Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Learning end-to-end robotic manipulation of deformable objects

Supervisor:
Andrew Davidson

Author:
Jan Matas

Second Marker:
Edward Johns

PHD Supervisor:
Stephen James

June 17, 2018

Abstract

Manipulation of rigid objects has been consistently researched in the last couple of decades which resulted in multiple end-to-end learning algorithms capable of inferring the joint velocities directly from RGB observations. However, to the best of our knowledge, no one has tried to learn end-to-end visuomotor policy for manipulating deformable objects and transfer it directly to the real world. We have used a combination of state-of-the-art deep RL algorithms to address the problem of manipulating cloth, and we evaluate our approach on three tasks - folding a towel up to a mark, folding a piece of cloth diagonally and draping a face towel over a hanger. Training of the agent was seeded with a small number of demonstrations collected in a simulator and executed entirely in the simulation with strong domain randomisation. We then show a successful transfer of the learned policy to the real robot, without ever seeing any real-world data.

Acknowledgements

I would like to thank:

- **Andrew Davidson** for his invaluable advice and encouragement with regards to all aspects of this project.
- **Stephen James** for his expertise in the field of RL for robotics and his willingness to share it, for giving me invaluable advice when I got stuck and for being always there for discussion of any problems I have encountered.
- **Edward Johns** for his help with redacting the final CoRL publication and many discussions of the project progress.
- Every member of **Dyson lab** for warmly welcoming me in their workspace and sharing the compute resources without which this project would not be possible.
- My mother, **Sona** and my girlfriend, **Marina** for their love and support.

Contents

Chapter 1

Introduction

The majority of state-of-the-art work in robotic manipulation focuses on working with rigid objects that either do not deform when they are grasped or have negligible deformation. However, deformable object manipulation has many important real-world applications. Key domains of interest are home assistance robotics (cloth folding [1], bed making [2], getting dressed [3, 4]); medicine (robot surgery [5], suturing [6]); and industry (cable insertion [7]). Robots attempting to work with these objects are however presented with many new challenges, most notably the large configuration space the object can be in, the difficulty of accurate modelling of object behaviour, and the large change in the configuration resulting from manipulation attempts. All these factors contribute to the difficulty of deformable object manipulation. The conventional approach to manipulate an object in robotics would consist of a series of small tasks [8]. For example, consider a robot that is supposed to put a cube into a basket. It would find the cube; decide where to grasp it; plan the path to reach grasp point; grasp the cube and verify grasp success; locate the basket and plan the trajectory to reach above it and only then it would execute the trajectory and finally release the cube. Each task is programmed separately, often using very error-prone open-loop approach. Moreover, the errors propagate from one step to another (e.g. inadequate grasp point results in poor or unsuccessful grasp). This problem becomes even more pronounced when dealing with deformable objects. Even very small inaccuracy of robot movement can cause the object to deform in an unpredictable way, so the robot needs some servoing method to adapt to changes happening during the manipulation. It is, therefore, necessary to look for different approaches if we want to enable the robots to handle deformable objects confidently.

Instead of using this pipeline of steps, new research suggests using machine learning to teach the robot how to perform actions based purely on raw sensor data, such as camera images or joint angles. This technique is particularly interesting as it would allow the next generation of robots to learn to accomplish new full tasks directly without hand-engineering steps to perform each sub-action. In other words, the robot would be able to look at the data from sensors, such as RGB camera images and directly infer the motor velocities using a single, task agnostic learning algorithm.

One specific branch of machine learning applicable to the task of robotic manipulation is Deep Reinforcement learning. RL, in general, builds on behavioural psychology and investigates how to teach an agent to execute actions that lead to the highest cumulative reward. In practice, this would mean that the engineer would only need to define what is the goal of the robot (and hence when it gets the reward) and the robot would learn how to achieve it by itself. This is a massive decrease in the workload compared to a conventional approach where the engineers need to design and implement each subtask necessary to achieve the goal. Moreover, a robot learning to achieve the task is likely to make mistakes, so it needs to learn how to recover from them. In contrast, the conventional robot would either need to be hard-coded to cope with each possible mistake it can make, or it would fail to accomplish the goal. Deep RL uses deep neural networks to learn the right robot policies, and it has found applications in many other robotic domains, including rigid object manipulation [9, 10], UAV control [11] or bipedal robot control [12].

Unfortunately, Deep RL is known to be extremely sample inefficient - it requires vast amounts of training data to achieve good performance. While it is possible to collect the real-world data using robot farms, it is extremely time consuming and prohibitively expensive for most research teams. An alternative is to train robots in simulation, where one can cheaply create thousands of manipulators executing tasks in parallel. Moreover, optimised simulation runs significantly faster than a real-time robot, which further speeds up the learning process.

However, it has been shown that transferring the models learned in simulation to the real world (crossing the reality gap) is sometimes challenging and the robots perform poorly when they need to act in an unseen domain. There are, however, multiple emerging techniques to cope with the challenges imposed by domain transfer, and recent state-of-the-art research shows that it is possible to train the robot exclusively in simulation and apply the trained policy in real-world without further training. [13, 14, 15]

We hypothesise that applying deep reinforcement learning on deformable object manipulation can result in a system which is:

- robust to the new challenges arising from object deformation
- not dependent on task-specific engineering
- trained without the use of real hardware in simulation and usable in the real world

This work combines the state-of-the-art research in three different related domains: deformable object simulation, deep reinforcement learning and transferring trained machine learning models over the reality gap. To the best of our knowledge, this is the first time Deep RL is applied to the task of deformable object manipulation with a sim-to-real transfer.

1.1 Objectives

Our goal was to investigate whether it is possible to use deep RL to learn to manipulate deformable objects in end-to-end fashion (from RGB images to robot control signal). As we mentioned in the previous section, the task we want to achieve combines the work in 3 different domains, and we can set the objectives accordingly:

1. **Simulation**
    - Research available physics engines and select a simulator which has adequate support for robotic simulation as well as deformable object simulation.
    - If necessary, build upon the simulator to introduce new necessary features and improve the stability/believability of the simulation.
    - Define a set of manipulation tasks that can be reasonably attempted by the robot both in simulation and in the real world. Take inspiration from existing literature on deformable object manipulation.
    - Incorporate the deformable object simulation into well-defined simulated environments where the robot can learn to accomplish various manipulation tasks.
2. **Deep Reinforcement learning**
    - Research and evaluate various Deep Reinforcement Learning algorithms and choose the best candidate for the project.
    - Research literature and implement the latest extensions to the algorithm, so it learns to accomplish the manipulation tasks in the simulation.
    - Evaluate the final model through an extensive set of ablations to understand the behaviour of specific features and inform further research in the domain.
3. **Domain transfer**
    - Implement domain randomisation [14, 13, 16] in the simulation and train new models robust to domain changes.
    - Implement the "transfer environment" that allows the model to control real robotic arm seamlessly.
    - Evaluate the transfer on the manipulation tasks in the real world.

1.2 Challenges

Overall, the project was challenging because it was dealing with state-of-the-art research and hence it entailed many concepts that are yet to be fully understood. Moreover, it ventured into initially unexpected areas such as: debugging huge codebases, OpenGL rendering or Python language internals. The most significant challenges we had to deal with were:

1. **No stable and supported soft body simulator** - even though we have found a simulator which supports deformable objects [17], the source code

maintainers warned us on many occasions that the functionality is experimental and not yet supported. We had to implement various new features including object rendering or anchoring, extend and debug Python extension code, hunt down memory leaks and solve intermittent segmentation faults in a massive codebase.

2. **No standard benchmark tasks for deformable object manipulation** - at the time of writing, there were no standard tasks or environments where the researchers can test their algorithms for deformable objects manipulation. We had to do extensive research to find a set of feasible tasks and implement own environments to evaluate the agent.

3. **Large variance of results** - Deep RL is notorious for producing hardly reproducible results because the environments are stochastic, the algorithms are stochastic, and the execution heavily depends on the element of luck - does the robot randomly discover the rewarded behaviour? We often had problems understanding what caused a performance decrease between two subsequent training runs. Was it a code change we did, a hyper-parameter we changed or pure luck?

4. **Immense training times** - Deep RL algorithms take very long time to yield representative results (at least 24 hours for experiments learning from RGB pictures, sometimes even more). This makes the iteration extremely slow, and it makes it hard to prioritise which experiments are worth the time spent executing them. Moreover, there is also a problem of experiment granularity - ideally, we would change one hyper-parameter at a time and test it with multiple random seeds to eliminate random effects. In practice, we often change multiple things between runs to save time, but it then becomes difficult to attribute the increase/decrease in performance to a specific change.

5. **Experiment failures** - we have encountered various issues that sometimes caused an experiment to end prematurely, such as machine running out of disk space or memory, machine restarts or segmentation fault in the library. Those issues were inconvenient and they slowed down the progress.

6. **Real robot** - we had both software and hardware issues with the real robot. From the hardware point of view, the gripper we used was damaged by previous experimentation, and it often caused the cloth to fall from it. We also experimented with new gripper parts later during the project but we were constantly worried about breaking them. The provided software SDK [18] was based on Python2.7 which caused some incompatibility with our codebase and the joint controls were not working reliably (gripper commands were sometimes unfinished or ignored). We had to do significant engineering work to create a working infrastructure to test the policy on the real robot.

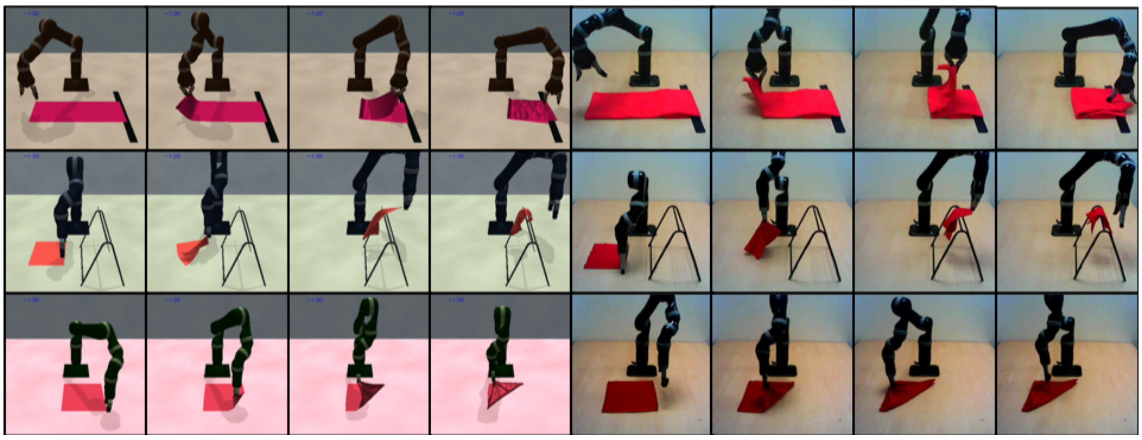


Figure 1.1: The project presents agent trained solely in simulation environments (left) and tested in real-world (right). We evaluate the algorithm on three different tasks: folding a large towel up to tape (top row), hanging a small towel on a hanger(middle row) and diagonally folding a square piece of cloth (bottom row).

1.3 Contributions

After extensive research of various deformable object manipulation tasks, we have decided to focus on cloth manipulation within this project. It would not be feasible to research manipulation of a wide variety of deformable objects due to time constraints and scope of this project. However, we believe that cloth is one of the most challenging objects to manipulate due to its strong de-formability and difficulty of accurate simulation.

To the best of our knowledge, this work is the first ever application of sim-to-real Deep Reinforcement learning to cloth manipulation. The main contributions can be summarised as follows:

1. **Reasonably realistic simulation of cloth grasping and manipulation** - we have extended Pybullet simulator with new features, such as strong cloth anchors or fake grasps to achieve reasonably accurate simulation of cloth behaviour when interacting with a robotic arm.

2. **Set of robotic environments for learning from RGB pixels** - we have created a set of environments for RL where the robot is receiving only pixel observations. There are two simple 2D environments inspired by OpenAI gym [19], a reimplementation of rigid object robotic manipulation environments inspired by OpenAI robotics gym [20] and finally, **3 new cloth manipulation environments**. The cloth manipulation environments embody 3 different manipulation tasks for the robot - folding a face towel diagonally (Folder environment), folding a large towel up to tape (Tape environment - inspired by [21]) and draping a towel over a small hanger (Hanger environment). The agent can act in those environments and learn the tasks for which it is getting a reward.

3. **Learning algorithm** - we have taken an implementation of DDPG [22] learning algorithm and implemented more 9 improvements presented in recent re-

search. Moreover, we sometimes further customised the implementation to improve the results (e.g. demo priority computation).

4. **Domain transfer** - we implement domain randomisation to aid the domain transfer, and we show that the policies trained in simulation using Deep RL can be directly transferred to the real world.

5. **Ablation studies** - we perform a broad set of ablation studies where we remove some features of the learning algorithm. This helps us understand which code changes we implemented are indeed useful when learning the manipulation tasks.

1.4 Publication

The work presented in this report was also summarised in a paper that was submitted to Conference on Robot Learning (CoRL) 2018. CoRL annually brings together hundreds of best researchers working on the intersection of robotics and machine learning.

1.5 Report Layout

Chapter 2 introduces the basic concepts in Machine Learning and Reinforcement Learning that are relevant for the rest of the report. The following chapter focuses mainly on the research in deformable object manipulation, and it has two aims - firstly to summarise what different approaches have been employed before and secondly to inform our decisions when designing the set of tasks we are going to use for testing our algorithm. The final parts of this chapter also introduce the fundamental algorithms used in Deep RL.

The core part of the report starts with Chapter 4, where we describe our contributions towards the Pybullet simulator and also the design process of the new RL environments. Afterwards, in Chapter 5, we evaluate a selection of learning algorithms to choose the best one for this project and follow-up with description and implementation of state-of-the-art improvements to the selected candidate. This ultimately allows us to train successful policies in the simulation. Chapter 6 discusses how we achieved the transfer of the trained policies into the real world. In Chapter 7, we critically evaluate our work on simulation, test the learning algorithm and assess the quality of domain transfer. Finally, in Chapter 8, we summarise the work we have done to achieve our results, and we follow up with a couple of suggestions for future work.

Chapter 2

Preliminaries

This chapter aims to introduce the fundamentals of Machine Learning and Reinforcement learning which will be relevant throughout the project.

2.1 Artificial Neural networks

Artificial neural networks (ANNs) are biologically inspired computational models capable of solving many complex tasks by observing a significant amount of example data. They are build up from simple units, called neurons, that are interconnected to form a directed graph. Each neuron continuously reads values from its inputs which might be either the outputs of the previous neurons or they might be the inputs to the network as a whole. The inputs have a weight assigned to them. Therefore, some inputs are very significant, and some are almost ignored. After reading the inputs, the neuron computes their weighted sum and applies an activation function that further transforms the output which is then either read by another neuron or the network outputs it.

We could divide all neural networks to two categories:

- **Feed-forward networks** - these networks form an acyclic graph, so a neurons output does not influence any of its inputs. The network is sending the information forward. Feed-forward networks are used nowadays in deep architectures, where the neurons are arranged in multiple layers to approximate extremely complex functions.
- **Recurrent networks** - these networks contain loops, so the output of one neuron can become the input of another. Networks of this kind are much more challenging to train, but they are better suited to represent temporal dependencies. We did not find it necessary to employ them in this work.

Training feed-forward networks is accomplished via backpropagation. The network is trying to approximate some function f such that $f(x) = y$ where x is the vector of inputs and y is the vector of desired outputs. In the ideal scenario, the network output (as read from the output neurons) $\widetilde{y}$ would be equal to y, but this is hard to achieve. We assign some cost function to measure the dissimilarity between y and $\widetilde{y}$, and we attempt to minimise it by changing the weights. If we shift any weight in the network a tiny bit to one direction, it could either increase and decrease the cost. We exploit this property to adjust the weights to the right directions by differentiating the cost function with respect to each weight and adjusting it by some negative multiple of the gradient. This algorithm is called gradient descent. Moreover, differentiation does not become harder as we move backwards through the network, because gradient with respect to weights of input at each node depends only on the output of that node.

