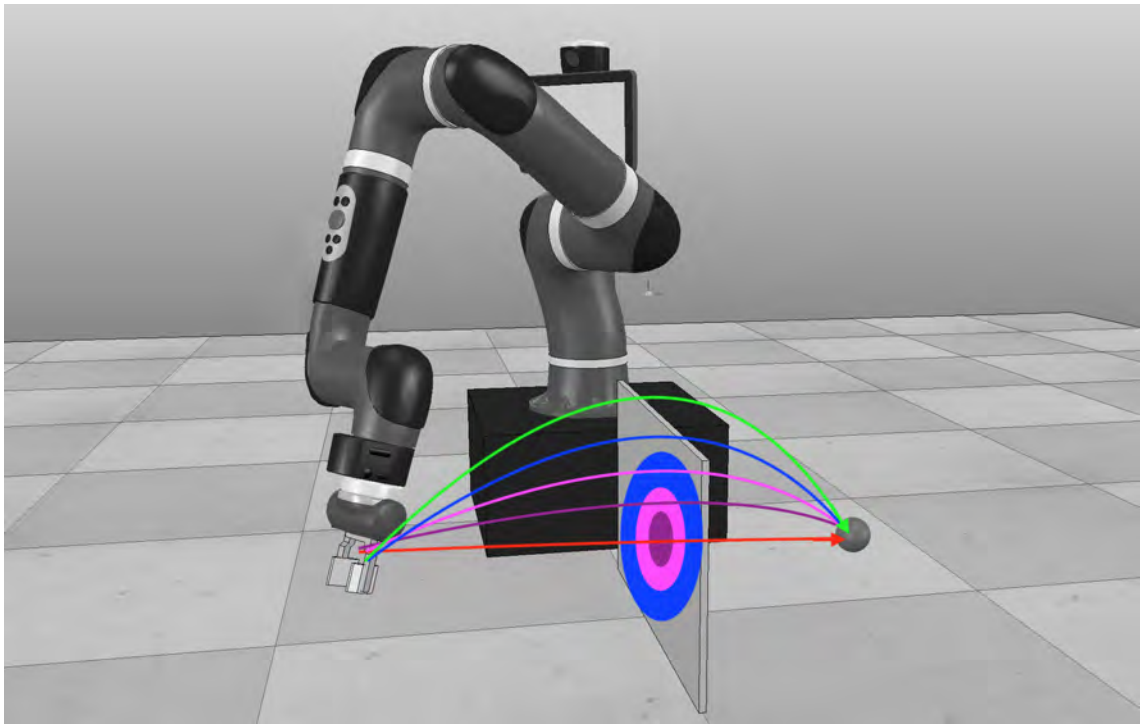

Figure 6.2: Diagram illustrating the general idea behind CuRL.

However this is likely to take a very long time. In practice, there is a time after which the current policy is suitable for the full task, but some elements in an environment are still not respondable. Likewise when the obstacle is not close to the target, there could be a time when the respondability sphere will not enclose any part of the obstacle. We can skip these parts to remove unnecessary training.

The DishRack task provides a good example. Once the sphere's radius is large enough for all of the plate and the four prongs closest to the target to be respondable, the policy has learnt to move above the rack high enough to not hit. We do not need to continue slowly increasing respondability until the rest of the prongs and the base are respondable. The robot already avoids these with the plate, just by keeping it high enough to avoid the parts that are respondable.

6.5 CuRL in Practice

In practice it is not possible with V-REP or any other simulation software we know of to have fine control of a shape's respondability at the vertex level. A shape in an object is either respondable or non-respondable. To create a part-respondable object, it must be split in to multiple shapes which can have their own seperate respondability. It is not possible to take a single shape and set the respondability for the individual vertices it is composed of. We could separate shapes into their vertices and manually set the respondability, but this would take unfeasibly, not to mention unnecessarily, long

Instead, we get around this software limitation by splitting complex objects into their constituent parts more coarsely. When each component is mostly inside the respondability sphere we make it respondable. For simpler shapes like the plate which is a cylinder, we simply set its radius to that of the respondability sphere

Algorithm 1: Curriculum Learning

Input : Environment env, Initial Reacher Policy ip, step_size, start_radius, end_radius
Output: Residual Policy rp

Begin training the residual policy on the non respondable environment;
radius ← 0;
env ← `update`(env, radius);
rp ← `train`(env, ip, *None*);
for radius ← `range`(start_radius, end_radius, step_size) **do**
    env ← `update`(env, radius);
    Continue training the same residual policy network on the updated environment;
    rp ← `train`(env, ip, rp);
end
env ← `update`(env, ∞);
rp ← `train`(env, ip, rp);
Result: rp

positioned at its centre. This is acceptable for the time being as we are not training with images and so this visual difference does not matter.

The process of training then remains largely the same as in Algorithm 1 then. The difference is that we don't have an `update` function. Instead we create versions of the environment corresponding to all the radii used during training. Between training runs we close the old environment and load a new one with respondable objects within the larger respondability sphere.

6.5.1 Summary

There are clearly many improvements which can be made to this process in the longer term. Editing the underlying code or creating a plugin for V-REP which would allow fine, programmatic control of an object's respondability was not feasible during this project. With the ability to edit respondability while training is ongoing comes many more possibilities for how the curriculum progresses. For example rather than using a rigid step size, a stochastic respondability radius could show a mixture of easier and more difficult environments to the same policy to actively determine an efficient step size during training. We discuss this idea and more for future work in Section 9.

Chapter 7

Transfer to a Real Robot

One of the project's initial core goals was to have a real robot complete the complex tasks we train in simulation. This sim-to-real aspect of the project became more of a secondary goal when we chose to focus more on developing a novel, general approach for robotic reinforcement learning, rather than just successful completion of individual tasks. Learning robust sim-to-real transfer is a task worthy of a whole project in itself, and one that has been difficult to balance with continued and extensive experiments in simulation. That said, we still spent a significant portion of time attempting to demonstrate curriculum learning's success in a real environment. We will detail our approaches here.

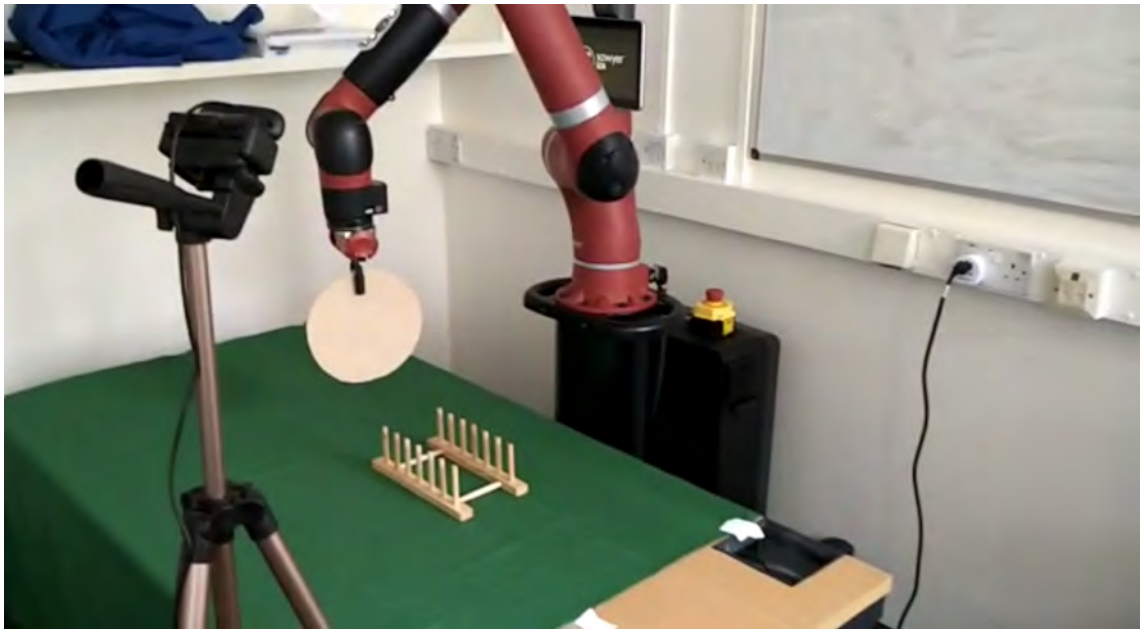

Figure 7.1: The real task environment. The Sawyer robot sits on a stand overlooking a table as the camera looks on.