Gradient descent can be implemented in multiple ways. The classic approach would be to average the gradient with respect to each weight over all available examples (this is also called batch gradient descend). Although this approach arrives at a correct result, it is very time consuming, offers only very slow convergence and because of its deterministic nature, it is likely to fall into local minima. An alternative is to use stochastic gradient descent, where we always present the backpropagation algorithm with only a single example at a time. Even though the updates might oscillate, on average they will tend to the right direction. A combination of the two methods is called mini-batch gradient descent, where we use a small subset of all examples at each training step. Increasing the subset size might cause convergence in fewer timesteps, as it prevents the oscillations caused by single updates but on the other hand, it makes the descend more likely to get stuck in local minima. It turns out that mini-batch size is one of the hyper-parameters that need to be optimised for each task.

A widely used neural network architecture is a convolutional neural network (CNN). This architecture has been applied to a large number of vision tasks with large successes compared to the previous state-of-art. Krizhevsky et al. [23] used a CNN to classify 1.2 million images into more than 1000 categories with considerably better error rate than any previous state-of-art solution.

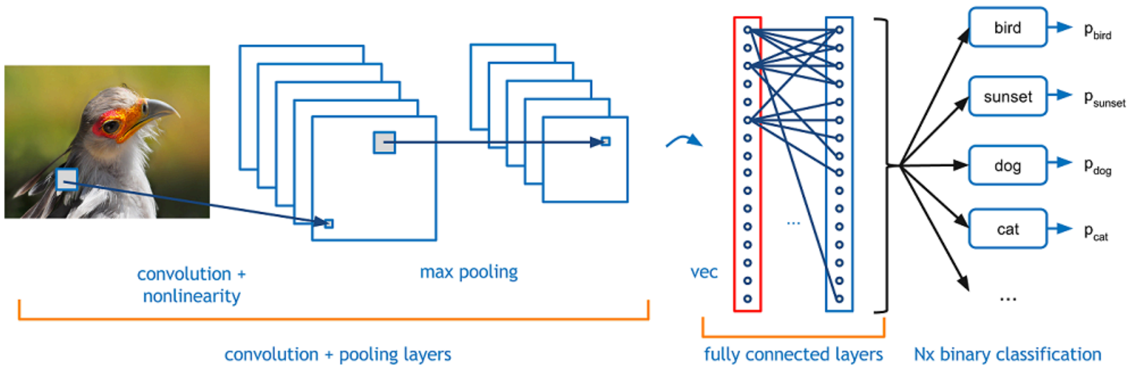


Figure 2.1: Illustration of CNN architecture. [24]

CNNs can consider spatial correspondences between pixels naturally. Similarly to feed-forward neural network, CNNs are composed of multiple layers forming a deep structure. However, it is not the case that each neuron on the first layer would be connected to each neuron in the second layer, as this would be computationally very expensive (e.g. a 100x100 image with a second layer of size 10x10 would require 1 million weights). Instead, only a small spatially-localised number of neurons in the first layer (they are together called a "receptive field") is connected to a single neuron in the second layer. The receptive fields of hidden neurons overlap so that each input pixel might be an input to multiple hidden neurons. The weights of the receptive fields are the same across all hidden neurons, and we call them a kernel. A number of kernels applied at each layer is a hyper-parameter of the network, and it decides the depth of next layer. The output of convolutional layer is often called a feature map, because the kernels sliding over the inputs detect features in images and if a specific neuron in output fires, we know that the corresponding feature has been seen in the image.

The convolutional layers might be followed by pooling layers that reduce the size of the resulting output, usually by taking the maximal output or by taking the average. Pooling layer still saves the spatial correspondence in the feature map, as it again only pools the outputs of collocated neurons in previous layer.

The output of a couple of alternating convolutional and pooling layers is then fed into a fully connected layer that understands the correspondences between high-level features recovered in feature maps and actual classes of objects. In classification tasks, the signal from last fully connected layer passes through a softmax function to create a probability distribution, where the output of each neuron roughly corresponds to the probability that an object of the specific class is present in the picture.

2.1.1 Summary

We will extensively use both fully-connected neural networks and convolutional neural networks when building the models for our Deep RL algorithms. While fully connected networks work well for low dimensional data, CNNs can be applied directly to RGB images. We will also be combining them to create models that take both RGB images and low-dimensional vectors to predict the agent policies. We will not be using a straightforward gradient descent as described in this section but a more advanced algorithm based on the same principles - Adam [25].

2.2 Reinforcement Learning

2.2.1 Introduction to Reinforcement Learning

Reinforcement learning is a machine learning paradigm derived from behavioural psychology. At its core lives an agent, which in our case is a robotic manipulator. The agent interacts with an environment and tries to accomplish a task. There are another four basic concepts:

- **State** - at each point in time t, the state s describes the current situation. It should encompass both the information about the robot (the joint angles, the software memory, gripper status) and the environment (position of objects around the robot, translation between the base of the robot and world frame, configurations of the objects etc.).

- **Action** - action a describes the different moves the agent could take in a given state. For example, it could move one of its joints by one degree in a clockwise direction. The role of the agent is to decide what is the best action to take at each time.

- **Reward** - when an agent advances to the next state as a result of the action it performed, it receives a reward r. This reward is most often a single scalar that can be either positive or negative.

- **Observation** - in most cases, the state of the environment is not fully observable, and agent only receives some partial observation o. This observation can be either low-dimensional (e.g. a 3D coordinates of the cube an object tried to grasp) or high-dimensional (e.g. an image of a table with a cube and robot).

The agent always tries to maximise the reward it receives over its whole running time. The runtime is possibly infinite, so if the robot could achieve some small reward at each timestep, it could just keep stepping back and forth to accumulate infinite expected reward. In order to guarantee convergence of expected reward, we can introduce a discounting factor $0 \leq \gamma < 1$. The resulting equation for reward becomes:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \gamma^4 r_{t+4} + \gamma^5 r_{t+5}...$$

As a result, rewards far in the future account for less than the rewards in the next few time-steps. Moreover, as t tends to infinity, r_t tends to 0. As it is an example of geometric series with ratio between elements $\gamma < 1$ we know that the sum converges.

In RL, **an episode** is one run of the agent interacting with the environment. It starts in some initial (likely stochastic) state and it finishes either when the agent accomplishes the task or when a predefined time limit expires.

It is not always the case that the result of agents action is entirely deterministic. In most real-world scenarios, the reward for taking an action is stochastic, and we can represent it by a probability distribution. Moreover, even the state where the agent ends after the action is often stochastic. If this probability $p(s_{t+1}, r_{t+1}|s_t, a_t)$ is independent of previous states and actions (it ignores the history), we say that the process has Markov property and can be called Markov Decision Process (MDP). A majority of RL algorithms assume that they are dealing with an MDP and it is, therefore, necessary to have this property. We can achieve it by encoding all relevant historical information into the state. For example, assume we have a robot that tries to put a cube in a basket, so it needs to know if it has already grasped the cube. Instead of looking at previous states, we can add another field *isGrasping* to the state vector, and hence we can have the information without breaking the Markov Property.

Reinforcement learning aims to learn the optimal policy π (mapping from observation to an optimal action to perform in that state) for the agent to follow. In order to decide which policy is optimal, we need to define value functions for state $V^\pi(s)$ and for action $Q^\pi(s, a)$. Those functions give us the expected cumulative reward of following the policy from state s or of taking an action a in state s respectively. Intuitively, the optimal policy would be the one that assigns highest value function to initial state s (we can call this policy objective function J).

One way of finding optimal policy is value iteration and policy improvement. We can initialise π randomly, and we then evaluate policy by computing the value $V^{\pi}(s)$ of each state. Then we can improve the policy by greedily selecting the action that brings the agent to the state with the best value (also taking into account transition rewards). We evaluate the new policy and continue until the algorithm converges.

This approach would require us to have complete knowledge of the environment and consider each state in a possibly enormous state-space in each iteration. An alternative is to follow the current policy and remember the return associated with following it fully. If a state is visited multiple times, the algorithm takes the average of the rewards. This approach has en exploration/exploitation trade-off - if we set the policy to always follow the route promising highest rewards, we might never discover other even better options. One solution, called $\epsilon - greedy$ method, is to follow the policy in most cases, but take a random action with a small probability ϵ, which will allow the agent to explore new states.

2.2.2 Q-Learning

Instead of taking into account the rewards of full episodes, we can update our understanding of the environment after each transition. This approach is called Q-learning. The Q-function is always updated according to

$$Q_{new}(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r + \gamma max_a Q(s_{t+1}, a))$$

where α is the learning rate. This function is known as the Bellman equation. Practically, we would have a table storing current Q estimate for each state-action pair and we would update it whenever a transition with that particular pair is executed.

Even tough Q-learning can perform fairly well on discrete low-dimensional state space, using it in high dimensional or continuous state spaces would require some discretization schema and extremely large state/action table. An alternative is to use a function approximator to approximate the Q-value of a given state-action pair. This approach allows the model to also predict the Q-values in states it has never seen before (generalisation). For example, if we used a simple linear combination of state features as function approximator for Q-value, the predicted Q-value for similar states would be likely very similar (unless the coefficients for some features are extremely high). However, a linear combination of the features is rarely expressive enough to capture the true Q-value of a state and it results in poor approximation for complex tasks. Some Deep RL methods solve this problem by using deep neural networks to approximate the Q-function and sometimes they even approximate the policy with deep neural network to avoid the need for searching action with maximal Q in each state. We will look at those methods further in section 3.4.

One last thing we would like to introduce is the classification of RL algorithms. Firstly, we can divide them into model-based and model-free. **Model-based** RL algorithms attempt to learn the transition probability distribution $T(s_{t+1}|s_t, a_t)$. In other words, they explicitly model what is the most likely state s_{t+1} that results from execution of action a_t in state s_t. On the other hand, **model-free** do not explicitly model the behaviour but they simply rely on experience. For example, let's assume a student wants to get home from college quickly at 5 pm and can either cycle or take a bus. A model-based behaviour would be to plan both routes and use a previously learned model of traffic dynamics to estimate the delays caused by congestion and choose the faster option. The model-free system would say that at 5 pm, the buses are always slow so the student should cycle. In general, model-based approaches do not scale well to high-dimensional state spaces, so we will only consider model-free algorithms.

An RL algorithm can also be off-policy and on-policy. **On-policy** algorithms are learning only from the experiences generated by their current policy. In other words, they are estimating the value of actions assuming that the current policy will be followed. **Off-policy** algorithms can learn from any experiences in the environments and it does not matter which policy generated them. The agent could even behave randomly and still bring in valuable data. We will be mostly focusing on off-policy algorithms because they can be easily extended with demonstrations.

2.2.3 Summary

RL is a machine learning method where an agent interacts with an environment and tries to maximise the reward it receives over time. It is straightforward to see how this paradigm can be applied to the manipulation tasks. For example, a robot controlled by software agent can attempt to fold a small towel and receive a reward if it produces an adequate fold with nicely aligned corners. We will be doing precisely this as a part of the project.

More specifically, the agent will control a robotic arm on a table via an action a representing the velocity of the gripper and the velocity of the fingers. It will receive observations of the environment in the form of RGB images, gripper position and joint angles o and the environment will also have a high-dimensional state s containing the information about robot configuration as well as cloth configuration. It will be receiving reward r for folding the cloth successfully. The agent will use a deep neural network to approximate both Q-function Q and policy function π. The exact implementation of this will be described in Chapter 5.

Chapter 3

Background

The following section aims to survey current research of deformable object manipulation and also give a brief historical context to more recent work in the area. We will initially focus on classic methods of manipulating deformable objects and outline what sort of tasks have been already accomplished by the researchers. This review should reveal what we can hope to achieve in the scope of this project. We will then continue by looking at the literature on using machine learning to manipulate both rigid and deformable bodies to illustrate the variety of approaches we could use.

3.1 Manipulating Deformable Linear Objects

The simplest deformable objects are one dimensional, commonly called deformable linear objects (DLOs). Some examples of DLOs are cables, ropes or threads. Manipulating them has a wide variety of applications in manufacturing (e.g. positioning and securing cables on assembly lines), robotic assistance (tidying up rooms) or medicine (sutures). Very early work with DLOs was done by Inaba and Inoue [26], who created a small robotic arm coupled with a stereo camera that was able to manipulate a piece of rope. The system did not take advantage of physical modelling of the rope, but it instead relied solely on visual feedback. The robot could accomplish simple tasks such as putting the rope through a small circular opening. Another early work was presented by Remde et al. [27], who showed that grasping of DLO can be accomplished with only 2 pixels of visual information (obtained by light barriers). Their robotic arm assumed that a DLO is hanging from a ceiling somewhere in front of it. It first scanned the area horizontally to locate the object with a forward facing light barrier, and it later moved the gripper forwards until it sensed the DLO between its fingers. The team was able to achieve 100% grasp success rate if moderate scan speeds were used.

Saha and Isto [28] presented a dual manipulator robot that was able to tie knots using needles. The robot employed a motion planner taking advantage of a mathematical model of a DLO, as opposed to only sensing information used by previous research. The code was initially tested in simulation environment where authors tweaked the rope parameters to behave as it would in the real world visually. The robot was controlled by a planning algorithm that created a probability road-map of different configurations with a rope model determining if the configuration transition

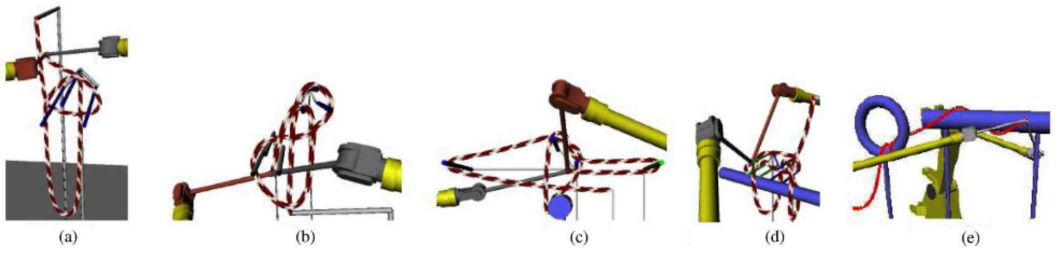


Figure 3.1: Knots executed by dual-arm manipulator as described by Saha and Itso[28].

is feasible. Hence, the roadmap essentially encoded some new rope configurations that can be achieved from the current state as well as how to achieve them. The researchers made sure that the roadmap formation is biased towards the goal state (knot tied) so the system does not need to explore large portions of the large state-space unnecessarily. After the system discovers the route on the map leading from current state to the goal by employing only feasible transitions, it can execute the motion. The robot was able to tie 5 simple kinds of knots (4 only using the DLO and 1 with also using a rigid object) as seen in Figure 3.1. Even after they introduced large Gaussian errors to the physical characteristics of the rope in the model, the robot was still able to complete the knot in a majority of the cases suggesting good generalizability of the approach.

Most systems use two robotic manipulators to tie knots. However, Wakamatsu et al. [29] created a system which is using only one simple robotic manipulator (3 translational degrees of freedom (DOFs) and 1 Rotational) to tie simple knots. Their paper introduced a topological description of configurations of DLOs, and it then categorised all transitions between states into four basic operations. Planning algorithm was used to create a chain of those operations to accomplish any tying task given only the start and final configuration.

The task of discovering which sequence of moves (plan) leads to the desired configuration can be simplified by providing a demonstration. Kahl and Henrich [30] demonstrated a robot that was instructed in virtual reality how to accomplish a DLO manipulation task and then it was able to repeat it. The robot did not learn exact coordinates of the path the object should follow, but instead it only remembered contact states (e.g. first go forwards until the object hits a specific face of an object in front of it, then go sideways until there is no contact - find an opening, then go forwards through opening). This approach, however, had difficulties with modelling complex objects and handling unstable contacts. A similar method of learning from human demonstrations was used by Hirana et al. [31] who employed special hardware to collect observation of human performing the task of hose insertion and then made a robotic manipulator repeat it.