7.1 Pose Estimation

Our first successful policies on Dish Rack used observations containing joint angles, absolute x, y rack position and rack rotation θ around the x axis. On a real robot,

we have joint angles and images from an external camera available as observations. Therefore, all that's needed to use our simulation policies are the observations concerning the dish rack. If we had an oracle taking images as input and returning the correcy (x, y, θ), our policies would work as well in reality as they do in simulation. As an alternative to length training in simulated environments with image observations, we set out to train a pose estimator with minimal error.

7.1.1 Domain Randomization

We train our pose estimator on simulated images, the reason for this is two-fold. Firstly, we can generate many more simulated images with much greater variation than real images in a short span of time. Secondly, with real images our human measurements of the true pose will be noisy. In simulation the positions are exact. However, for a pose estimator trained on simulated images to be useful in reality, it has to be accurate on real images. To help with this we use domain randomization, discussed in Section [Background]. This can help the pose estimator learn to ignore differences in the environment and focus simply on the relevant features present in both simulated and real images.

In the Dish Rack example we begin by randomising camera position and angle, lighting position and intensity, as well as colours of the plate, rack, back wall and cloth. This cloth covers the table to increase the difference in color between the wooden dish rack and its background. This randomisation is of course in addition to that relevant to the full state of the task: rack pose and joint angles. The camera and lights are positioned uniformly at random within small cubes centred at their default positions. All colours are sampled from a uniform distribution centred at the object's default colour. Light intensity is varied in a very crude way by altering the number of lights active, but this still produces a good range of lighting in simulated images.

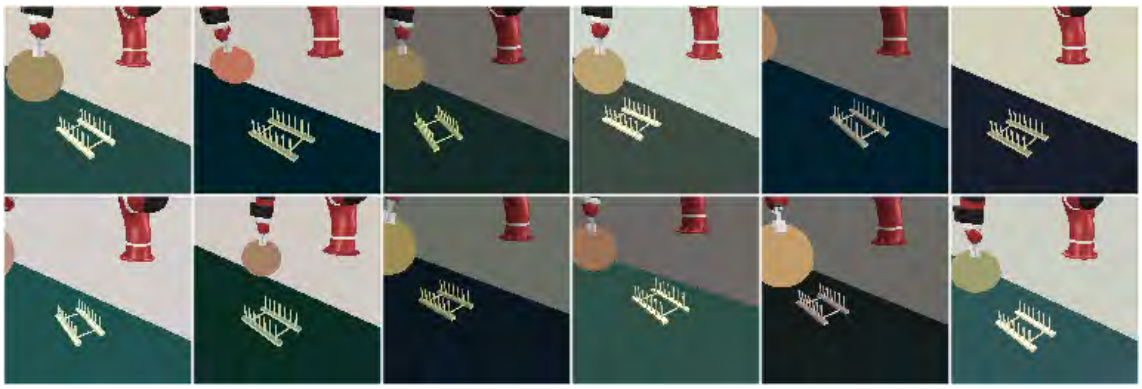

Figure 7.2: A sample of simulated images with domain randomisation as detailed in Section 7.1. Though simple these images were relatively effective, but were improved upon later.

7.1.2 Selecting an architecture

Ikostrikov's implementation had a policy architecture for reinforcement learning with image observations. This was a convolutional neural network taking 84×84

resolution images as observations. It had three convolutional layers with 32, 64, and 32 channels respectively, followed by a fully connected layer with 512 nodes, and a final fully connected output layer. This network was used in training agents to play Atari games, so while it was not designed for pose estimation we decided to use it as a starting point seeing as it was available.

We developed a script to generate training data from an environment. The script rolls out a pre-trained policy, capturing images and recording the true rack pose at every step. A policy is used so that the images show the robot in positions it is likely to be in. The pose estimator should learn that it will not always have a perfect view of the dish rack. Sometimes there will be a plate in it.

After using the new script to generate a sizeable training set, we trained the network using the Adam optimiser and the mean squared error loss function. The network trained to convergence but only really learned to output the mean rack pose. This is a sign of underfitting, one source of which could be that the network architecture was not large enough for the task.

We searched papers on pose estimation looking for a more suitable architecture, and eventually found *Domain Randomisation for Pose Estimation* [40]. In this paper, Ren et al. aim to estimate the pose of objects in real images, after training on simulated ones. The paper is interesting not only for describing its architecture, but also for proposing a unique loss function. We use this loss later during training, but first lets look at the architecture.

7.1.3 Modified VGG-16

The architecture used in *Domain Randomisation for Pose Estimation* is similar to the 16 layer variant of VGG [41]. The CNN, illustrated in Figure 7.3 features 13 convolutional and 3 fully connected layers. The convolutional layers are organised into 5 groups. The first layer in all but the last group doubles the number of channels, starting with 64 and increasing to 512 by the 4th group. A ReLU activation function is used between each layer and a max pooling layer sits between each group. All convolutional layers have a 3×3 kernel and use zero padding of size 1. This means that the resolution of the images does not change after each layer. Instead, max pooling layers with a 2×2 kernel are used to half the resolution.

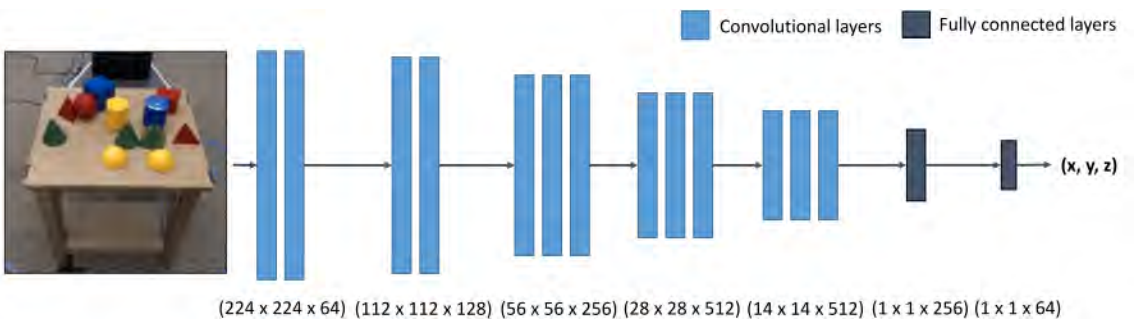

Figure 7.3: The model architecture, adapted from VGG, used for pose estimation. Using Dish Rack as an example, the input is a 128×128 image and the output is the rack's predicted (x, y, θ).

7.1.4 Initial Tests

On a dataset of 8192 images and absolute rack poses, we left out 256 samples as a test set and trained on the rest. Using random weight initialisation and a learning rate of 1×10^{-4}, our architecture achieved a mean distance error of **11.43 mm** and a mean rotational error of **0.00938 radians**. As a first attempt without any parameter turning this was a good result, and showed the suitability of this architecture for pose estimation. We then performed a search over hyperparameters to train the best pose estimator possible. This is detailed in Section 8.2.3, along with further experiments and tests on the real robot.

7.2 End-to-End Policies

Though pose estimation has proved successful on Dish Rack in simulation, it is not really suitable for the aim of our project to show an effective sim-to-real strategy for a general obstacle interaction task. As the current state-of-the-art cannot train a perfect pose estimator, using pose estimators limits CuRL to tasks which can tolerate a certain degree of error in estimations for target and obstacle poses. An end-to-end policy, though time consuming to train, would be a better approach for demonstrating CuRL's suitability to real-world tasks.

If instead of full pose information, our policy could take images of the scene and joint angles as input and return joint velocities, it would be usable in the real world. Therefore, we aimed to use CuRL to train an end-to-end controller from scratch, with visual domain randomisation to aid policy transfer.

7.2.1 Challenges

End-to-end training takes far more time than training with full state information. Training with RL generally requires more timesteps as the function is harder to learn and a deeper network is used. To add to that each policy update takes a lot longer as well. To give a ballpark figure, in experiments where a full state agent takes around 40 seconds for a single policy update, the end-to-end agent takes around 180 seconds, 4.5 times as long. To fully train a full end-to-end policy with curriculum learning will take a length of time in the order of days.

After a policy has trained, it may still be unsuitable. In our experiments, training the initial reacher policy took around a day and the policy only moves in generally the correct direction. It certainly didn't work out where the rack was, and that it should move there. In this particular experiment the architecture was unsuitable, so to retrain we needed to change the architecture and train again for another day to try a new one. The point being that training end-to-end is verging on being unfeasibly time-consuming.

To make matters worse, we can only properly test whether the policy is capable of sim-to-real transfer once it has completed the curriculum and is usable on the real task. If we find that different domain randomisation is needed for example, we would need to change our approach and retrain.

In summary, everything that can be optimised with a full state simulation should be. Even if a user's only concern is training a policy for use in reality, they should still use trial runs in simulation to find suitably large step sizes and where training

can be pruned. This will help minimise the number of retrains needed. For our project, in spite of these challenges we still make attempts at end-to-end learning as it is important to demonstrate CuRL's utility for solving real world tasks.

7.2.2 Architectures

Early in the project we naively thought that the out-of-the-box convolutional policy network in Ikostrikov's implementation could make some headway at end-to-end training. This network had only 3 convolutional layers with 32, 64 and 32 features respectively, and 2 fully connected layers. It only learned to move in the general direction of the rack but couldn't distinguish its position and move towards it.

This was not a particularly surprising result. The architecture has proven successfuly on the Atari games baselines but of course the features (such as a character sprite) in those images are much more uniform across images and therefore easier to recognise. We needed something more complex.

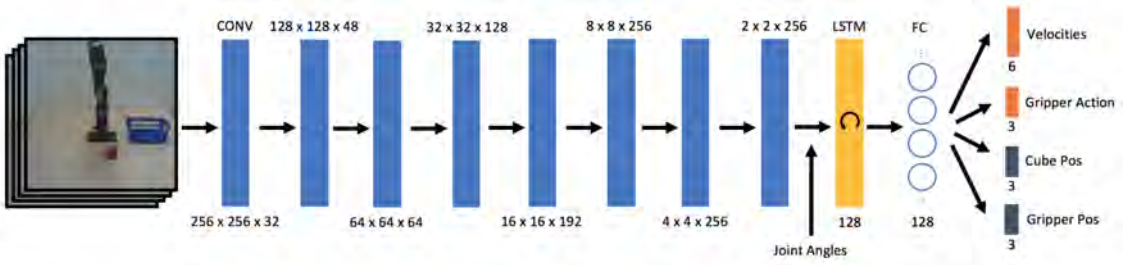

Figure 7.4: Network architecture diagram from *Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task* [4, p. 5].

We tried two more architectures. One was a version of the pose estimation architecture from Figure 7.3, modified to take joint angles as well as output from the convolutional layers as input to the fully connected layer. Another, shown in Figure 7.4 was used to successfully train a robot with supervised learning in simulation to complete a multi-stage grasping task in reality. We removed a convolutional layer to account for the difference in resolution, and changed the LSTM to another fully-connected layer. The LSTM was shown to be necessary for the task in that paper as it was a multi-stage task. Here it is unnecessary. We also changed the output layer as we only needed joint velocities.

We attempted to train initial reacher policies for Dish Rack with both these architectures. While better than the simple CNN, neither of them were actually successful at end-to-end training with RL. They could travel generally towards the rack but again never decisively to the target.

7.2.3 End-to-end with Supervised Learning

The prohibitive length of time taken to train each end-to-end policy made continuing in this way in search of a suitable architecture and level of randomisation entirely unfeasible. Attempting to train policies in this way was probably a mistake. Though it would have been nice to show CuRL being used to train an end-to-end controller, RL is not the only way to train an end-to-end controller. A more efficient method is with supervised learning.

Using our trained full-state policies, we can generate datasets similar to the one used for pose estimation, where the target values are the actions executed by the policy instead of poses. At time of writing we have trained one end-to-end policy in this way, and tested it on the real robot. The policy used the same modified VGG architecture as in Section 7.2.1 and was trained with the same dataset used in Section 8.3.6.

When viewing the policy in simulation, it seems to perform competently, not always placing the plate in the correct dish rack but always coming close. Over 50 test episodes in simulation, the policy is 38% successful. This would ideally be better but was good enough for us to test in reality and at least observe the differences arising from sim-to-real transfer. We do this in Section 8.3.6.

Chapter 8

Evaluation

In Section 5.3.2 we show how it is possible to train a residual policy to complete Dish Rack given an initial reacher policy. Given this information we knew that whatever form CuRL took it had to work on Dish Rack as a bare minimum. Furthermore, as our coarser approaches can complete it, solving the task with CuRL should be easy. Therefore we used Dish Rack as our main task for developing and tuning CuRL whilst keeping generality in mind. We can then test the method's performance on Reach Over Wall to test that it overcomes the pitfalls encountered by simpler approaches.

8.1 Dish Rack

We want the robot arm to place the plate in the rack and stop. In this task we define the plate to be in the rack when it is within 1.5 cm of the target position and within 0.1 radians (5.7 degrees) of the target orientation. We define the policy to be successful in an episode if the plate is in the rack on the last step of an episode. We evaluate performance on dish rack by percentage of successful episodes when executing the trained policy on 50 seeded test episodes.

Our most successful policy on Dish Rack achieves a **96% success rate** over the 50 episodes, using the hyperparameters shown in Table 8.1 below. This is the curriculum used in all comparisons later in this chapter. It features the non-respondable environment, the fully-respondable environment and 5 intermediate steps. Including the initial reacher, this means that the policy is trained for 4 million timesteps in total. Optimising the step size or different methods for advancing the curriculum may greatly improve this training duration.

Steps per training run	500000
Start radius	7 cm
End radius	11 cm
Step size	1 cm

Table 8.1: CuRL hyperparameters used when training the most successful Dish Rack policy

Appropriate start and end radii can be selected intuitively and do not need optimising. The step size was also easy to find and though it could likely be increased we choose not to explore optimising it, as recreating intermediate environments for

new step sizes is time consuming. The results would also be task specific and provide little insight to CuRL in general. We hope coarse step sizes will quickly be rendered obsolete by future work discussed in Section 9, should it go ahead. However, we did explore appropriate number of steps per training run. We trained with a varying number of timesteps and display their results in Table 8.2.

Steps per training run	Success over 50 episodes
$200,000$	84%
$500,000$	98%
$1,000,000$	88%
$2,000,000$	54%

Table 8.2: Effect of varying timesteps per training run on Dish Rack

As we can see lower steps per training run not only mean we train on the full task faster but they help the policy learn better. Many steps leads to fully converged policies after each step which are focusing more on exploitation than exploration. With 500,000 steps and below the policies have not converged and the next training run picks up where the last left off, now on an environment with an increased respondability radius.

8.1.1 Step Size Necessity

Section 5.3.2 shows that when using dense rewards it is possible to train a residual full task policy on an initial reacher policy. With CuRL this is not the case. We use sparse rewards to avoid the Undesired Optimal Policy Problem and due to the absence of guidance provided by a dense reward setting, the maximum amount of deviation learnable in the same training time is smaller. This is why we use a CuRL step size even on Dish Rack. Here we show why this is necessary.

Our best policy trained with CuRL uses a step size of 1cm and a starting respondability radius is 7cm. We do this because all of the rack tongs remain non-respondable until the radius is 6cm. There is no need for the policy to change until the radius is greater than 6cm. The policy achieves a non-zero mean reward at the start of training which it improves upon in a consistent upward trajectory.

Figure 8.1 shows two training graphs from when we did not use a step size. Instead we step immediately from the non-respondable environment to the full task. The left-hand training graph is from a policy trained early in CuRL's development. As a policy trained with dense rewards could step directly from the non-respondable to the full task, we tested if sparse rewards could do the same. Given 1 million timesteps to train, it barely manages to consistently achieve non-zero average returns. This result was key in our realisation that sparse rewards require a finer curriculum than dense rewards.

During our experiments with CuRL we found that using slightly longer episodes helped the agent learn on Dish Rack. Doing so improves the chances of the robot completing the task and earning sparse rewards early in training. For the experiment in the left-hand graph we use an epsiode length of 32 steps. Now for the results with CuRL presented above we use 48 steps per episode. We use the same number of steps when training a more recent experiment shown in the right-hand graph.