All of the above research only considered the deformations in the static state. The robots were slow enough so all vibrations of the objects died away before the next manipulation step was considered. This approach, however, does not scale well to industrial applications where robots need to be able to accomplish tasks quickly to avoid creating bottlenecks in manufacturing process. Manipulating DLOs which are vibrating presents a new set of issues to do with the evolution of configuration in time. Yue and Henrich [32] proposed a robot that inserts a DLO (steel ruler) which is vibrating quickly into a hole. The experimental setup can be seen in Figure 3.2. The robot first samples the vibration using force/torque sensor and computes the moment when the DLO will be in a position aligned with the hole. It then executes the forward motion of the arm to insert the DLO into the hole at the right time.

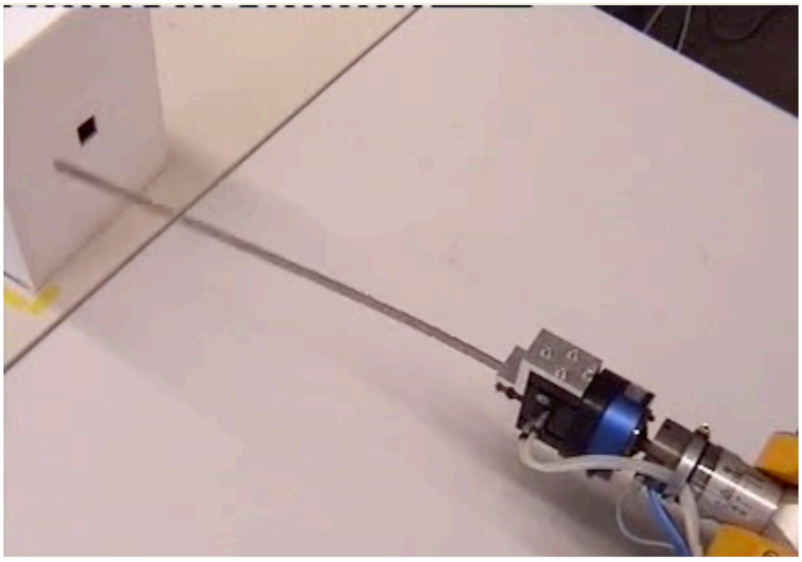


Figure 3.2: Experimental set-up for insertion of vibrating DLO to a hole [32].

Overall, the researchers handling the problem of DLO manipulation worked on the following set of tasks: grasping the DLO, inserting it into a hole, putting it into a specific location, throwing it on or tying it around a fixed rigid object and tying a knot. Manipulating a deformable object which is not resting presents some further challenges.

Most of the researchers approached the problem by planning the motion in advance and then used either open-loop approach or simple visual servoing to execute the motion. Some works also used demonstrations either collected in virtual reality or with some specialised hardware.

3.2 Manipulating Cloth

Handling garments presents all of the challenges of handling deformable objects we outlined so far: the cloth has very high dimensional configuration space which creates a large number of possible visual appearances. It can also occlude grasping points, making it hard to find desirable grasping locations (such as corners). After successful grasp, any manipulation again influences the configuration of the cloth thus making planning tasks considerably more difficult.

The easiest solution to most of the above problems is to use specialized hardware, specific initial configuration and constrained environments. For example, Nair et al. [33] presented an extremely simple robot that folds laundry using 4 DC motors and specially built frame. Similarly, just introduced commercially available robot

FoldiMate can use specific folding hardware to fold up to 10 pieces of clothing per minute [34]. However, we would like to focus on cloth manipulation using general purpose robotic manipulators that can be deployed on household robotic assistants in the future.

Early work on manipulating cloth with the general purpose robotic arm was done by Fahantidis et al. [35]. They identified four subtasks in the domain of cloth manipulation:

- **Grasping** - this is a two-phase process of grasping the cloth. The system first needs to identify the ideal grasping position and then execute the movement to reach it with its end effector.
- **Laying** - this is a task of putting the cloth to the desired configuration. In the paper, the authors limited the desired configuration space to the cloth laying on a table, but this can be easily extended to any configuration.
- **Folding** - Putting one edge of the cloth over another.
- **Flattening** - task of removing wrinkles from a cloth laying on a table.

The task of folding is usually implemented as a pipeline of other steps, including initial grasping, re-grasping to arrive into the specified configuration, flattening and final folding. We will now look at the specific research papers that made substantial contributions to the specified subtasks and some of which also demonstrated an end-to-end working system.

3.2.1 Grasping

The most straightforward approach to selecting the ideal grasping point to manipulate a cloth is to grasp it repeatedly at different locations until a suitable configuration is found. This approach usually requires a dual manipulator, as described by Osawa et al. [36] Their robot is using two robotic arms to repeatedly grasp the lowest hanging corner, which is determined by analysing the curvature near the lowest point of the cloth. The corners are assumed to have high curvature as opposed to sides of the cloth.

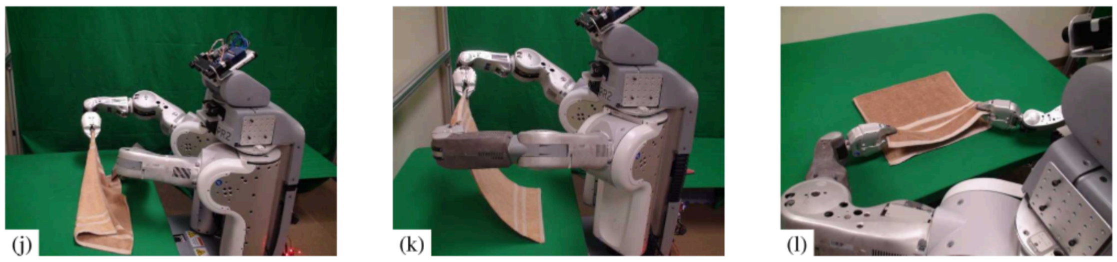


Figure 3.3: Final steps of the first fold - the configuration with robot grasping two adjacent corners has been achieved by re-grasping [37].

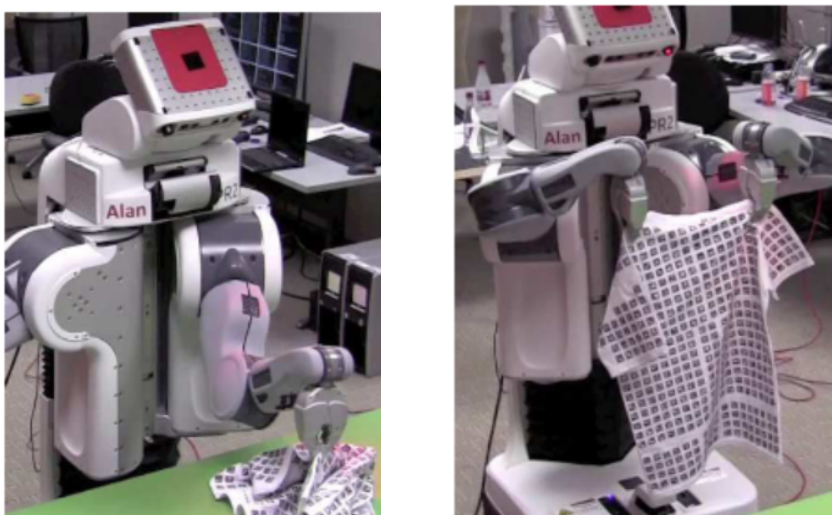
Particularly successful implementation of this approach is presented by Maitin-Shepard et al. [37]. The robot aims to fold a couple of previously unseen towels randomly placed in a pile on the table. It first picks up one or more towels by grasping at a central point obtained by foreground-background segmentation. Afterwards, a corner is localised by evaluating discontinuities on the depth image and finding the lowest point likely to be a corner. The robot then grasps the towel with its second arm and then releases the first arm, which also releases other towels (if the robot initially picked up more than one towel). In the next step, the robot then grasps the second lowest corner (to ensure holding towel by adjacent corners) by its free gripper and straightens the towel on the edge of the table. Finally, the towel is folded by aligning the grasped corners with free corners as shown on Figure 3.3. Multiple re-grasps might be necessary at each step to ensure corners are held. In case of repeated failures, the robot drops the towel and starts again. Overall, it was able to fold the towel successfully in 100% of the trials.

More advanced method of determining the grasping point used machine learning algorithms. Wang et al. [38] used 2D Shape features in Support Vector Machine to determine the configuration of socks before selecting the ideal grasp location. Ramisa et al. [39] used a bag of features algorithm, which is known to be reasonably robust in the presence of deformations because it ignores large-scale spatial configurations of the objects but instead focuses only on localised features. The algorithm was trained with a dataset of images of a polo in random configurations with good grasping points labelled by the human supervisor. During a test run, the system first selected a couple of grasping candidates by sliding a window over the image and evaluating the probability that well-graspable collar is in the window based on a bag of features paired with logistic regression. Best candidates from the first layer are then further refined by applying SVM with Gaussian Kernel.

3.2.2 Brining the cloth to known configuration

Bersch et al. [40] tried to solve the problem of bringing the cloth into a specific configuration by repeatedly grasping as outlined previously. However, they have decided to use fiducial markers to simplify the estimation of the grasp point. Their method can be seen in Figure 3.4 Their robot first picks up the cloth at the highest point in the pile and then uses the depth data along with the markers to estimate the position of the first grasp. The robot then greedily grasps the point which it estimates to be as close as possible to the grasping location in the desired configuration (holding the T-shirt by shoulders). These steps are repeated as necessary. Each grasp candidate point is first evaluated using an SVM trained on almost 500 grasp attempts, and if the model predicts that grasp would fail, the point is rejected, and next best point in terms of the distance to the desired configuration is selected. From the final configuration, it was simple enough to fold the cloth using open-loop routine.

Cusumano-Towner et al. [41] are also solving the problem of bringing the cloth to a known configuration. Unlike the greedy approach used in the methods mentioned above, this system can use prior information about articles of clothing to reduce the number of grasps to reach the desired configuration. The robot first picks up an unknown article of clothing and uses a series of re-graspings as described by Osawa [36]. In this phase, observations are collected and Hidden Markov Model is used to



(a) Robot grasping garment from a pile

(b) Robot holding the garment with fiducial marks visible

Figure 3.4: Different stages of folding as described by Bersch et al [40].

estimate the most likely grasping point as well as identify the article of clothing. The result is then fed into planner and simulator (which is using prior knowledge about the clothing article). The simulator then iteratively constructs a directed graph of grasping points where an edge between two points means that starting from the first grasping point, it is feasible in the simulation to grasp the cloth at the other point. Obtaining the plan that results in the desired configuration becomes the shortest path problem in this graspability graph.

A similar simulation-based approach was employed by Kita et al. [42]. They used cloth function in Autodesk Maya [43] to simulate an article of clothing in different configurations with different grasp points. The robot then picks up the cloth at random location and compares the observed data with the simulation results. The 3D model which best describes the observation is selected and the information is used to find the best grasping position for the second arm to achieve the desired configuration.

3.2.3 Cloth Folding

As we have seen, many authors have found that after the robot grasps the cloth in a specified configuration, it is possible to fold the cloth with an open-loop routine. Specifically, for T-shirts, it involves grasping the garment by shoulders, rotating the sleeves inwards and placing the t-shirt on the table by a folding motion [40]. In case of towels, the robot uses outlined re-grasping methods to arrive at a configuration where the towel is grasped at 2 corners and then a closed-loop routine incorporates visual data in order to align grasped corners to free hanging corners on a table. [37]

More complex folding routines can be also implemented, usually if the robot needs to fold many types of garment. These can be roughly divided into geometry based and simulation-based. Berg et al. [44] pioneered geometry based approaches.

They described how to reliably fold a garment laying flat on the table by exploiting gravitational pull. A geometrical approach might be inaccurate because it only cares about the shape, not the physical properties of the garment. Yamakawa et al. employed a simulation approach [45]. Their robot folds a piece of cloth by rapid, smooth motion inspired by observations of human subjects. In this dynamic approach, timings are crucial for the success of grasping. They, therefore, employed simulation of the cloth to exactly estimate when the robotic arms should be in different trajectory states. Their simulation is enhanced by visual feedback in later papers.

However, most of the outlined methods are manipulating cloth using a dual manipulator, which is not planned for this project. The body of research concerning cloth manipulation task with a single manipulator is significantly smaller. One work is done by Petrik et al. [46], who define a physical model to compute a path for a robotic arm given the grasping location and the line where the garment should be folded.

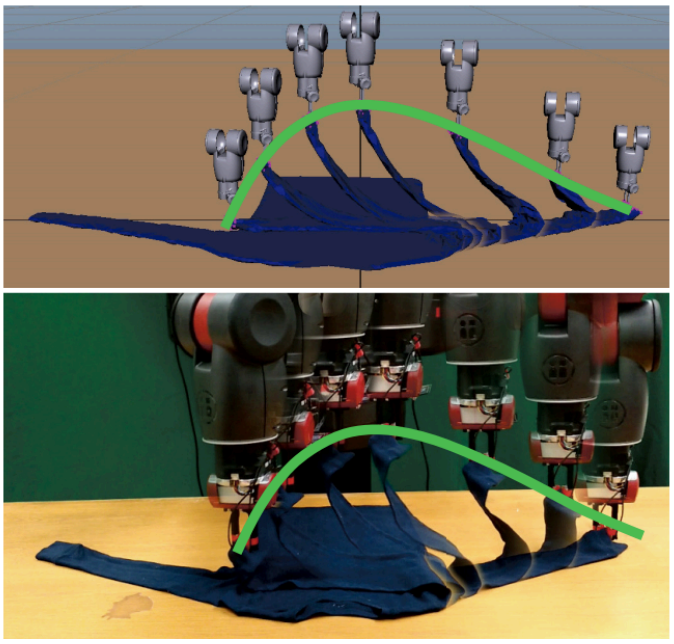


Figure 3.5: The comparison of simulation produced by Maya and real folding scenario as developed by Li et al. [47]. The lines show optimized folding trajectories.

Another paper which describes folding procedure with single end effector was written by Li et al. [48]. This research is particularly relevant for our work because it heavily employs simulations and gives guidance on how to tune simulation parameters. The authors used simulation software Maya [43] in order to simulate cloth as seen in Figure 3.5. They found that most important parameters are frictions between the garment and the table as well as the shear resistance of the garment. Friction can be set experimentally by lifting the garment until is starts to slide and then adjusting friction in simulation until it starts to slide in the same configuration (lifted corner should be elevated to the same altitude in simulation and in real world in the moment when the cloth starts sliding). For shear, a garment should be first laid flat on the table and distance between two extreme points (e.g. opposite corners of the towel) should be measured as L_1. The cloth should then be hung from one extreme point and the distance to the other point should be measured as L_2. The authors then adjusted the shear resistance in Maya until the fraction $(L_1 + L_2)/L_2$ became the same in real world and in simulation. The simulation environment was then used to iteratively optimize the paths of end effectors in order to fold the cloth.

3.2.4 Cloth flattening

Both robots with the ability to accomplish some folding tasks with only a single end effector were assuming the initial configuration of the garment - it was laying flat on a table. Another body of research considers the task of unfolding and flattening the cloth with wrinkles which can be used to get the cloth to lay flat on the table. Sun et al. [49] tackled the problem initially in a simulation environment and then transferred the resulting code to a real-world scenario. They have decided to reduce the infinite search space of possible force applications on a cloth to 8 possibilities - pulling the cloth by its corners and by its edges. They then implemented an algorithm for wrinkle detection by clustering the depth deviations as measured by a depth camera. In each iteration, they selected the longest wrinkle and applied force in one of the eight directions which was the most aligned with wrinkle bisector. A

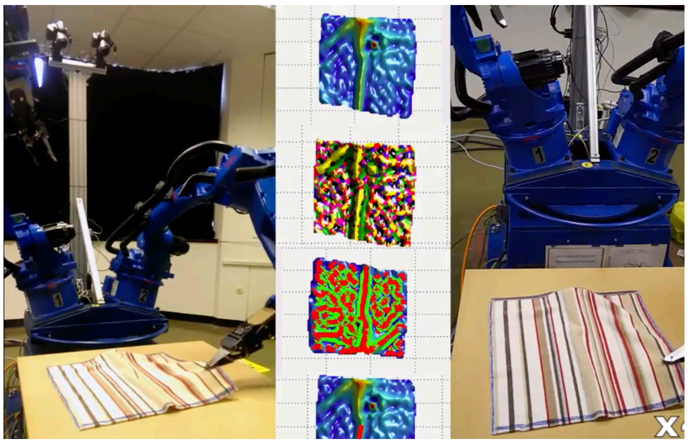


Figure 3.6: Clopema robot flattening a towel with observed depth map in the middle [49].

specific application of cloth flattening is making a bed. The robot needs to flatten the sheets and align them over the bed to adequately cover it. A work heavily related to this project was presented by Laskey et al. [2] The authors used a robot with single end effector to grasp and stretch the sheet to make a bed. The robot learns how to do this task by observing 50 human demonstrations and then using regression to learn proper mappings from sensor data to control impulses.

Overall, we have learned that most cloth manipulation tasks can be split into multiple subtasks. The robot initially needs to grasp an object, then apply series of re-grasps to get it to desired grasping configuration and then apply a final manoeuvre to get the cloth into the desired configuration (e.g. folded). All those steps can be either navigated purely by sensors or also use simulation to aid the decision process.

While operating with only a single manipulator, only a limited set of tasks can be accomplished, such as cloth grasping, cloth flattening or cloth folding starting with the garment laid out on the table. We found the tasks outlined in this section to be very applicable in real world but we could not attempt them due to the limitations of our simulator (see section 7.1.1).

3.3 Machine learning approaches to manipulation tasks

3.3.1 Supervised learning

A significant body of research has been trying to leverage deep learning in the field of robotics. Lenz et al. [50] used 2 deep neural networks in order to select good grasp location for both rigid and deformable objects in Cornell Grasping dataset [51] which contains 1035 images (RGB-D) with positive and negative grasping points annotated in a form of bounding rectangles. The team used all input modalities - colour, depth and surface normals as inputs. Each rectangle was first rotated and scaled so left and right edges correspond to gripper plates and it was then whitened. The first smaller neural network was used to consider every possible rectangle (with some level of discretization) and select top T candidates for grasping. The second network with more features then only evaluated this subset to produce the final candidate. The approach was proven to be successful, as it outperformed other algorithms on the same dataset and achieved the accuracy of 93% (where the candidate rectangle was rejected if orientation differed by more than 30 degrees or if the overlap union of prediction and ground truth was more than 25% bigger than the intersection). The team claims that the results can be transferred to different robots with different grippers, given that labeled training data exists.

3.3.2 Large scale self-supervision

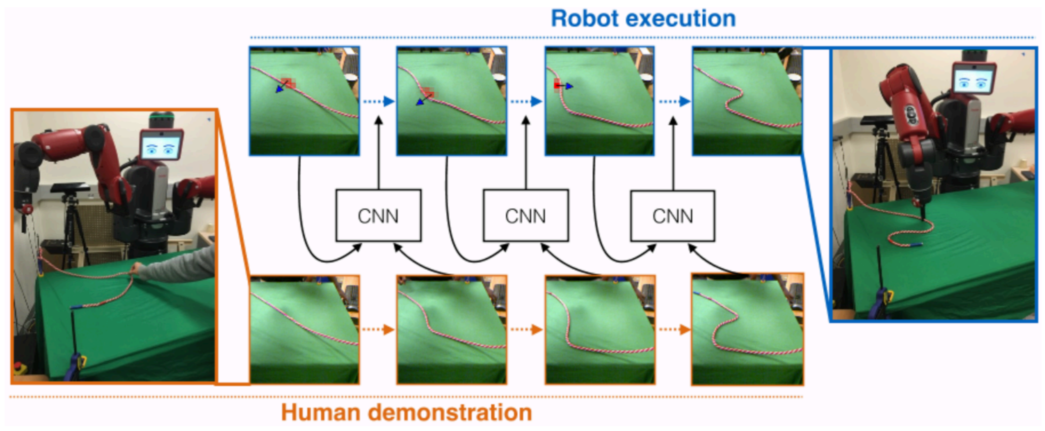


Figure 3.7: Baxter using the self-learned model to imitate the action of human supervisor [52].

However, creating labelled data is a challenging task, that might be very expensive, tedious and error-prone for a human supervisor. This is why some researchers explored the possibility of self-supervised robots collecting data autonomously. Nair et al. [52] tried to develop algorithm that would allow Baxter robot to manipulate a DLO into a set of predefined shapes ("S" shape, "W" shape and a simple knot). The team used a deep convolutional network with the first couple of layers duplicated into two streams. The input to those two streams is a 2D image of ropes initial configuration and 2D image of its next configuration after executing an action. The streams then connect in a fully connected layer which estimates the pickup location and displacement (in terms of orientation and length) that caused the difference between the two pictures. In the test phase, the robot uses this network to imitate the action of the human supervisor as seen on Figure 3.7. It is presented with keyframes of the human supervisor creating one of the 4 shapes. The network, fed with the current image and next "subgoal" outputs the pick up point, the orientation of displacement and length of displacement in order to achieve it. This research is particularly interesting because the robot was able to collect training data by itself - it repeatedly conducted different actions and remembered the image before and after it. This team collected more than 500 hours of episodes with minimal human interaction.

Other authors also investigated the possibility of self-supervision for large-scale data collection. Pinto and Gupta collected 50k grasps and 700 hours of video to create a grasping model based on CNN pre-trained on ImageNet network. The CNN, predicting grasp points, only observed the scene before the grasp begun and did not try to take advantage of continuous servoing. [53] This model showed good level of robustness with over two thirds success rate on unseen objects. Levine et al. [54] used 14 robotic manipulators to collect 800k grasps over 2 months and used the data to train similar network to the one used by Pinto and Gupta but with continuous servoing. The closed-loop approach achieved considerably better results. Gu et al. [9] also note that training times of deep-Q-learning systems (deep RL, see 3.4) can be significantly reduced by using multiple robots that pool their policy updates asynchronously.

Yet another approach to self-supervision was taken by Pinto et al. [16] They used an adversarial strategy, where one robotic manipulator was trying to learn how to grasp objects (both deformable and rigid) and the adversary was trying to break the grasp. One strategy the adversary employed was taking over the control of robotic arm holding the object and shaking it, which forced the weakly grasped object to slip out. Another strategy was to use a second robotic arm to snatch the object by pulling (as seen on Figure 3.8) or pushing it. The shaking adversary strategy saw 2 digit percentage improvements over a baseline initialized by self-supervised learning. Adding the snatching adversary also saw some small improvement. A particularly big improvement was reported for a deformable plush toy, where the original system only succeeded in 20% of the grasps, while the system trained with adversarial data collection succeeded in 60% of the grasps.

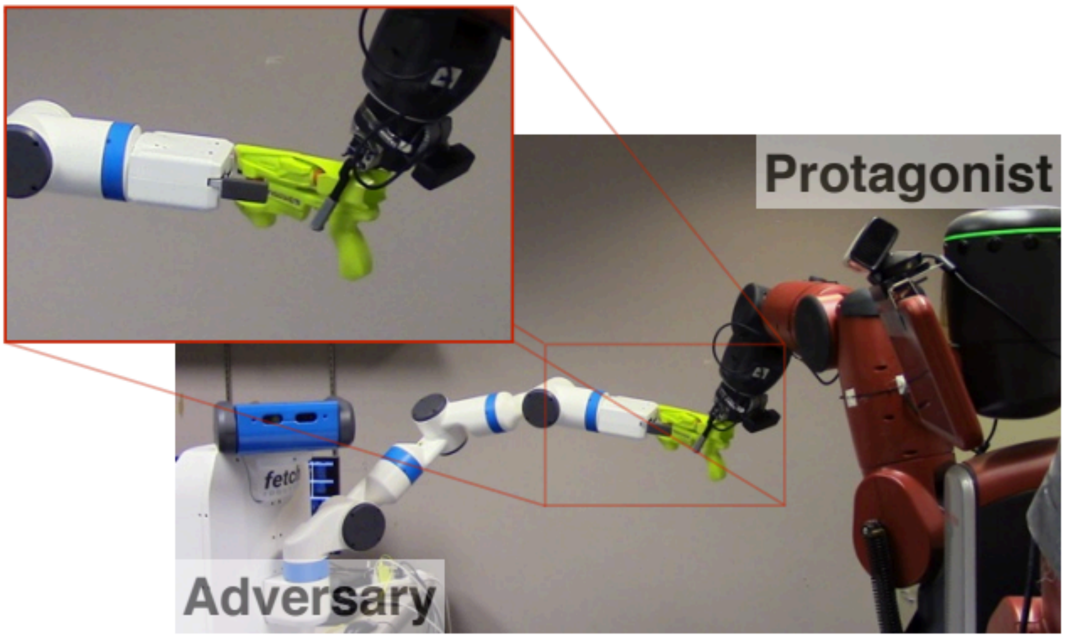


Figure 3.8: Adversary trying to pull a latex glove from the protagonist [16].

3.3.3 Learning in simulation

However, even if self-supervision is employed, it is still expensive to acquire and operate multiple robots for prolonged periods of time. Another body of research attempted to collect the training data instead in simulation and then transfer the learned model to the real-world robot.

The work by Rusu et al. [55] shows how to use progressive networks to execute this transfer. Progressive networks are a new model architecture that allows for a simple transfer of learned model between tasks. The network is built from multiple columns, where each column is a separate deep neural network. The model is initialized with one column and trained on one task (in our case grasping in simulation) and then the first column is frozen. After transfer, more columns are added to the robot with lateral connections to the trained column. The new columns are then trained on a real robot. The results show that the progressive networks outperform classic fine tuning paradigm (the robot trained is first trained in simulation and then the same model is trained further in real-world).

James et al. [13] were successful in fully avoiding the need for further training using a real robot and propose a model trained only in simulation. However, their work was only limited to rigid objects. The task of the single arm manipulator robot was built-up from multiple stages: arrive with gripper to a way-point above a cube, close a gripper, lift the cube, arrive to way-point above a basket and finally drop the cube to the basket. CNN was used to map pixel data to the velocities of the joints, gripper actions, position of the cube and gripper position. The trajectories in training phase were computed using inverse kinematics and all simulation steps were used as a training examples for the network. Authors randomized the environment

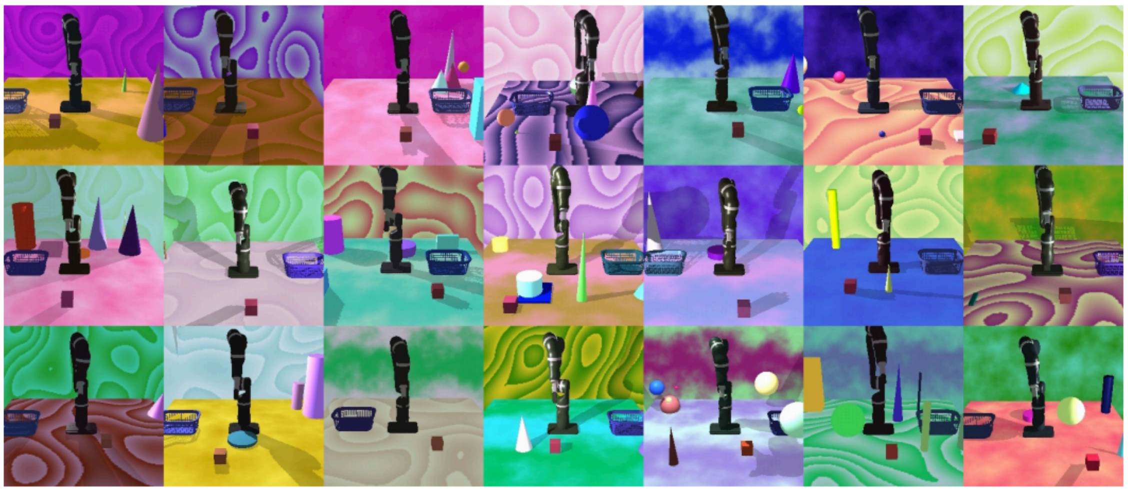


Figure 3.9: Examples of episodes generated by domain randomization. [16]

during simulation to ensure the robot would be able to generalize the knowledge to the task in real world. More specifically, they sampled colours of all objects from normal distribution, uniformly sampled the position of all objects as well as camera and light, sampled the arm configuration from normal distribution and also created random textures for the table and the background (as seen in Figure 3.9). In some episodes, they also added random shapes to distract the arm. The arm was able to achieve 100% accuracy after transition to the real world, if the environment stayed fixed and did not contain distractors. In the case of dynamic environment (moving basket/moving camera), the robot still completed the task in more than three quarters of the cases. From the architectural point of view, authors showed that the networks needs to contain LSTM module in order to be able to complete the task and speculate that otherwise the robot is not able to capture the state (eg. understanding if the gripper is closed). Authors also showed that using joint angles as extra input to LSTM and auxiliary data as network outputs (cube position and gripper position) improve the accuracy of the model after training, but are not essential for the task.

3.3.4 Summary

We have seen a short review of research on leveraging machine learning for solving grasping and manipulation tasks. Overall, we can classify the approaches we have seen into 3 categories:

- **Supervised learning** - the agent requires an extensive annotated dataset of objects with grasping points, which is expensive and tedious to produce. Moreover, this does not scale well to large number of objects or multiple gripper types.
- **Self-supervision in real world** - there are some approaches that simply leave the robot to experiment and collect data which are then used in supervised learning models. Those approaches require hundreds of hours of hardware time to gather the data, which is very expensive.

- **Learning in simulation** - learning in simulation does not require real robots which makes the learning process much cheaper and safer. However, applying the policy learned in a simulation to real world is a challenging task. There are approaches that require further real world training and also approaches that learn to be robust to domain change and hence can be applied directly.

We now move on from looking into supervised learning approaches to the domain of RL. We will however not put a strong emphasis on grasping and manipulation and instead we will review the applications of Deep RL in broader context.

3.4 Deep Reinforcement Learning

We have explained the basics of RL in preliminaries section and we went on describe one of the prevalent RL method called Q-learning where the agent learns to estimate the Q-value (cumulative future reward) of state-action pairs. This Q-value can be encoded in a table for each state and action, but this approach would not scale to high dimensional and/or continuous state spaces and action spaces. One solution is to use deep neural networks to approximate the Q-function.

3.4.1 DQN

Mnih et al. [56] pioneered this approach with a Q-learning algorithm using deep neural network - DQN. As in classic Q-learning, the network wants to approximate the Q-value of each state-action pair. However, they no longer had a table where they simply updated each combination of a action and a state. Instead, they made a neural network that minimised Bellman loss directly derived from the Bellman equation we have seen in previous chapter:

$$L = Q(s_t, a_t)(r + \gamma max_a Q(s_{t+1}, a))$$

where $Q(s_t, a_t)$ was the current prediction of the network and $(r + \gamma max_a Q(s_{t+1}, a))$ was the target. They evaluated this algorithm on a large selection of Atari 2600 games. The observation the agent received was a stack of preprocessed consecutive game frames and reward was a function of the achieved score. The system was able to achieve superhuman performance in some of the games.

However, the researchers have encountered multiple challenges unique to RL:

1. **Objective** - in supervised learning, the network simply needs to learn the mapping between inputs and outputs and the loss can be based on the difference between real and predicted output. In RL, the only feedback the network received has a form of sparse and noisy reward function. In many cases, this reward is not awarded immediately after the right action was taken and it may take hundreds of timesteps to see the results. The network does not have a clear objective on what it needs to learn.