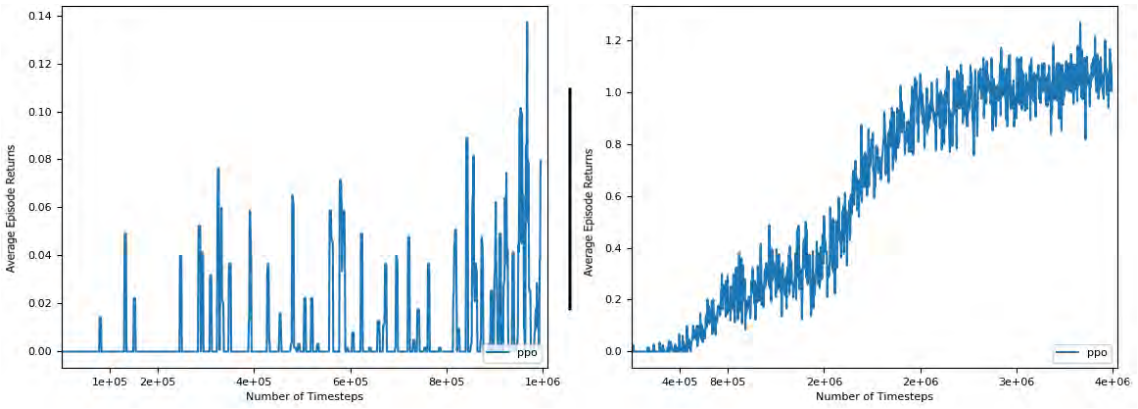

Figure 8.1: Training graphs when stepping immediately from the non-respondable environment to the full task.

When training starts we have a policy successful on the non-respondable environment. The best CuRL policy trains for another 3 million timesteps from this point. We allow a maximum of 4 million timesteps for the experiment in the right-hand graph. The longer episodes help and the policy consistently achieves non-zero average returns after around 400,000 timesteps. When training ends, the policy receives good average return and is very close to convergence. However, it is still only 82% successful over 50 test episodes. Using a CuRL step size helps converge to a better full-task policy in a shorter timespan.

8.1.2 Type of Observation

Earlier in the project we use absolute observations of target poses. This is not the only way of representing environment state information. We realised that if we were to use the relative observation between the subject and the target this might provide more useful information to the agent.

We compare the performance of 2 policies trained on the full curriculum with absolute estimations, and 2 trained with relative observations, in Table 8.3. As we can see, using relative pose estimations leads to policies that are much more consistently successful. In all experiments following this result we used relative observations.

Steps per training run	Abs. Obs.	**Rel. Obs.**
200,000	70%	84%
500,000	76%	98%

Table 8.3: Success rate of policies trained with absolute and relative observations over 50 test episodes.

8.1.3 Summary

The results here are good, but we developed and tuned CuRL on the Dish Rack task, so it is important to use it on other tasks to check that our method also works well on them. The first task we want to use it on is Reach Over Wall, to show that it has overcome the problems encountered earlier in the project.

8.2 Reach Over Wall

In this task we want Sawyer's gripper to be inside the target sphere. We define this to be when the gripper's centre is within 1.5 cm of the target position. We are not concerned with orientation here. We define the policy to be successful in an episode if the gripper is inside the target sphere in the final step of an episode. Perormance is evaluated using success rate over 50 episodes as before.

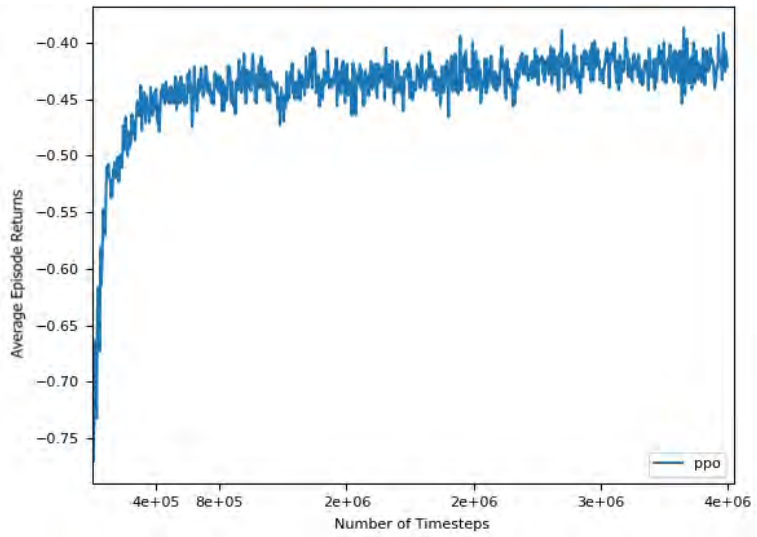

Figure 8.2: Training graph for a policy trained with dense rewards on Reach Over Wall.

8.2.1 Failure of other methods

We have already shown how training a residual policy on an initial reacher policy using dense rewards fails at this task. In another experiment conducted early in the project, we attempted to train with dense rewards from scratch. Here we show that training in this way also fails.

We train a policy using PPO with our dense reacher reward function. In the training graph, shown in Figure 8.2 we show how we allowed the policy to train for 4 million timesteps in case it does manage to get past the wall, but it only needs 1 million steps to converge. When rolling out the policy, shown in Figure 8.3 we see that the robot consistently moves to the wall and stops. It is 0% succesful at the full task, as it does not learn to travel over the wall and reach the target.

This policy was trained with a static wall. We did this to remove the added complexity of a topplable wall, hoping that we could use residual learning on a successful policy to transfer the policy to an environment where the wall could be toppled. Training with a dynamic wall from scratch also fails. It behaves similarly to in 5.3.3 where for high weighting to the wall's orientation penalty the robot stops before the wall and for low weights it bashes right through.

8.2.2 Success with CuRL

In Reach Over Wall, the obstacle does not move with the target. As such, in each episode the obstacle is at a different distance from the target. As we cannot work

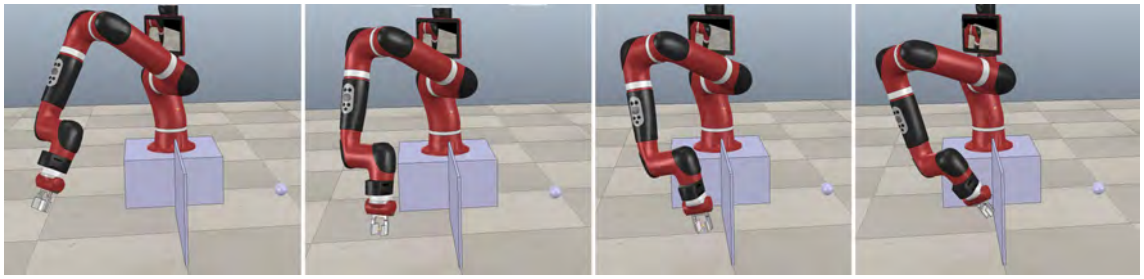

Figure 8.3: Left to right: Frames from a video of a rollout of a policy trained with dense rewards on Reach Over Wall.

out which parts of the obstacle should be respondable and which shouldn't be given current simulation abilities, we choose to set the respondability sphere at a fixed point. This is the centre of the range that the target can appear in. We then extend the respondability radius until part of the wall is respondable, shown in Figure 8.4, and set this as the start radius.

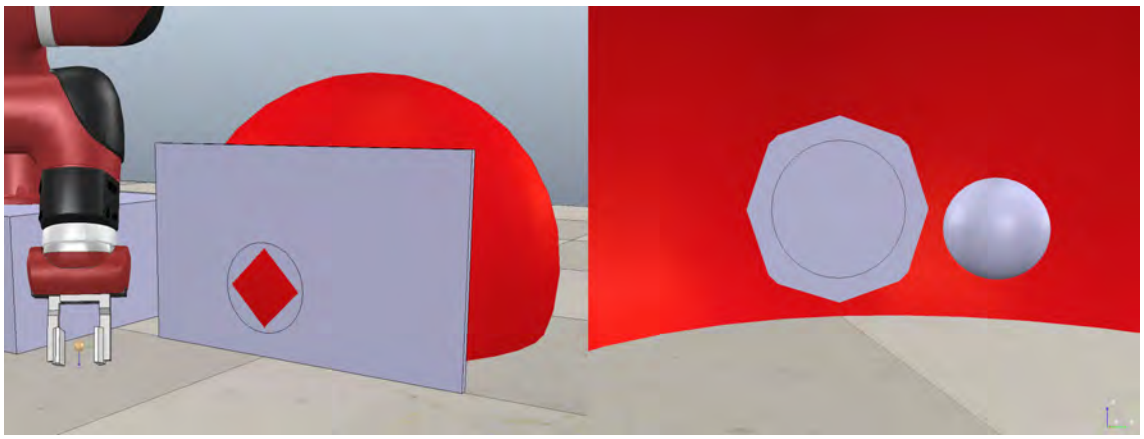

Figure 8.4: Left: The respondability sphere covers part of the wall, seen from the front. Note that the part protruding from the wall looks like a red pyramid due to vertices coarsely approximating a sphere. Right: Looking at the back of the wall from inside the sphere. We use a respondable cylinder, the ends of which are seen as circles on the wall, to represent the respondable part of the wall.

As Reach Over Wall is a more difficult task than Dish Rack, it required a more substantial curriculum. From a start radius of 300mm, we use a step size of 14mm until the respondability sphere is 440 mm and almost all of the wall is respondable. Figure 8.5 shows the respondable parts of the wall in some of the curriculum's steps.

We allow the policy to train for 400,000 timesteps on every step of the curriculum. This means that over the total 13 steps of the curriculum (14 including initial reacher policy), the policy trains for 5.2 million timesteps (or 5.6 million including initial reacher). The final policy is successful in 90% of 50 seeded test episodes. Samples showing how the policy is augmented over training are shown in Figure 8.6.

8.2.3 Summary

Reach Over Wall was originally intended to be a simple extension to Reacher for trying residual reinforcement learning methods on a non-trivial task. However, it

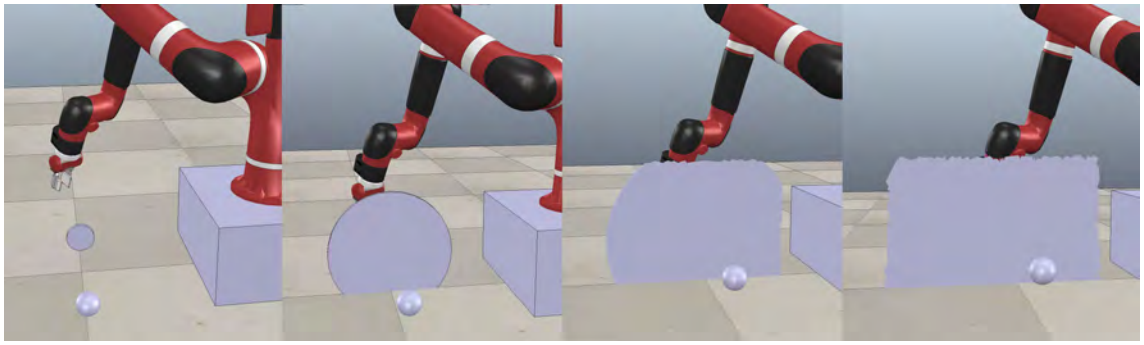

Figure 8.5: Left to right: Respondable sections of the wall under increasing respondability radii. In the two right hand images we manually extract the shapes seen from cylinders by manually selecting vertices. The right-most image shows the `end_radius`, when the wall is almost fully respondable.

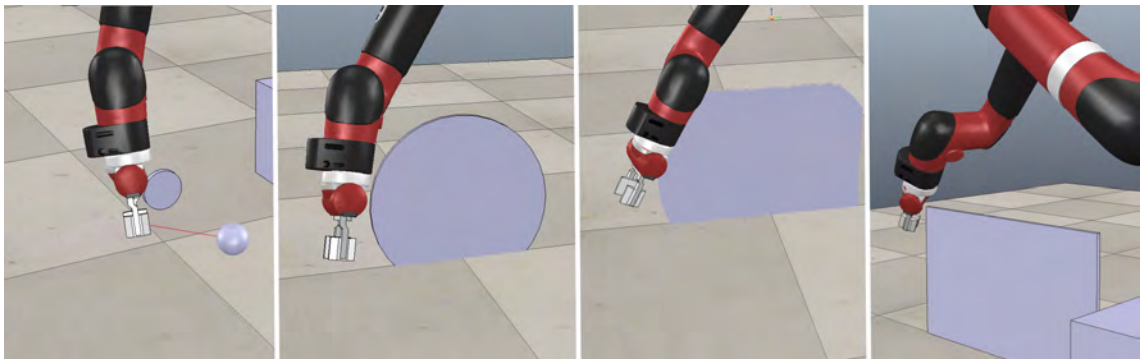

Figure 8.6: Left to right: Robot position as it crosses the wall after training at different steps in the curriculum.

proved much more complex than we anticipated. The height of the wall relative to the arm's starting configuration is the key contributor to this. The agent must learn to move the robot in a specific way that is hard to model with dense rewards. It is one of the key success of the project to have overcome our initial and repeated failures on Reach Over Wall, doing so with our new reinforcement learning method, CuRL.

8.3 Pose Estimator

While our attempts at sim-to-real transfer are limited when compared to the in-depth exploration of CuRL in simulation, we did train and compare many different hyperparameters when searching for a strong pose estimator for Dish Rack which we will detail here.

8.3.1 Training Set Size

First we wanted to make sure we were generating enough data to train our models properly. So we trained pose estimators on datasets of different sizes to examine the effect of increasing training data on performance. The results are shown in table 8.4. Our best Dish Rack policy, which we wanted a pose estimator for, uses

observations of the rack position relative to the current plate position. Therefore the pose estimators in this table were trained to output relative observations. They were trained using Adam[42] with a learning rate of 1×10^{-4} and a mean squared error loss function. All reported values are on a 256 sample validation set that was not used during training.

Samples	Min loss	Distance Error (mm)	Rotational Error (radians)
2048	2.41×10^{-4}	14.66	0.00722
4096	1.52×10^{-4}	9.27	0.00686
8192	1.03×10^{-4}	**6.82**	**0.00433**
16384	$\mathbf{8.61 \times 10^{-5}}$	8.04	0.00474

Table 8.4: Effect of training set size on performance of a relative pose estimator.

The minimum test loss is achieved with 16384 examples. However, both that pose estimator's mean distance error and mean rotational error are higher than the one trained on 8192 samples. This would suggest that though the mean error is better when 8192 samples are used for training, there are more outliers which lead to a higher squared error loss.

As we had found a disagreement between the loss function and the errors we were actually aiming to minimise, we searched for a new loss function that would be more representative of actual estimator performance.

8.3.2 Different loss functions

As mentioned in Section 7.1.1, Ren et al. [40] presented a unique loss function suitable for pose estimation. It uses the L1 (or mean absolute error) loss for the position difference and an L1 loss on the cosine orientation difference, as follows:

$$L(x, \theta) = ||x - \hat{x}|| + ||cos(\theta - \hat{\theta}) - 1|| \tag{8.1}$$

where x and $\hat{x}$ are the actual and estimated positions, and θ and $\hat{\theta}$ are the actual and estimated orientation. We compared MSE loss to this "Ren loss", a variant on it using MSE loss instead of L1 loss on the cosine orientation difference, and a further variant using MSE losses for both components of Ren loss. All pose estimators were trained with 16384 examples and the results are shown in Table 8.5 below.

Loss function	Distance Error (mm)	Rotational Error (radians)
MSE	8.04	0.00474
Ren Loss	5.47	0.00386
Ren with MSE for orientation	**5.23**	**0.00327**
Ren with MSE for both	8.03	0.00400

Table 8.5: A comparison of the effects of different loss functions on pose estimator performance.

The results clearly show that using mean absolute error instead of mean squared error leads to better positional estimation. Using our variant on Ren Loss with MSE on the cosine orientation difference provided the best overall pose estimator though this improvement is marginal.