2. **Changing distributions** - as the agent learns, it changes its policy and therefore it is likely to do different actions in the same states as the time progresses.

This becomes a problem when estimating Q-function because taking different future steps makes the current estimate of Q-function inaccurate. In other words, the network trying to map state-action pairs to Q-values is learning a distribution that changes as it learns.

3. **Variance in results** - RL is very sensitive to the selection of the random seed, which makes experimentation extremely time consuming and error prone. For example, an agent that randomly "stumbles" upon good states that are likely to bring rewards will learn the right behaviours quickly. However, agent based on same code might be exploring wrong areas of state space for a very long time, making the code seem buggy, even tough the agent was just "unlucky".

The DQN paper [56] addresses some of those problems. The researchers mitigated the issue of changing distributions by introducing a new trick called experience replay. Instead of just training the network on most recent experience tuples $(s_{t+1}, s_t,$
$a_t, r_t, o_t, o_{t+1})$, they stored each experience in memory and at each step, they replayed a random subset of experiences. This approach greatly improved stability of the agent as it made the distribution change at much slower rate. Moreover, it removed the problem of samples fed into the network being temporally correlated which often led the network to a local minima.

Another important improvement they introduced was the use of a target network. As we can see from loss equation, the target the network tries to predict is $(r + \gamma max_a Q(s_{t+1}, a))$ I also contains network prediction $Q(s_{t+1}, a)$, hence it is a recursive definition. This "feedback loop" can easily cause divergence of learning when all Q-values explode towards infinity. To mitigate this, DQN introduced a concept of target network Q^*, which changes the loss equation to:

$$L = Q(s_t, a_t)(r + \gamma max_a Q^*(s_{t+1}, a))$$

We can see that the main network is no longer optimising with respect to itself (which can cause the network to diverge), but it started optimising with respect to target network. The target network is a copy of the main network that gets re-copied every couple of timesteps. In other words, the target network follows the updates of main network at much slower pace preventing the oscillations and divergence.

One of the limitations of DQN is that the agent needs to evaluate the Q-function multiple times in a single state in order to determine which action seems to be the most promising. This becomes impractical if the agent operates with a large number of possible actions. It is however possible to change the network architecture so it has one Q-value output for each possible action. The network would then take the current observation and compute Q-values for each action in a single forward pass.

This algorithm can only account for experiences of one agent and therefore it can't be parallelized on multiple machines. Deepmind also published a new RL algorithm by Mnih et al. [57] called A3C (Asynchronous Advantage Actor-Critique). This algorithm is in a way similar to DQN, because it uses deep neural network to estimate the quality of states. However, instead of Q function it estimates value function V of the states (critic) and also it estimates the policy π function (actor) which we will explain in more detail in the next section.

Instead of using single agent, the algorithm allows to have multiple workers with each having its own copy of the network. After the workers collect enough experience, they use the data to compute the value function loss and policy loss based on their data. The losses are then used to update the global network, which essentially centralizes the knowledge of all workers. This algorithm not only allows for parallelism, but it also improves learning stability as all agents have different experiences.

3.4.2 DDPG

A large limitation of A3C is that it is only applicable to discrete action spaces. This might become problematic when one tries to control a robotic manipulator with multiple degrees of freedom. Discretizing the action space too coarsely would throw away the capability to do fine manipulation while many discretization levels would create a massive action space (because of dimensionality). An alternative is to use an algorithm capable of working directly in continuous space without the need of discretization.

One of such algorithms is Deep Deterministic Policy Gradient (DDPG [22]) based on prior work in Deterministic Policy Gradient methods [58]. Unlike DQN (and Q-learning), policy gradient methods are trying to learn a policy function directly instead of greedily taking the action with best estimated Q value. It is therefore effective in high-dimensional and continuous action spaces. Before diving into deterministic policy gradient methods, we will first describe stochastic policy gradients that follow a policy π_θ which samples actions from a parametrized probability distribution $\mu(a|s,\theta)$ (θ are trainable actor parameters). Stochastic policy gradients find a good policy by establishing a policy objective function J (for example the value of initial state when following the policy) and trying to optimise the parameters by gradient ascent w.r.t. to J, more formally $\theta_{t+1} = \theta + \nabla_\theta J(\theta)$.

In practice, we can play an episode until the end and store all the transitions (s, a, r). We can then use a so called policy gradient theorem which states that:

$$\nabla_\theta J(\theta) \ = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)]$$

to compute the gradient. Even tough we do not know the true $Q^{\pi_\theta}(s,a)$, we can use the rewards we have collected as an unbiased sample. This is called Monte Carlo Policy Gradient and it is the basis of Reinforce algorithm presented by William back in 1992 [59]. Silver et al. proved that similar method is applicable to deterministic policies and introduced a family of Deterministic Policy Gradient algorithms [58].

However, using just a single episode roll-out at each update introduces a lot of variance in the algorithm which becomes unstable. An alternative is to use a critic network which directly estimates $Q^{\pi_\theta}(s,a)$. DDPG does exactly that - it uses a critic to estimate $Q(s,a)$. Critic is trying to minimize loss based o Bellman equation, which is similar to DQN:

$$L = (Q(s_t, a_t) - r_t - Q^*(s_{t+1}, \pi^*(o_{t+1})))^2$$

The most important difference is that we are no longer maximising with respect to all actions (which would not work in high dimensional continuous action space)

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

.

Figure 3.10: DDPG pseudocode [22]

but we are instead directly plugging in the action proposed by the actor in the next state.

DDPG also adopted the improvements of DQN that guarantee the stability of learning. Instead of training on most recent episodes, it uses a replay buffer to store all the transitions and it uses random samples from learning. As we can see from the loss equation, it also uses target networks ($Q^*(s, a)$ being the target critic and π^* being the target actor). The pseudo-code of the algorithm as presented in the original paper [22] is on figure 3.10. Please note that the authors do not make a distinction between a state and observation, hence the difference in notations.

The exploration/exploitation trade-off in DDPG is addressed by adding noise to the policy actions, such as Gaussian noise. However the authors has found that Ornstein-Uhlenbeck process [60] can produce temporarily correlated actions which is more efficient in physical control task.

DDPG was the main building block for many new algorithms, such as DDPGfD [7], TD3 [61], D4PG [62]. We will describe the main contributions of those algorithms in Chapter 5 where we also describe how they apply to our project and how we used them to improve the performance of our agent.

3.4.3 Other algorithms

We will also do some brief experiments with other algorithms within this project, most notably HER [63], PPO [64] and TRPO [65] but we have decided not to use them extensively. We offer a very brief overview of those algorithms in Section 5.2.3 and we also justify why there are not suitable for the task.

3.4.4 Summary

In summary, we have seen two important Deep Reinforcement Learning algorithms we will further use in this project: DDPG and DQN. The two algorithms are both off-policy, model-free and based around Q-learning paradigm we have introduced in previous chapter. We will evaluate both algorithms and we will use DDPG extensively throughout this project.

3.5 Summary

The background chapter was divided into 3 sections: manipulating deformable linear objects; manipulating cloth; and review of machine learning approaches to manipulation tasks with emphasis on deformable objects.

The first two sections gave us insight about what sort of tasks have been already accomplished in the domain of deformable object manipulation. We have used this data to compile a list of manipulation problems we could consider for this project which we later narrowed down to 3 tasks. Moreover, the previous research motivated us to use demonstrations in our projects and gave us multiple insights on how to successfully tweak cloth simulation to get results close to the real-world behaviour.

In the last section, we examined supervised, self-supervised and RL methods to solve manipulation tasks. This research motivated us to focus on RL in this project, mainly to address the poor scalability of supervised approaches that require human labelling and extreme costs of acquiring data through self-supervision. We have then described the main learning algorithms we will use in the scope of this project.

Chapter 4

Simulator and learning environments

The main prerequisite for the development of good RL algorithm is having an accurate and fast simulator in which we can do our experiments. We begin this chapter by in-depth comparison of 3 simulators that have either direct or indirect support of deformable objects. We later justify our simulator choice and outline further development work we had to do in order to get realistic simulation of a robotic arm interacting with a piece of cloth. In the last part of this chapter, we describe the RL environments we have created for our agent to learn in.

4.1 Simulation approaches investigated in depth

Most general purpose simulators that directly support deformable object simulation are currently targeted at the gaming industry and they, unfortunately, provide little support for robotic applications (eg. loading robot models, interfacing with robot joints etc.) We have looked at PhysX [66] from NVidia and Havok [67] in some detail, but we were not able to find any previous robotics research that used those engines which undermined our confidence in their fitness for our purpose. Moreover, both engines have proprietary codebase which would make it impossible to do small code adjustments when necessary.

We have therefore shifted our attention to Open Source solutions and investigated 3 of them in depth: V-Rep [68], Blender [69] and Pybullet [17].

Figure 4.1: PhysX [66] (left) and Havok [67] (right) physics engines have very good support for cloth and other deformable objects. However, they offer minimum support for robotic applications and they are not open-sourced.

4.1.1 V-Rep

The first simulation tool we tried was V-Rep [68]. The simulator had a model of the robotic arm we wanted to use in the default library and it was easy to control it from Lua scripts. Moreover, the authors of prior work investigating domain transfer, James et al. [13], have used the simulator for this purpose and even provided us with some code to control the simulation from Python. Those two factors justified the need for a more in depth experimentation.

V-Rep provides a front-end to a wide variety of physics engines that can be seamlessly interchanged. It also provides a nice interface allowing the user to create new objects, customize them and generally control many aspects of the simulation. We anticipate that the interface can be extremely useful for quick prototyping and debugging.

On the flip side, the main scripting language of V-Rep is Lua, so the direct interactions need to be written in Lua and Python code only calls Lua functions, which adds another layer of complexity. Moreover, V-Rep does not have first class support of deformable objects and they can only be approximated by different combinations of rigid objects and constraints, which we attempted.

As the first step, we tried to simulate DLO (rope) using rigid cylinders connected by spherical joint constraints. We extended the PyRep library to allow the creation of joint constraints and we managed to assemble a first rope. We found it to visually behave very similarly to a real-world rope, but we did not attempt to create a more objective metric to evaluate the simulation behaviour. It was however not possible to adjust the parameters of the joints directly and the only way to edit the physical properties of the rope was to edit the mass and inertia of the cylindrical links. We believe that this may be a significant problem in future because we won't be able to accurately model arbitrary DLOs. Testing the interaction with robotic arm went well and we were able to manually grasp the rope when controlling the arm with keyboard using a short Python script. We then tried to extend the same

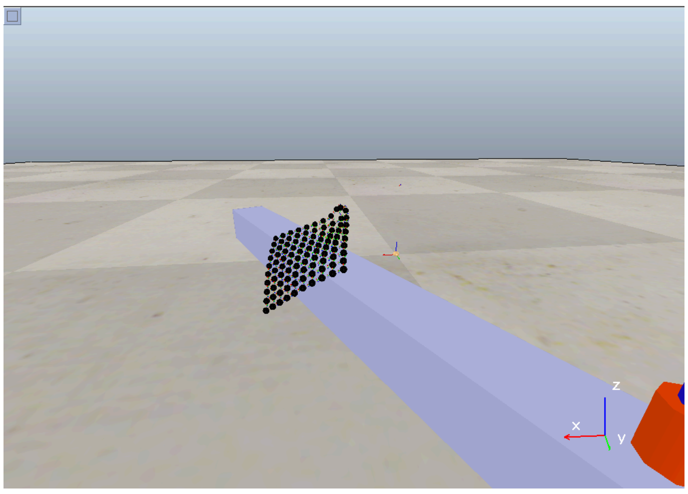


Figure 4.2: Cloth simulation in V-Rep implemented as square lattice of rigid spheres connected by dummies.

approach to a 2D object, such as a towel. In V-Rep, objects exist in a parent-child relationship tree, where a child is usually attached to its parent by a constraint. This was sufficient to model the rope which was a chain of alternated cylinders and joints connected by parent-child relationships. This approach however does not extend to 2D objects because it is impossible to create stable 2D structure without loops. A solution is to use so-called dummies, which are a separate entity in V-Rep and can be attached as a child to any other object. Dummies can be also linked together by overlap constraint, which forces a pair of dummies to move together, so their relative position and orientation with respect to each other stays fixed. We succeeded in simulating 2D deformable object by creating a square lattice of rigid spheres with 4 dummy children. The dummies of neighbouring spheres are linked by an overlap constraint, which forces all spheres to stay together. It is unfortunately impossible to control the forces exercised by the dummies and again the only way to change the physical properties of the cloth is to play with mass and inertia of the spheres. In the case of cloth it however did not work that well - the default cloth was too rigid and resembled a rubber while the cloth with decreased inertia was too prone to breaking. Moreover, the simulation was extremely slow, where only a small cloth of 10x10 spheres contained almost 400 constraints and the simulation was no longer runnable in real time on my mid 2015 MacBook Pro. We anticipate that we would need at least 30x30 or larger lattice to create a visually believable cloth simulation, which would require at least ten times more compute power.

Taking into account the heavy limitations in parameter customization and performance we have decided that V-Rep is currently not suitable for simulating 2D deformable objects, but might be a viable option for simpler simulations with linear deformable objects.

4.1.2 Blender

Another tool we investigated in depth was Blender [69]. This tool is primarily used for animation and graphic work, but it is also possible to use it for simulations. Blender have multiple rendering modes, where the default "Blender Render" is an accurate tool with all the features of underlying physics engine, but it is very computationally expensive and it can't be controlled once the run starts. An alternative mode used by a a community of researchers leveraging Blender for robotics simulations if called "Blender Game Engine". This renderer can be synchronously stepped from a Python API, which makes it suitable for RL simulations where agent decides on the next action of the simulated robotic arm. As a proof of concept, we mention Morse [70], which is a academic-grade robotic simulator that uses Game Engine at its core and allows simulation of dozens of robots.

We attempted to use Blender for our simulation. In the initial work, we tested if the cloth simulation provided by Blender would be accurate enough. We found, again only by subjective judgment, that the simulation acted very closely to how a cloth would behave in real world (as seen in figure 4.3). Building up on a promising simulation of cloth, we tried to create a robotic arm to manipulate it. We ideally wanted to find a simulator that supports URDF imports. URDF files describe how the robot is constructed from meshes and allow users to import their robot descriptions into a simulator. The URDF files are widely used in ROS [71] ecosystem

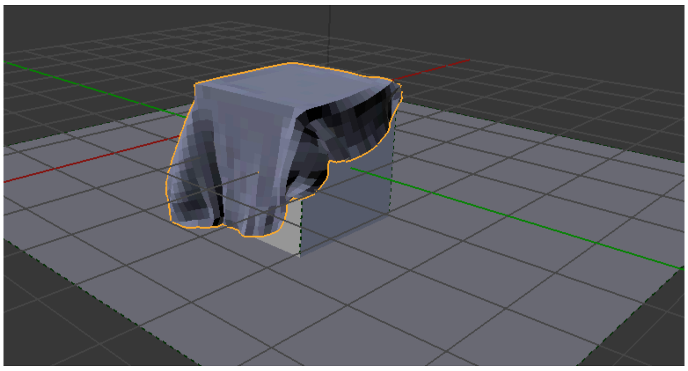
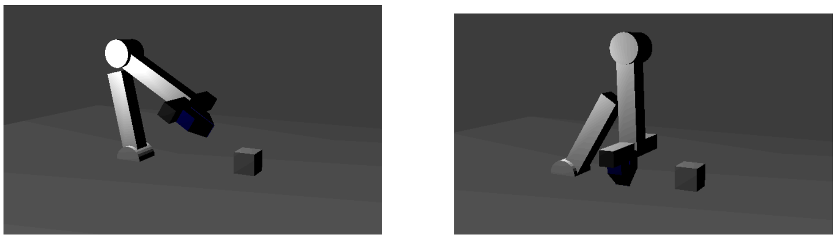


Figure 4.3: Cloth simulated in Blender render engine.


(a) Simple hand-crafted manipulator when there are no collisions

(b) Velocity control allows infinite torque to be exercised, which deforms the joints.

Figure 4.4: Blender game engine arm simulation.

- a large collection of tools used in robotics. Blender does not support URDF robot descriptions out of the box, but there are couple of plug-ins that claim to achieve the task. We experimented with Phobos [72] and URDF importer [73], but unfortunately we did not have much success. The latter seems to be currently abandoned and it did not load at all while the former is still under active development, but it was unsuccessful reading the Mico arm description package. The issue seems to be a known unsolved problem with loading ROS URDF packages. We did not attempt to fix this problem in Phobos code-base and instead we started with a simpler proof of concept custom robotic arm.

We modelled a simple robotic arm with 4 DOF (two hinge joints and gripper claws allowing for linear displacement). However, we found it impossible to accurately control the joints of the robotic arm. When we used velocity control to set angular velocity of each joint, all physical constraints were ignored as the joints were able to exercise infinite force. Alternatively, we could use torque control of the joints and set maximal torques, but this would require another significant development effort to implement PID controllers that could bring the joint to desired position/velocity. Implementing and fine tuning the robot model would take considerable amount of time. Moreover, we found that it is not possible to run Blender game engine in headless mode, which would bring extra problems during prolonged training episodes on cloud machines.

Overall, we found Blender to be a reasonable tool for simulation of even more complex deformable objects (such as cloth), but the absence of features for importing and modelling our robotic manipulator made it unsuitable for our task.

4.1.3 Bullet physics

The last simulation environment we considered was Bullet physics engine [74]. Both Blender and V-Rep are using this engine as a back-end for their simulation, so we might consider them wrappers with good user experience and some extra functionality. We have decided to look into Bullet because it has:

- first-class support for deformable objects (although experimental)
- support for importing URDF robots
- python API with lot of examples (including reinforcement learning ones)
- strong and responsive community with ongoing effort to improve soft-body physics

Again, the first attempted task was to model a cloth. Unfortunately, there are currently no python bindings for creating it, so we had to experiment with underlying C++ API. We also found the library to be fairly unstable with a lot of unexplained segmentation faults (eg. in the same scene, creating a cube caused segfault while creating a cylinder behaved normally). Although we consider this to be a red flag, we found it simple to either fix or work around most of the issues so far. It is possible to create a fairly adequate cloth simulation by our subjective measure, however the cloth did not respond well to our attempts to grasp it. Different settings of parameters (most importantly collision margin and bending constraints) caused the cloth to either "tunnel" through the grippers (collision only happen with the vertices of cloth, which means that it often tears and the gripper slips through the hole) or to be repelled from the gripper.

The library has declared support for URDF imports, so it should not be problematic to import the model of robotic arm. Even all examples provided in the library codebase load robot models written in URDF.

Overall, we found Bullet to be the best option for the task at hand but we still foresee the following challenges:

- **No support for soft-body textures** - soft-bodies are only rendered as a wire mesh in debug renderer, they do not appear at all at renders from simulated cameras.
- **very experimental** - The code using deformable objects seems to encounter segmentation faults quite often and it might be a big development effort to fix some of those.
- **immature** - A lot of features are completely missing, not yet available in Python (many things are only accessible from C++) or work in progress. We will likely need to implement a lot of new features and bind them to python (eg. there is no soft-body anchor removal at all, there are no soft-body constraints exposed to python, no cloth creation exposed to python...). We will describe this work in the next section.

	V-Rep	Blender	Bullet
Soft-body simulation	n	y	y
Importing robot model	y	n	y
Stable	y	y	n
Soft-body textures	n	y	n
Python bindings	y	y	y
Headless operation	y	n	y
GUI for scene creation	y	y	n

Table 4.1: Summary of simulation engine features.

4.1.4 Summary

Overall, we found that each simulation engine has it's own trade-offs. We have chosen Pybullet as the most suitable candidate, mainly because it has first class support for soft bodies and allows us to import the robotic arm with ease. The summary of our findings can be seen on Table 4.1. For more detailed comparison of Bullet with other engines, we refer the reader to study by Erez et al. [75]. In the next section, we will describe PyBullet in more detail and discuss the implementation changes we did to make the simulation more realistic.

4.2 Work with Pybullet

As outlined in the previous section, we have chosen Bullet [74] as the physics simulator for this project. However, the simulation did not work out of the box and we had to make a couple of bugfixes and implement new features. In this section, we will first give a a more in depth description of the library and we will then describe the most important changes we had to implement in the original codebase.

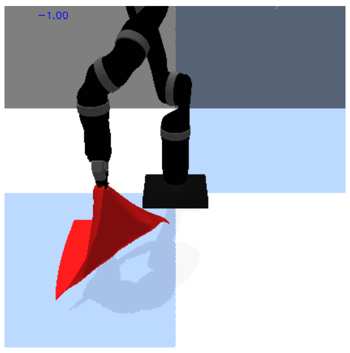


Figure 4.5: Kinova Mico arm holding a piece of cloth simulated and rendered by Pybullet physics engine with our improvements described in this section.

4.2.1 Library overview

The library, currently hosted on GitHub, is dominantly written in C++ with some functions implemented in C and Python. The whole repository has currently almost eight hundred thousand lines of code in those languages, so it can be classified as a large code base. Most of the code has been developed since 2003 by its current maintainer, Erwin Coumans, who joined Google Brain team in 2014. The library sits in the core of many animation and rendering software products used in various industry domains, including film-making. The maintainer along with other contributors won the Academy award in 2015 for the development of this library, because it has been used in simulating complex visually believable destruction scenarios with a large numbers of constrained rigid bodies.

There are multiple APIs available to control the simulation engine. The default one is written in C++ and directly calls the simulator functions. Alternatively, there is also a Bullet C-API that has been wrapped into a Python C extension to create PyBullet. The Bullet C-API is designed to be independent form the physics engine and render engine. The intention of the authors was to allow the user to use only a very small subset of functionality (for example only collision detection or only rigid-body dynamics without the soft-dynamics) so they made the library very modular. The main components relevant to this project are command processor, dynamic worlds, renderers and Python bindings.

The command processor is responsible for accepting commands and routing them to the relevant simulation component. It serves as an entry point for the rest of the system and the file itself has almost 10k LOC. Most of the logic inside this file parses the user command, initializes new objects if necessary, executes actions on them and constructs the response. Many commands need to execute a different set of actions based on the type of objects they are invoked on (eg. creating a constraint between two rigid bodies is different from creating a constraint between a soft body and a rigid body). Moreover, the commands often have multiple flags that alter functionality which further complicates the code. This component also holds most of the state - it has pointers to all objects created by the user, all constraints between those objects, all loaded textures and the pointer to dynamics world those objects live in.

The dynamics worlds are responsible for running the simulation and they execute the step function, which advances the simulation by one timestep. This is done by executing the following:

- **Predict unconstrained motion** - integrate the velocities to get the new 6DOF position of rigid objects (or nodes for soft bodies).

- **Detect all collisions between objects** - this is done by finding one or more contact points between the objects. Each contact point has a distance (either negative for penetration of positive for separation) and a normal vector of the contact. The collision detection is sequential, consisting of 2 phases. The first phase, also called BroadPhase, prunes away pairs of objects that can't be in contact with each other. This phase needs to be fast (avoid $O(N^2)$ algorithms) and is based on axis sweeping. The algorithm walks through each axis and maintains a list of active objects. When a start point of an object is encountered (the point on the object with the lowest coordinate in the given axis), the object is added to the active list. When the object endpoint is encountered, the object is removed. After sweeping all axes, the algorithm needs to only further consider object pairs that have been overlapping (in active list at the same time) in all 3 axes. Those object pairs then advance to NarrowPhase, when they are all analyzed by an algorithm most suitable for the respective type of objects in the pair (eg. (box, box) employs different, simpler algorithm than for example (non-convex mesh, convex mesh) pair).

- **Find constraints** - the simulation usually links multiple bodies by constraints, such as hinges, prismatic joints etc. The role of a constraint is to remove a DOF of the object. If the solver is unable to satisfy a constraint, there is a gap between objects (as seen in figure 4.4). Apart from the static constraints created by the user, there are contact constraints (based on contact points found in previous step) and friction constraints based on the contact normals.
- **Solve constraints** - the constraints in Bullet are solved on the position and velocities level, which means that the solver makes sure that penetrating objects are moved out of penetration state and that objects in contact cease to move towards each other. The constraints are expressed in terms of a system of linear equations and inequalities which are then solved iteratively using variants of the Gauss-Seidel method.
- **Integrate transforms** - after all constraints have been solved, the simulator integrates the acceleration and velocities to get the final transforms, using semi-implicit Euler method.

Each dynamic world implements this functionality for all types of object that live within. In the case of our simulation, those are rigid bodies, multi-bodies and soft bodies. Multi-bodies are essentially an improved version or rigid bodies that implement Featherstone algorithm [76] for computing constraints. In the classic rigid body setting, a constraint removes one or more DOFs by linking two objects that stay independent. In contrast, linked multi-bodies live in parent child relationships and their behaviour is different. By default, the child has no DOFs with respect to its parent. Adding a constraint adds a degree of freedom, so the child can move in its parents coordinate frame.

For example, suppose we have a prismatic joint between two objects. If those objects are rigid bodies, they both have their 6 degrees of freedom and the constraint solver removes 5 degrees of freedom from one of them. On the other hand, if the objects are multi-bodies, one object would be a parent with 6DOF. The other object would be a child with no default degrees of freedom. However, the addition of prismatic joint allows it to move along one axis in parents coordinate frame. In practice, multi-bodies are significantly more stable when working with objects constructed from multiple parts. We will therefore use a multi-body to model the arm in our project.

`Pybullet.c` is a C extension that enables us to call the library functions from Python. It, unfortunately, exposes a very limited subset of all library features and there is a lot of work being done to extend it.

The final component we will be working with extensively are the renderers. Pybullet supports two rendering options: one is based on OpenGL3 and therefore uses

GPU acceleration while the other one, tinyrenderer, is written from scratch and does all computation on CPU. OpenGL renderer is slightly faster (10%) when taking images in large resolution(640x480), but the performance advantage fades when making smaller renders for RL (resolution we used for benchmarking was 80x80). Moreover, OpenGL implementation in Pybullet requires a GUI which makes it difficult to use in headless mode on cloud servers. We subjectively did not notice any significant difference between the quality of OpenGL and tinyrenderer renders.

4.2.2 Original status of softbody support

Even tough soft bodies are implemented in the library, they are not yet officially supported. Erwin Coumans (the main author and maintainer of the library) stressed this on multiple occasions when we submitted a pull request to fix a bug or when we asked for advice on library forums. Originally, the library allowed the user to load a soft body from a .obj 3D model and simulate its interactions with other rigid and deformable objects. There was also limited support for creating constraints between soft and rigid objects, but the constraints were unstable (either too weak to hold the objects in desired relative positions or too strong causing the objects to explode). Finally, there was no support for rendering soft objects whatsoever - they were only visible through debug draws in Pybullet GUI application.

4.2.3 Contributions

We had to do a significant amount of development in order to create a stable, visually believable and reasonably accurate simulation of deformable objects. Whenever the time allowed, we also contributed the code changes back to the library repository for the benefit of the wider community. This section summarizes the most important work on the library:

1. **API for deformable object creation** (including cloth) - the library did not expose any commands for creating deformable objects through Python code. We have identified the most important features of the underlying C++ codebase and wrote Python bindings so they can be called from Python code. The biggest challenge was to ensure proper memory management - Python uses reference counting to work out the lifetime of an object and we had to make sure all objects used on the borderline between C code and Python code are destroyed at the right moment.

2. **createCloth method in C++** - so far, the only way to create deformable object was to load a .obj file. This method was not a good fit for a task of creating 2D cloth, because many internal parameters and consequently the behaviour of the object depended on the 3D model used. Instead, we created an API function that allows users to create 2D cloth directly, specifying all important parameters (location of corners, number of internal nodes, mass, node linkages). The implementation took advantage of undocumented helper function in soft body codebase that creates a soft body patch from a grid-like arrangement of nodes. We have also exposed this function to be called directly from Python.

3. **Cloth parameter search** - a significant engineering effort was necessary to understand the internals of cloth simulation and to search for a reasonable set of parameters to create a realistic simulation of cloth interacting with the robotic arm. Overall, there are two main implementations of soft body simulation - default and cluster. In default simulation, soft bodies detect collisions using the vertices and faces. However, if there are not enough vertices in a given area, it is likely that a soft body would miss a collision and rigid body will simply fly through it without any interaction (so-called tunnelling). Another method of simulation decomposes the soft body into sets of convex clusters that can use more accurate collision detection for convex shapes and decrease the likelihood of tunnelling. In our experiments, we found that cluster collisions work fairly well for volumetric bodies that mostly return to original shape after deformation. However, a cloth simulated with cluster-collisions was always floating far away from rigid objects (due to collision margin), it was too stiff and it did not hold the deformed shape well (crumbled cloth would simply flatten out over time). We have decided to use the default simulation and work on parameter selection. We found a good value for the friction parameter in a similar fashion to the method suggested in [48] (lift a corner of the cloth until it starts sliding and record the height, then lift in simulation to the same height and decrease friction until sliding occurs), found the approximate mass by measurement and found linear and angular stiffness by experimentation with a real and simulated cloth. An especially important parameter was cloth damping used to reduce the buildup of kinetic energy stemming from the inaccurate simulation. It accelerates the decay of node velocities. Small damping would make the cloth waving in the air even in absence of any manipulation while large damping would make it impossible to move. We made all parameters easily configurable from Python code to allow quick changes in the future.

4. **Rendering** - the original library did not have any support for rendering deformable objects. The only way to see them was to enable "debugDraw" functionality in the Bullet ExampleBrowser. This functionality shows wireframes of all simulated entities in OpenGL GUI window. However, those wireframes were not rendered when using the camera APIs. We have implemented softBody rendering both in OpenGL and in Tinyrenderer to be able to contribute this change back to the main library repository. We have used the functions for rendering a rigid mesh shapes as an inspiration and implemented code that draws the vertices and faces(triangles) of the soft body, which allowed us to render the body once. It was however not possible to synchronize the rendered body with rigid simulation as the renderers only support 6DOF translations. We have ended up deleting and re-drawing the body after each time step to render the object in perfect synchronisation with physics simulation. We also had to disable backface culling in some situations as the deformations sometimes caused a backface of an object to be facing the camera.

5. **Memory leaks** - the soft body implementation had a large number of memory leaks that caused a massive increase of memory use during training. We

have logged the RSS (resident set size) on every time step during training runs and we have noticed that the simulator is loosing almost 5GB per hour, which meant that even a machine with 64GB of memory will OOM after approximately 13 wall-clock hours of simulation. We have spent a couple of days by profiling the code and trying to identify memory leaks. We have used Valgrind to identify the leaks, but this proved to be difficult due to a large amount of noise generated by Python and the library itself (even after installing python suppressing files). We have then used strace to find the code sections allocating biggest chunks of memory, but this was again too noisy and it was hard to filter out legitimate uses. Finally, we have tried a Mac tool Instruments, which pointed us directly to a couple of small memory leaks and also helped us identify large leak by investigating the lines allocating most memory (similar functionality to strace but with an option to aggregate system calls by issuing line). This largest leak was caused by a soft body collision detection component `btSparseSDF`. The component was allocating memory in a hash table as a memoisation technique and resetting the memory when it reached a certain threshold. However, resetting the simulation during training (for example after the end of a training episode) did not free the memory of this component and left the massive hash table undestroyed. Fixing it was as simple as resetting this component on each simulation restart. We have also identified a couple other memory leaks caused by faulty reference counting in my bullet C extension, some of which were in original code and some of which were in the new code we have added. We show the overall decrease in memory use in Evaluation (Chapter 7).