It is worth noting that the rotational error achieved is much smaller than it need be for the plate to be placed successfully into the rack. We did weight the rotational and positional components to try and make training favour network updates that lead to improved positional estimations. This had no particular effect, indicating that there is not a noticable trade-off between positonal and rotational performance.

8.3.3 Translation to performance on Dish Rack

We tested our best pose estimator from Table 8.5 on our best policy in simulation. The policy usually achieves 94% accuracy but with a pose estimator this reduced it to 0%, much worse than expected given our error figures. Watching the policy back helped better understand what the problem was. The robot arm moves in roughly the right direction, but never lands in the right rack slot. It usually places the plate one or two slots in front.

The error in relative observations meant the policy progressively slips from paths usually followed by on the way to the second to last rack slot. As the plate leaves the path it would fully with a perfect pose estimator under this policy, we hypothesise that it enters unknown states. The plate is typically lower than usual as it moves across the rack, and it seems like given the joint angles the policy believes it should be lowering the plate into the rack.

These results make pose estimation seem completely unfeasible with this degree of error. However there is another alternative. As the joint angles are available in a real world environment, it is possible to calculate the position of the subject. Therefore if we can produce absolute pose estimations we can turn them into relative observations for our policy. Our hypothesis was that with a good estimator, absolute estimations are unlikely to change much when the robot moves. This would mean the robot can aim for an estimated target and not deviate from its path to that target very much.

8.3.4 Estimating Absolute Rack Position

We trained absolute pose estimators in much the same way as the relative ones. First we trained using the 4 different-sized datasets, with Adam, MSE loss and a learning rate of 1×10^{-4} as before. The results are as follows:

Samples	Min loss	Distance Error (mm)	Rotational Error (radians)
2048	3.25×10^{-4}	17.27	0.0244
4096	1.80×10^{-4}	12.58	0.0141
8192	1.13×10^{-4}	10.61	0.0113
16384	$\mathbf{4.96 \times 10^{-5}}$	**6.66**	**0.00760**

Table 8.6: Effect of training set size on absolute pose estimator performance.

From this point we trained using 16384 samples on the four loss functions as before:

Loss function	Distance Error (mm)	Rotational Error (radians)
MSE	6.66	**0.00760**
Ren Loss	**4.89**	0.0125
Ren with MSE for orientation	5.36	0.036
Ren with MSE for both	5.64	0.0164

Table 8.7: Exploring the effect of different loss functions on absolute pose estimator performance.

Here the standard Ren Loss gives us the best performance. An interesting difference between absolute and relative pose estimation is that the relative pose estimator in general produces a better estimation of the rotational rack position, even though both types of estimator are attempting to produce the same value for rotation. This stays absolute across both types of estimator. It is odd then, that changing what else the network aims to predict affects how well it can estimate the same observation.

8.3.5 Best Pose Estimator Performance

To begin we tested the best pose estimator trained with 16384 samples and a Ren loss function on our best full task policy in simulation. The combined estimator and policy was successful in 84% of the 50 episodes, meaning it failed 7 more episodes than when using full state information. This was a much more exciting result, and following it the next logical step was to test this policy and pose estimator in a real environment.

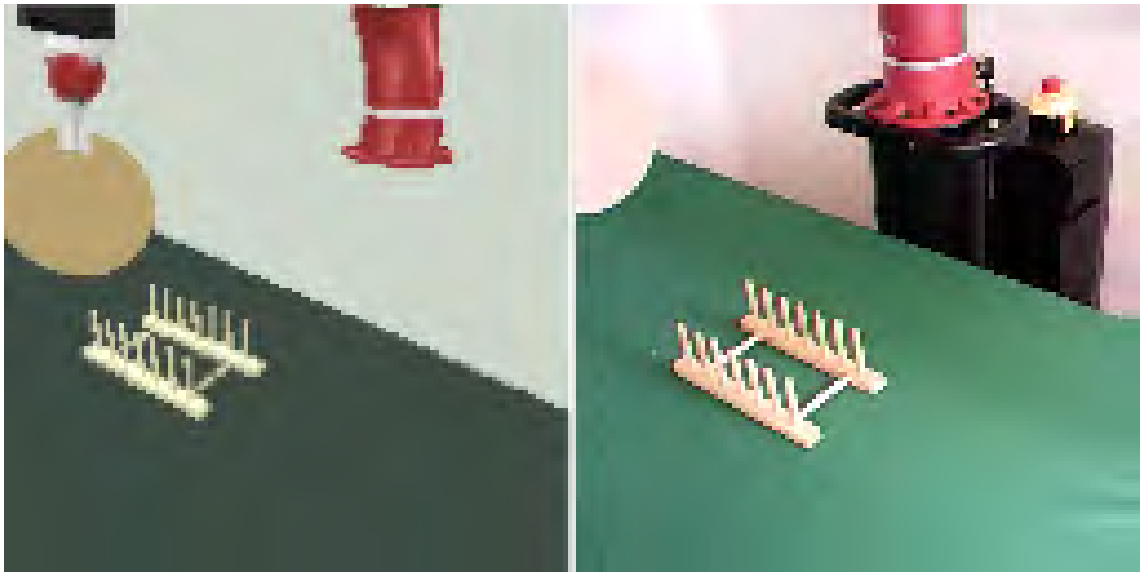

Figure 8.7: Left: A simulated image. Right: An image of the real environment.

The real environment setup can be seen in Figure 7.1. The robot sits at the same height from the table as the robot did in simulation. The plate is made from a thin block of wood to avoid causing accidental damage to a real one and a cloth covers the table as planned to increase variation between the wooden dish rack and the table. The camera is positioned so the images resemble the simulated environment as closely as possible, and a comparison photo is shown in Figure 8.7.

The model was not successful in reality, but showed promise. The estimations for y position (distance perpendicular to the robot) and rotation were accurate. The robot moves the plate to the correct y distance from it at the correct orientation. However the estimate for x position was consistently off by roughly 7 cm. This led to the plate being placed in the rack, but in the wrong slot, shown in Figure 8.8.

Figure 8.8: Frames from a video of the policy and pose estimator being used on a real robot. The plate should have been placed two slots further right in the rack.

The robot does not behave the same way in reality as it does in simulation under the same policy and the same pose estimator. This is due to the error introduced when transferring from simulation to reality. The failure is most likely a sign that we do not randomise the domain enough when generating the simulated images. The real images do not fit into the distribution of generated simulated images and as such we are asking our model to extrapolate to unknown states. We should be able to improve this performance with more randomisation.

8.3.6 Increasing Domain Randomisation

One of the key differences between the simulated and real images, compared in Figure 8.7 are the objects not modeled in the simulation. This difference could be what is causing the error in our estimation for position along the x axis.

Random Textures

We decided to randomise the background by applying textures to the back wall and the table at random from a large number of very different textures. In doing so we greatly increase the variation in background in the hopes of reducing error when crossing the reality gap. Figure 8.9 shows a sample of the new simulated images.

Unfortunately, training the pose estimator on this dataset was not as successful as on the previous, less randomised dataset. This is perhaps to be expected due to the higher variation in images, but Figure 8.10 shows that the model actually

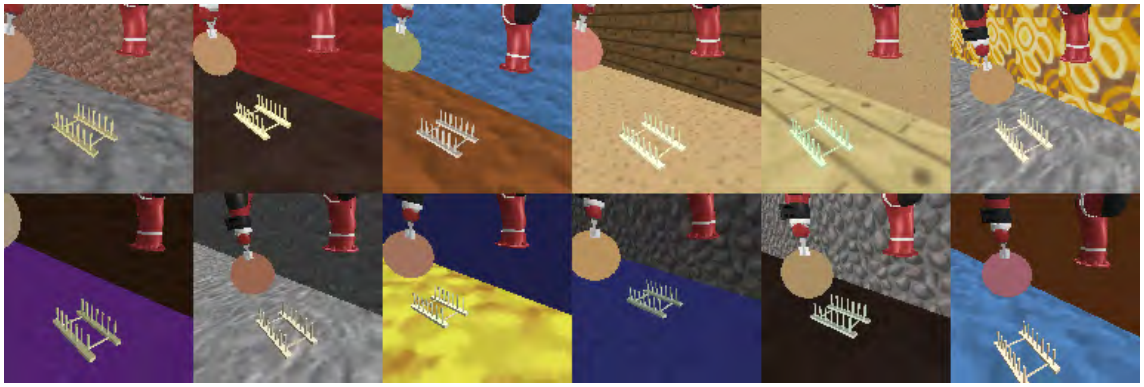

Figure 8.9: Random textures applied to the wall and cloth.

suffers from severe and immediate overfitting. On the validation set, the average distance error of the trained model is 18.4 mm and the average rotational error is 0.0316 radians.

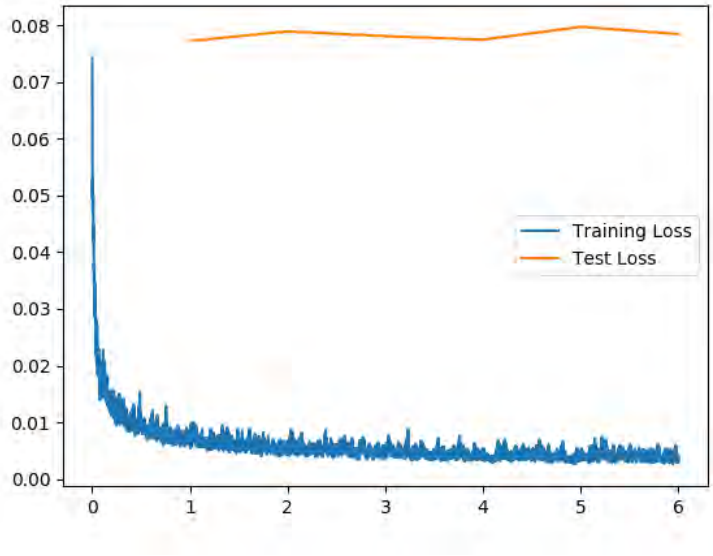

Figure 8.10: Graph of training and validation loss while training pose estimator with random textures. Number of epochs is on the x axis and loss is on the y axis. Training loss measured every mini-batch update. Validation loss measured after every epoch.

We have a hypothesis as to why the model overfits. The texture pack features 632 unique textures. They are computer drawn and brightly coloured. This will make them relatively easy to distinguish. We use two textures, one on the wall and one on the table, meaning there are almost 400,000 possible combinations. In our small dataset there are 65,000 images making it unlikely that many of the combinations of textures are ever repeated. Because of this, we believe that the pose estimator actually learns to recognise the texture combination and output the corresponding rack observation, instead of learning anything about the rack position.

Matching the real environment with greater randomisation

As an alternative to random textures, we next tried to improve the simulation environment's representation of the real one. We do this by modelling the stand, the machine that sits behind it, and the button that sits on the machine, and showing the end of the table and adding a left wall. We randomise position and rotation of the table, the machine and the button. We randomise the height of the stand, and randomise colours of everything, including the Sawyer joint angles. We generate a new dataset of 75,000 images, and show a sample of the new images in Figure 8.11.

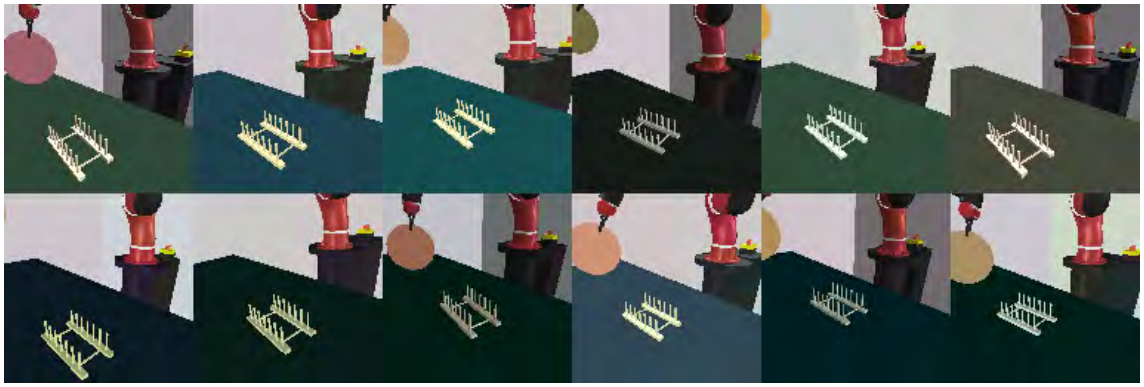

Figure 8.11: Random textures applied to the wall and cloth.

We trained an absolute pose estimator on this dataset, which trained to convergence at validation postional and rotational error of 5.68 mm and 0.0125 radians respectively. We also tried altering the architecture, using a stride of 2 in the final convolutional layer in each group instead of a MaxPool layer for dimensionality reduction. The thinking behind this is that max pooling layers often help architectures have positional invariance, meaning movement of the rack might not lead to large differences in activation, hindering the models ability to estimate position. This trained model was not as good as the one with max pooling, with mean distance error of 7.52 mm and orientation error of 0.0161 radians. Therefore we kept the max pooling layers in our architecture.

8.4 Most recent tests in reality

We tested our new absolute pose estimator and the end-to-end controller trained in Section 7.2.2 on Sawyer. The pose estimator performs slightly worse than the first pose estimator trialled earlier in Section 8.3.4. Its position estimation is slightly worse in both the x and y direction and the rotation estimation is quite inconsistent. Figure 8.12 shows some end positions of the plate when using this pose estimator. Note that these images were taken separately to those used for the pose estimation.

The end-to-end controller's performance was much more variable. The absolute pose estimations don't change much over an episode, meaning the pose estimator essentially picks a point and the plate travels there. With an end-to-end controller, the error in each step can push the agent further and further into unknown states which it will not recover on. We found that the controller was very sensitive to changes in the environment, in particular camera position.

Figure 8.12: End positions of the plate when using the new pose estimator on our strongest simulation policy.

We attempted to match the room as closely to the simulation as possible, altering camera and table poses to see what effect they had on the policy. A new comparison of simulated and real images is shown in Figure 8.13 Out of 10 different tests, in 1 the policy was actually successful. In 3 other tests, the robot comes close to the rack but does not place the plate in the correct slot, similar to in Figure 8.12. In the remaining 6 it did not move anywhere near to the rack.

Figure 8.13: Left: A simulated image from the most recent dataset. Right: An image of the real environment.

The end-to-end controller was very sensitive to changes in camera pose. It is more difficult to find a real camera pose that gives similar images to the simulated camera than we had expected. This is due to differences in zoom rate between the real camera and the simulated vision sensor, meaning we cannot measure where the camera should be placed. While the camera pose is randomised, this is kept to a small range to ensure that the base of the robot and the rack are always visible. Moving forward we are going to increase this range to randomise camera pose much more. We hope to show more progress in this area at the project presentation.

Though the end-to-end controller has very wide variation between successful and unsuccessful runs, it has succeeded once where our pose estimators have not. At the

beginning of Section 7.1.4 we also discussed the limitations current state-of-the-art pose estimation would impose on CuRL. This positions an end-to-end controller trained with supervised learning as the most suitable method for transferring CuRL policies to reality, based on the experiments completed to limited success in this project.

Moving forward we hope to train a new end-to-end controller with further increased domain randomisation and a much larger training set. Related work shows that the datasets required for successful end-to-end training are much larger than those we have been using [4]. We will also train an absolute pose estimator on this larger dataset, so we can properly evaluate the conclusion drawn in the last paragraph.

Chapter 9

Conclusion

We have successfully applied Deep Reinforcement Learning to obstacle-driven goal-oriented robot control tasks. These tasks have significant relevance as the state-of-the-art moves toward being able to produce commercially available robots for use in everyday life. In particular, putting a plate in a dish rack is a common task and a desirable feature in any household robot. We have designed and implemented a robotic environment to simulate this real world task and shown our new approach, CuRL, is able to solve it to a 98% success rate.

Beyond designing the method for solving this task, we attempt to demonstrate the policies learned in simulation in the real world. Doing so is particularly important for proving CuRL's worth when compared to the state-of-the-art for robot learning. As mentioned, we are targeting the CoRL for a paper submission and will be up against papers demonstrating strong real world performance. Achieving this with CuRL will greatly improve the chance of our paper being accepted. While we have not achieved significant success in this area what we have done is conducted a study into the suitability of a range of methods for transferring policies across this reality gap, and are continuing to make progress in this area. We hope to show more progress in this area at the project presentation.