6. **Segmentation faults** - We have noticed that the library sometimes throws a segmentation fault after many hours of operation. The issue did not appear deterministically at the same point in time. Sometimes it happened after 6 hours of training, sometimes after 20. At first, we wanted to overcome the problem by checkpointing the training after every 5 epochs and reload the model if the fault occurs. However, we were not able to also checkpoint the replay buffer (16GB+ of data) and we observed that restarting learning from checkpoint with an empty replay bfuffer adds noise to the network and causes both immediate and long-term drop in performance. It was also prohibitively expensive to find the fault using traditional tools such as Valgrind because running a version without compiler optimisations with instrumentation overhead would likely make it occur only after multiple days. Finally, we looked at stack trace generated by the fault handler decompiling the optimised library and identifying roughly the location of the fault based on offsets from symbols. We found that the fault is happening in Tinyrenderer shader and is stemming from out-of-bounds array access in certain combinations of light position and position of the fragment being rendered. We resolved this by properly checking array bounds before executing the access.

7. **Softbody anchors and fake object grasping** Initially, we have spent a lot of time trying to optimise cloth parameters so they work well with grasping based only on the physical forces between the cloth and the gripper. How-

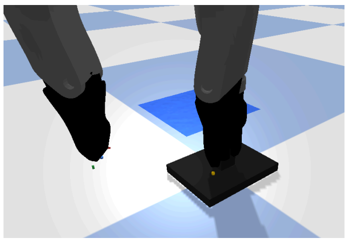


Figure 4.6: Close up look at the gripper with rigid objects used to anchor the cloth for the purposes of creating "fake grasp". There are 6 objects in total to improve stability of the grasp and to avoid the spinning behaviour.

ever, we have encountered multiple issues. The gripper was often tunnelling through the cloth or it was repelled from the cloth due to the large collision margin. Decreasing the collision margin would cause the cloth simulation to be unstable, which can for example manifest by cloth falling through the ground. The strong repelling caused the cloth to pop out of the gripper even when it was properly held. The solution we introduced was to create a fake grasp if the gripper fulfils two conditions: it needs to be closed (joint angles indicating that the gripper fingers are close together) and the end of gripper fingers need to be very close (approximately 2 cm - configurable) from the closest cloth node. However, the library did not have any support for the soft body/multi-body anchor and we have decided to model the arm as a multi-body. The workaround was to introduce a rigid object at the tip of each finger and anchor the cloth to those rigid objects with a rigid body/soft body anchor. Those objects have fixed positions in finger reference frame so they move with the finger. Even though two such objects (one on each fingertip) should be theoretically sufficient, we have found the cloth to be much more stable when we added 3 objects for each fingertip (one in the middle and one on each side). Multiple constraints stopped the cloth from spinning unnaturally in the air when held above the ground and made the grasp more stable. The next step was to create anchors between those rigid bodies and the closest cloth nodes. However, we have found that the basic anchor code is not sufficient as it can't create a rigid attachment. If the anchor strength parameter was set too low, the cloth would be levitating in the air a couple of centimetres under the gripper. Setting it higher made the impulses coming from constraint solver too strong and the cloth exploded when it was touched. We rewrote the anchor constraint implementation to make anchored node exactly follow the position of rigid object no matter what other forces act on it while significantly decreasing the impulse generated by the constraint. This implementation of the constraint resulted in fairly stable and believable interaction with the gripper but it probably would not be generally applicable in other scenarios. We also had to expose the creation of soft body-rigid body anchor to the Python API.

8. **State saving and soft bodies** - Pybullet already had APIs for serialization of rigid objects into a file, which is then used for saving and restoring the state of the simulation. The engine creates a `.bullet` file which contains information about positions, velocities and constraints of all objects. It is therefore possible

to create a simulation checkpoint by saving the state to a file and then restore exactly the same situation by reloading the file. However, soft bodies were not included in those checkpoints. We have added the information about node positions and node velocities to the file so it can be easily reloaded. We were worried that saving just the positions and velocities would not be sufficient and the cloth would be unstable when reloaded, but this luckily was not the case. This feature is important for resetting to demonstration improvement we describe in section 5.3.5.

9. **Other contributions:** writing and exposing code to query cloth internal state (returns positions of 4 corners in the world frame), resolving compilation on Mac (this was also since fixed independently by library maintainers), experimentation with compiler options on different architectures to improve performance

Existing limitations

Even after the changes we have implemented, the simulator still has some serious limitations, mostly to do with soft body simulation. The largest problem we have experienced is the lack of self-collision capability of soft bodies, which is especially relevant for working with cloth. The deformation of a cloth when hanging freely is not completely realistic because the sides of the cloth pierce through each other. When folding, the agent learns to drape the grasped edge of the cloth too close to the surface because the cloth does not collide with itself in simulation. When the cloth is folded, we can see visual artefacts caused by two layers of cloth being exactly at the same height because they do not collide with each other, only with the surface. Even tough those are serious issues that certainly negatively impact the performance of trained agent, they are not enough to justify the time investment of implementing cloth self-collisions.

Another serious limitation is the lack of deformation stability. If we crumple a piece of cloth in real world, it tends to stay in the crumpled state. However, if we do the same in simulation, the cloth slowly unrolls until it reaches its original shape. This can be mitigated to some extent by selecting correct cloth parameters (large mass, small stiffness coefficients, large damping), but those negatively affect the behaviour of the cloth in other situations. As a consequence of this limitation, we could not attempt some of the tasks we have planned to do after reading the relevant literature (background section). More specifically, we could not do cloth de-wrinkling or flattening, because cloth left alone would simply de-wrinkle over time without any robot interaction. Similarly, we could not try unfolding tasks and tasks of getting cloth into a known configuration because the cloth would be unrealistically changing its configuration in simulation and the domain transfer would be therefore impossible. We tried to fix this by changing the cloth deformation parameters or even freezing the cloth when it is not manipulated, but we could not achieve realistic behaviour.

Summary

We have decided to use Pybullet [17] (and underlying BulletPhysics C++ library) as the simulation engine for this project. The engine can be used for simulating deformable objects, but it is still an experimental feature not officially supported by developers. We have leveraged the existing code, fixed the most serious problems and implemented new necessary features in order to get realistic cloth behaviour in some subset of manipulation scenarios. Even tough the engine became sufficient to simulate some manipulation, we still felt limited in the range of tasks we could use to evaluate our algorithm. In the following section, we will describe the environments we have ended up implementing.

4.3 Environments

A vast majority of contemporary RL research is leveraging a battery of test environments called OpenAI gym [19]. Each environment in OpenAI gym toolkit is a well-defined task that makes no assumptions about the internal workings of the RL agent. All environments implement a very simple API that the agent uses to interact with them:

- `env.action_space` - the field contains information about the type and shape of the action the environments expects. For example, Atari game environments use discrete actions corresponding to button presses while most robotic manipulation environments expect continuous multidimensional actions corresponding to joint velocities or end-effector velocities.
- `env.observation_space` - the field contains information about the type and shape of observations the environment offers. Those can be low dimensional vectors or high dimensional pixel representations.
- `env.reset()` - resets the environment instance to initial state and returns initial observation (which has the type and shape as described by `env.observation_space`).
- `env.step(action) -` advances the simulation one step forward. Takes the action the agent will execute and returns a 4 tuple `(observation, reward, done, info)`. `observation` - represents the result of agents action on the environments, `reward` is a (possibly sparse and delayed) measure of agent's success, `done` contains information whether the environments is in a final state and should be reset and `info` is a dictionary with additional user-specified data.

The idea behind offering a suite of RL tasks with simple API is to allow researchers to benchmark their algorithms and compare the results on some well-defined environments. OpenAI gym even exposes functions to collect the data from an experiment and upload them to a shared leadderboard.

We have also decided to use OpenAI gym for this project because we wanted to have access to the results of other algorithms on a given task. This would allow us to easily judge if our algorithm is flawed or if the task is simply too complex to complete. In the following subsections, we will give a summary of environments we have used and implemented over the course of this project.

4.3.1 2D environments

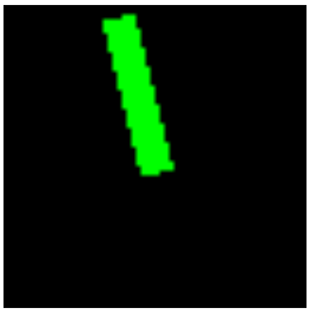
We have started focusing on RL aspect of this project in mid February after getting some initial visually realistic cloth behaviour in simulation. We have been very optimistic and tried to apply some off-the-shelf RL algorithms on the cloth-folding task and hoped to get some quick results. However, none of the algorithms we tried showed any "signs-of-life" and acted more or less randomly. In order to sanity-check the algorithm implementation, we have tried running it on two classic control OpenAI environments - `cartpole_v0` and `pendulum_v0`. Both environments are very popular benchmarks in RL community and they have been solved by many published algorithms.

In `cartpole_v0`, the role of the agent is to balance a pole on a small car moving in 1D. The agent can only issue two discrete actions - go left or go right. It gets a reward of 1 for every time-step while it manages to keep the pole upright. The observation is the position of the cart, velocity of the cart, angle of the pole and angular velocity of the pole, so it is a low-dimensional vector. In `pendulum_v0`, the role of the agent is to apply torque on a pendulum to make it stay pointing upwards. The agent can only apply a scalar action corresponding to a torque of a pendulum joint. Too strong action will result in pendulum swinging over the top and going to far, while weak action will not prevent the pendulum from falling to its natural position pointing downwards. The observation is the sin and cos of pendulum angle θ as well as the angular velocity ω of the pendulum.

We have found the implementation of DDPG from OpenAI baselines [77] to converge quickly on both environments. The next step was to try working with high dimensional observation (learning from pixels). However, we could not find a simple classic control OpenAI environment that provided pixels outputs. We have therefore decided to adapt the `cartpole_v0` and `pendulum_v0` to provide pixel observations.

We used OpenCV [78] to draw shapes corresponding to the pendulum. It was straightforward to translate the low dimensional state to the coordinates of shape to draw (first find corners of a rectangle with desired dimensions and then use $sin(\theta)$ and $cos(\theta)$ to construct a rotation matrix to rotate the corners and finally draw a polygon with vertices at the rotated corner points). The result can be seen on Figure 4.7.

However, applying the algorithm to the new environment with pixels observation was disappointing and the performance was only marginally better than the performance of random agent. We have eventually found out that the problem was not the learning algorithm, it was the environment. The render for pendulum turning clockwise and counterclockwise was exactly the same and the agent, having no memory, had no way of inferring the angular velocity of the pendulum. As a result, it could not know what torque to apply . We have fixed this issue by colouring the pendulum according to its angular velocity so the colour changes based on the


(a) Our observation of pendulum environments rendered with OpenCV.

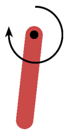
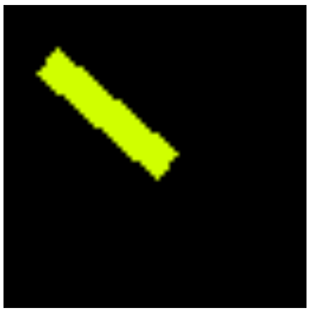
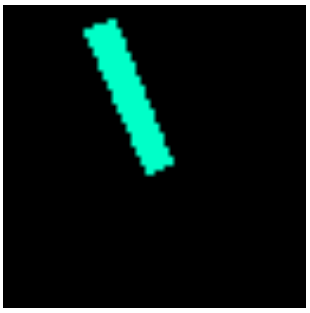

(b) Original OpenAI pendulum render, which is not accessible through API and can be only visualized in window.

Figure 4.7: Original render of `Pendulum_v0` not available through API and our render, which can be used as observation for learning from pixels.


(a) Pendulum travelling counter-clockwise.


(b) Pendulum travelling clockwise.

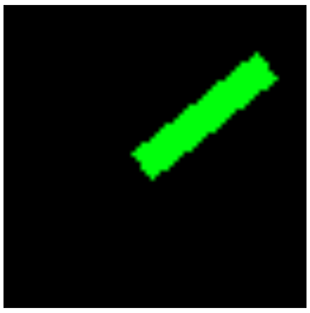

(c) Stationary pendulum.

Figure 4.8: Pendulum captured with three different angular velocities.

direction the pendulum turns in. The RGB formula was the following:

$$color_{RGB} = (\omega * 32, 255, -\omega * 32)$$

The result was clipped so it was always in (0, 255) range. The pendulum turning counter-clockwise (positive ω) was therefore yellow and when turning clockwise, it was a mix of green and blue (aqua). After this change, the agent learning from pixel observation was able to quickly learn a correct policy and converge to a result comparable to the agent learning from low dimensional state.

We have also tried stacking a sequence of four consecutive observation frames instead of colouring the pendulum, which allowed the agent to infer the the velocity and acceleration. Although this also worked and the agent was able to achieve good results, the final average reward was lower than for the colourized version and the training took longer. We have used the 2D environments for our initial experiments, because they are much easier to solve and quicker to simulate. We could therefore get the results quickly.

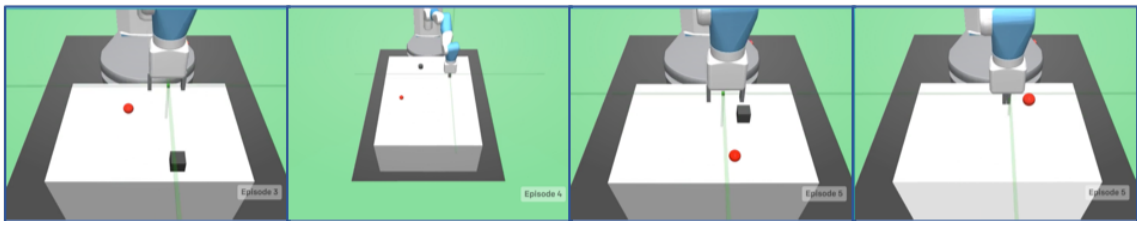


Figure 4.9: 4 OpenAI robotic manipulation environments using Mujoco simulation and Fetch robot platform. From left: Grabber, Slider, Pusher and Reacher.

4.3.2 3D rigid object environments

After successfully solving the 2D environments, the natural next step was to attempt some 3D environments. We have initially tried to use OpenAI robotics gym environments [20] based on Mujoco simulator [79] to get a benefit of benchmarking against other implementations. However, those environments require Mujoco simulator license. It is possible to obtain 1 free license for educational purposes, but even this license is tied to a specific machine (the user needs to upload a machine-unique fingerprint which is validated on each simulator launch). Changing the licensed machine involves a lengthy process of communication with Mujoco sales department. Therefore licensing our development machine (personal MacBook Pro) would prevent us from training on a DoC machine which would considerably slow down our iteration speed and overall progress. Moreover, OpenAI robotics environments are based on Fetch robotic platform [80], which is not available in our lab.

We have decided to avoid using environments tied to Mujoco simulator or using Fetch robot platform. Instead, we reimplemented the environments in Pybullet using a robot platform we have access to. There are 4 OpenAI robotic environments we have re-implemented:

- Reacher - the task of the agent is to manipulate the robotic arm to reach a specific target location sampled from a box area in front of the robot. In sparse reward case, the agent gets the reward when the effector reaches the target position (within a small threshold). In dense case, the reward is computed as euclidean distance between the end effector and the target position.

- Pusher - the agent needs to control the arm so it pushes a small cube to a target location. The position of the cube and the position of the target are randomly sampled from the area in front of the robot and they are both guaranteed to be on the table so there is no need to grasp the object. Sparse reward is awarded when the position of the cube is within a threshold from the target position, otherwise the agent receives -1 on each timestep. In dense reward case, the reward is proportional to the euclidean distance between the cube and the target position.

- Grabber - similarly to Pusher, the agent needs to move the cube to the target position. However, the target position is usually sampled in the air, so the arm needs to grasp the object in order to lift it. The rewards structure is the same as with Pusher.