Not only has CuRL shown success on Dish Rack task, it was also 90% successful at Reach Over Wall. This is an even more significant result as this task was unsolvable with state-of-the-art deep reinforcement learning methods from scratch. Furthermore, CuRL uses standardised and simple reward functions, making it a straightforward to define and solve new complex tasks. All that needs to be defined are step size, start and end radius, and length of training run. Most of these can be found intuitively by hand without significant fine-tuning.

Overall, the results in simulation clearly show that CuRL is a promising new method for robot learning. Due to its novelty, the project lends itself to multiple options for further study that could greatly enhance CuRL's performance and usability. We will now discuss some of these potential directions.

9.1 Future Work

Improvements to Simulation Software

As discussed, V-REP in its current form limits the degree of control we have over an object's respondability and in doing so makes CuRL quite a coarse algorithm in its current implementation. Extensions to simulation software can help improve this. In particular, we suggest being able to set the respondability of a shape at the vertex level, possibly using a mask with a flag representing respondability for each vertex. It would also be useful if, given a respondability sphere the simulator could automatically set respondability. Where necessary, the simulator could split large vertices into more smaller ones so that the error in this is minimised.

Standardising the pipeline

We have created a method capable of training a wide variety of tasks. It would be useful then, for the process of setting up a new task to be made as simple as possible. Improvements in the last section will mean that a user does not have to manually create the environments for the curriculum. Our code should also be standardised so that a user need only create a subclass defining the subject, the target, obstacles and the CuRL hyperparameters. The reward functions are standardised so these too could be abstracted away. We intend to make these changes through some minor refactors and release a Python package with them over the summer. This could act as a baselines package for future work on CuRL.

Improving the training process

Currently training is split into multiple training runs of a predefined length. This can likely be improved by finding a metric that can decide when training should advance to the next stage. Such a metric may use median reward or minimum reward over a number of previous episodes to track when the policy is effective enough with the current level of respondability to increase the size of the respondability sphere.

A good median or minimum reward may be difficult to find and could require some tuning for each individual task. In our general pipeline we would like required tuning for each new task to be minimal. In addition, the maximum possible reward will decrease over training as obstacles increase in size and avoiding them means taking longer to reach the goal. So a fixed metric may not be suitable.

Alternatively, what if increasing the respondability radius didn't only happen between training runs but instead actively during a single training run? And what if the respondability radius was sampled from a distribution? We could sample the respondability of an environment for each episode from a distribution. Mehta et al. presented Active Domain Randomisation (ADR) in their April 2019 paper [43], in which they pose domain randomisation as a reinforcement learning problem. The randomisation agent maximises its reward by finding environments which the agent can currently learn the most from.

In the context of Curriculum Reinforcement Learning, ADR would learn to show the agent environments with low respondability radii early on in training. As the agent becomes better at avoiding smaller obstacles, it will learn more from seeing environments with higher respondability radii and so ADR can learn to show these

environments more frequently. Training would continue until the policy is successful on the true, reference environment.

Perhaps future research will find that the shape of the respondability region can be optimised too. Using uniform spheres may not be optimal if we have control of vertex respondability. Perhaps growing a respondability region from vertex to vertex could find an optimal way to grow obstacles. Similar to ADR, this too could be posed as a learning task and would be an interesting setting for future study.

Bibliography

[1] Tom M Mitchell. *Machine Learning*. McGraw-Hill, Maidenhead, U.K., 1997.

[2] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods, and Algorithms: Second Edition*. John Wiley & Sons, Inc., Hoboken, NJ, USA, July 2011.

[3] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE Int. Conf. Intell. Robot. Syst.*, pages 5026–5033. IEEE, October 2012.

[4] Stephen James, Andrew J. Davison, and Edward Johns. Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task. In *Conf. Robot Learn.*, July 2017.

[5] Andrei A. Rusu, Mel Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-Real Robot Learning from Pixels with Progressive Nets. *Conf. Robot Learn.*, October 2017.

[6] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-Real: Learning Agile Locomotion For Quadruped Robots. *Robot. Sci. Syst.*, April 2018.

[7] Michael Laskey, Chris Powers, Ruta Joshi, Arshan Poursohi, and Ken Goldberg. Learning Robust Bed Making using Deep Imitation Learning with DART. *Conf. Robot Learn.*, November 2017.

[8] Rouhollah Rahmatizadeh, Pooya Abolghasemi, Ladislau Bölöni, and Sergey Levine. Vision-Based Multi-Task Manipulation for Inexpensive Robots Using End-To-End Learning from Demonstration. *IEEE Int. Conf. Robot. Autom.*, July 2018.

[9] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End Training of Deep Visuomotor Policies. *J. Mach. Learn. Res.*, 17(39):1–40, April 2016.

[10] Lerrel Pinto and Abhinav Gupta. Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours. In *Proc. - IEEE Int. Conf. Robot. Autom.*, volume 2016-June, pages 3406–3413, September 2016.

[11] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *Int. J. Rob. Res.*, 37(4-5):421–436, March 2018.

[12] 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. December 2018.

[13] Rethink Robotics. Sawyer Collaborative Robots for Industrial Automation.

[14] Sanmit Narvekar. Curriculum learning in reinforcement learning. *IJCAI Int. Jt. Conf. Artif. Intell.*, pages 5195–5196, 2017.

[15] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. \url{https://github.com/openai/baselines}, 2017.

[16] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual Reinforcement Learning for Robot Control. December 2018.

[17] Tom Silver, Kelsey Allen, Josh Tenenbaum, and Leslie Kaelbling. Residual Policy Learning. December 2018.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[19] Jason Bell. *Machine Learning: Hands-On for Developers and Technical Professionals*. Wiley, 2014.

[20] Kaggle. Dogs vs. Cats, 2013.

[21] Richard Sutton and Andrew Barto. *Reinforcement Learning: an Introduction.* MIT Press, Cambridge, MA, 2018.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vis.*, 115(3):211–252, December 2015.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 770–778, 2016.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, December 2013.

[25] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science (80-.).*, 362(6419):1140–1144, December 2018.

[26] Ethan Knight and Osher Lerner. Natural Gradient Deep Q-learning. 2018.

[27] Yan Duan, Marcin Andrychowicz, Bradly C. Stadie, Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-Shot Imitation Learning. In *Adv. Neural Inf. Process. Syst. 30*, pages 1087–1098. Curran Associates, Inc., March 2017.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. July 2017.

[29] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. In *AAAI Conf. Artif. Intell.*, September 2018.

[30] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *Proc. Mach. Learn. Res.*, 37:1889–1897, February 2015.

[31] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Int. Conf. Mach. Learn.*, February 2016.

[32] Pieter-tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A Tutorial on the Cross-Entropy Method. *Ann. Oper. Res.*, 134(1):19–67, 2005.

[33] Erwin Coumans. Bullet physics engine, 2010.

[34] Reza Mahjourian, Risto Miikkulainen, Nevena Lazic, Sergey Levine, and Navdeep Jaitly. Hierarchical Policy Design for Sample-Efficient Learning of Robot Table Tennis Through Self-Play. pages 1–100, 2018.

[35] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. TossingBot: Learning to Throw Arbitrary Objects with Residual Physics. March 2019.

[36] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Int. Conf. Artif. Intell. Stat.*, pages 627–635, June 2011.

[37] Ilya Kostrikov. PyTorch Implementations of Reinforcement Learning Algorithms. \url{https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail}, 2018.

[38] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. pages 1–4, 2016.

[39] Eric Rohmer, Surya P N Singh, and Marc Freese. V-REP: a Versatile and Scalable Robot Simulation Framework. In *Proc. Int. Conf. Intell. Robot. Syst.* 2013.

[40] Xinyi Ren, Jianlan Luo, Eugen Solowjow, Juan Aparicio Ojea, Abhishek Gupta, Aviv Tamar, and Pieter Abbeel. Domain Randomization for Active Pose Estimation. 2019.

[41] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. September 2014.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. December 2014.

[43] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active Domain Randomization. April 2019.