	Options	Description
Action	IK (default)	The control mode inspired by OpenAI Fetch robotics environments. The action contains velocities of end effector (3 scalars) and velocity of the gripper (opening/closing - 1 scalar). Velocities are scaled so the robot never moves faster than 0.5 m/s.
	Velocity	The classic control mode used in most of manipulation research. The action directly controls the angular velocities of all joints within the arm (10 scalars)
Reward	Dense	The reward is equal to the distance between end effector and target (Reacher) or cube/puck and the target (other environments)
	Sparse	+3 if distance of cube/puck/end-effector(Reacher) to the target is bellow the threshold, -1 otherwise
	Positive (default)	+100 if distance of cube/puck/end-effector(Reacher) to the target is bellow the threshold, 0 otherwise
Observation	low_dim	Contains low dimensional state of the system (position, target position, gripper finger angles, cube/puck position, cube/puck relative position to the gripper, arm IK goal, arm fingers goal, gripper velocity, wether gripper is holding and object).
	pixels (default)	RGB render of the scene from external camera overlooking the robotic arm and the workspace. The dimension of the observation is (84, 84, 3) corresponding to width, height and number of channels in an image.
	pixels stacked	Same as pixels, but containing a sequence of 4 consecutive frames, so the dimension is (4, 84, 84, 3). The agent should be able to infer acceleration and velocities at a cost of extra computational load.

Table 4.2: The action, observation and reward modes of 3D manipulation environments implemented in Pybullet [17] based on OpenAI Fetch robotics gym [20]

- Slider - the agent needs to slide a puck on low friction surface towards a target, which is sampled out of reach of the robot. The reward structure is again the same as in Pusher case.

All OpenAI Fetch robot environments are controlled by a continuous 4-dimensional action that contains end effector velocities along x, y and z axes as well as gripper velocity. The state in those environments is low-dimensional and contains the position and velocities of the gripper and cube/puck (if present). The observation contains the state, the desired goal (position of the target) and also achieved goal (this is the position of the gripper for reaching or position of the cube/puck for other environments). The composed observation is used so the environment can be used by algorithms based on Hindsight Experience Replay (HER) [63] (described in more detail in section 5.2.3). The composed observation can be flattened for use with algorithms that do not user HER.

Our implementation of the environments follows the same action and reward structure, however we add other options configurable via keyword arguments. The summary of the options can be seen in Table 4.2.

In IK (inverse kinematics) control option, the environment remembers a goal position of the end effector which is an IK target for each step. The velocities passed in through action actually act on IK target, so at each step, we just add the action to arm goal position. This implementation effectively acts as a low pass filter. If the desired velocities are too large and the arm can't keep up with them, the changes to goal position are clipped so it is always within a short distance (7 cm) from the current end effector position. Hence, we do not need to always scale the actions so they are achievable by the arm (this would be very difficult because maximal speed of end effector depends on its position). Both IK goal position and real position are included in the low dimensional state, which helps the agent infer the direction of the arm.

We encountered a couple of challenges when re-implementing the OpenAI environments in PyBullet:

- **URDF description** - we have found that the robot description provided in Kinova SDK [18] does not work with Pybullet. We had to first translate the .xacro provided by the manufacturer to .urdf file which can be loaded by the simulator. Xacro is a file format that aims to reduce the size of urdf descriptions by removing the unnecessary repetition of elements. There is a Robot Operating System (ROS) [71] package that can do the translation. We have found that ROS support for MacOS X is very experimental and we have encountered a large number of problems during the installation which ultimately prevented us from using ROS on our development machine. As a workaround, we used an AWS EC2 instance with Ubuntu installation on which we installed ROS without problems. We still had to add a world to robot link to the urdf description in order to make the simulator load the file.

- **3D models** - the 3D models supplied by the manufacturer were not compatible with Pybullet, which was manifested by the simulator rendering a simple shape instead of the mesh. We have loaded the models in Blender [69] and exported them in `.STL` format. We knew this format was supported because it was used to store models used in exampled provided with Pybullet codebase.

- **Target rendering** - we are not aware of a way to render a visual-only object in Pybullet, which we found problematic when rendering the target. Ideally, the target would be visible on renders (so it can be used as pixel observation) but it should not collide with any other objects. At the end, we used a visual shape of the desired size (sphere with radius of 2.5 cm) with a tiny collision object at the centre. We did not encounter issues with the object colliding with the arm.

- **Grasp quality** - as we explained in the section about work with simulator, it is often insufficient to rely on accurate physics simulation when grasping deformable objects. We have observed similar issues when grasping rigid objects. The cube grasp was unstable even if the gripper held the cube at ideal grasp points (centres of opposite cube faces). We have applied fake grasp solution again - when both fingers were in contact with the cube and the finger positions were in closing motion, we create a constraint between the finger and a cube with origin at the contact point. This resulted in a stable grasp.

The implemented environments can be seen on Figure 4.10. We will elaborate more on agent performance in the next section.

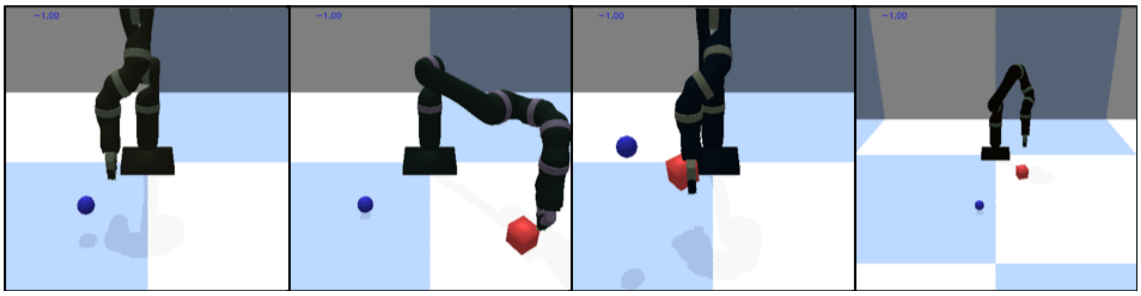


Figure 4.10: Rigid object environments implemented in Pybullet. From left: Reacher (arm needs to reach the blue sphere), Pusher (arm needs to push the cube to the blue sphere), Grabber (arm needs to lift the cube to the sphere which is above ground), Slider (arm needs to slide the cube towards the blue sphere which is out of its reach)

4.3.3 3D cloth environments

Final set of environments used in this project finally employs a deformable object - a piece of cloth. Cloth manipulation has been selected as the main focus of this project from a variety of reasons. On the one hand, we chose it over deformable linear objects (ropes, cables) because we were interested in the challenges of cloth simulation (much harder than DLO simulation), because there is a more significant literature gap on the topic of applying machine learning to cloth manipulation and finally because it has more applications in home assistance robotics, which is a domain we are interested in. On the other hand, we chose it over deformable 3D objects (plush toys, sponges etc.) because the deformation behaviour of those objects is much weaker and often can be neglected in manipulation tasks. In fact, some robotic research on grasping [16] uses exactly the same approach for grasping rigid and 3D deformable object. Hence we have decided to dedicate this project to manipulation of cloth and more specifically we focused on manipulating face towels and large towels.

Due to the limitations of the simulator (as explained in section 4.2.3), we were not able to focus on any cloth unfolding/flattening tasks because the simulated cloth simply flattens itself over time and we were not able to fix this problem as it seems to be a fundamental limitation of the simulator. We have instead decided to attempt tasks which start with a cloth laying flat on a table (spawned in a random position) and we propose 3 different tasks (as also seen on Figure 4.11):

- **Diagonal folding** - a square piece of cloth (with configurable size) is spawned at a random position in front of the robot. The task of the agent is to grasp any corner of the cloth and fold it diagonally (forming a triangle). The agent is successful if a.) two diagonal corners are aligned within a small threshold (6cm), b.) the cloth is no longer grasped by gripper, c.) all pairs of adjacent corners are in a distance larger than 66% of their original distance (this is to prevent crumpling of the cloth).

- **Folding up to a tape** - this task was used for evaluation of another deformable object manipulation algorithm (based on trajectory aware registra-

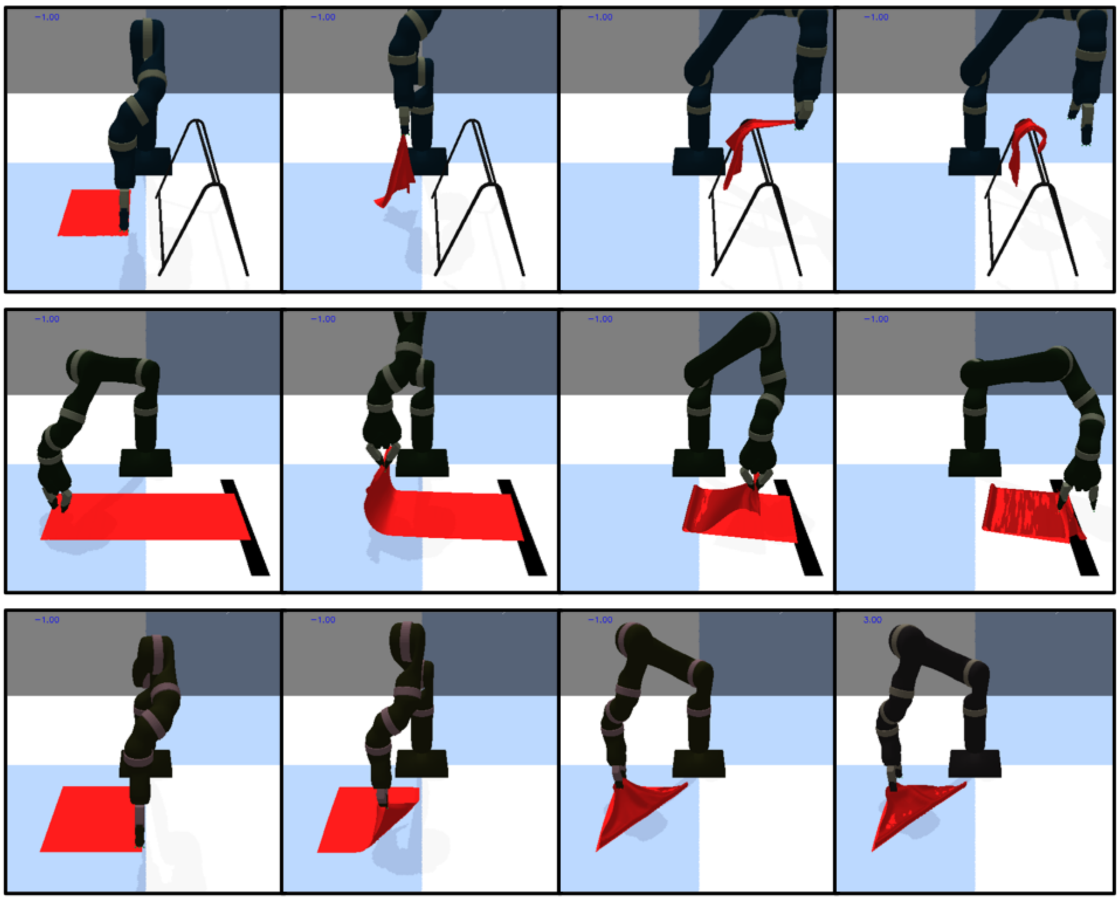


Figure 4.11: Cloth manipulation environments we are solving in this project. From top: Hanger (arm needs to pick up a square cloth laid on the left and drape it over a small hanger), Tape (arm needs to fold a large towel up to a tape), Folder (arm needs to fold a square piece of cloth diagonally to create a triangular shape). Each row represents a sequence of selected frames showing an agent successfully completing the task.

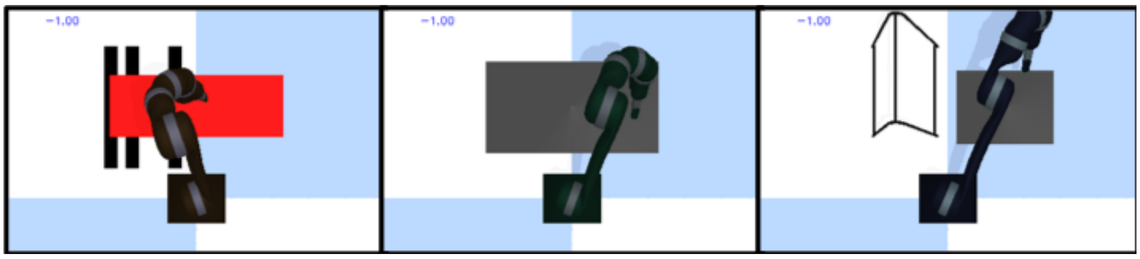


Figure 4.12: The level of randomness of our proposed environments. From left: different initial positions of the tape in Tape environment, space where the cloth can be spawned in Folder environment and space where cloth can be spawned in hanger environment (top views).

tion) [21]. A classic bath towel is laid in front of the robot and folded to be approximately 0.6m long and 0.2m wide (this roughly corresponds to the sizes in original paper). There is also a black tape randomly spawned at one of the 3 possible positions - 5/8ths of the towel length, 7/8ths or at the very end of the towel. The tape serves as a mark up to which the towel should be folded. The robot therefore needs to grasp the towel on the right end and fold it over up to the tape mark. The agent is successful if both corners originally on the right end of the towel are within a small distance (6cm) from the tape and the towel is no longer grasped by gripper.

- **Draping on a hanger** - the final task should to some extent resemble the task of doing the laundry. An episode starts with a small rectangular towel spawned randomly to the left of the robot and a small triangular hanger spawned to the right of the robot in a fixed position. The hanger has infinite mass, so it it is impossible to knock it over. Its position is randomized by a few centimetres for the purposes of domain randomisation. The robot needs to grasp the cloth and drape it over the hanger so it stays on it and does not fall to the ground. The agent is successful if all corners of the cloth are above ground for more than 15 simulation steps after the gripper released it (this ensures that the cloth needs to be on the hanger, otherwise it would fall).

All environments have the following low dimensional state: gripper position, gripper finger angles, cloth corner positions (12 scalars), arm goal position, arm goal gripper fingers, arm velocities and whether the arm is grasping the cloth. Moreover, the Tape environment and Hanger environments also contain a single additional scalar representing the position of the tape or the hanger along y axis.

The environments implement sparse reward (3 on success and -1 on each other step) and positive reward (100 on success and 0 otherwise). The reward magnitudes were chosen experimentally by observing the learning performance. We did not implement dense rewards at all because it would be difficult for the hanger task. Moreover, we would need to hand engineer the reward function to explicitly tell the robot how to execute the manipulation, which defeats the purpose of end-to-end learning. Finally, we seen from our experiments with rigid object environments(section 4.3.2) that sparse rewards are sufficient when the learning is seeded with demonstrations (more on this in sections 5.2 and 5.3).

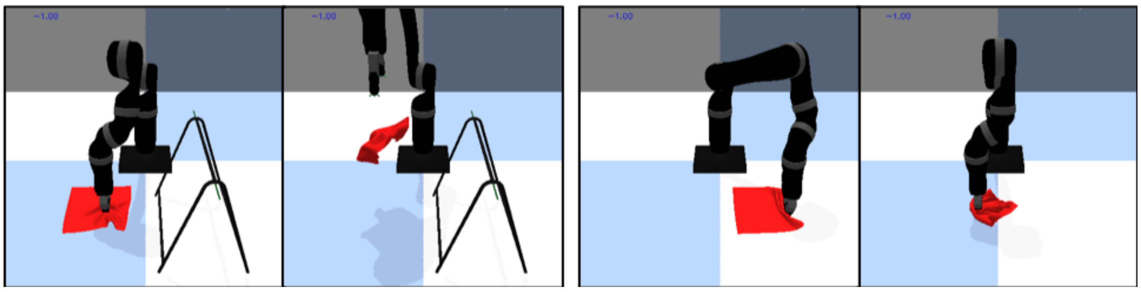


Figure 4.13: Two greedy behaviours the agent has learned in presence of ill defined reward. On the left, the agent grasps and throws the cloth in the air - it takes long enough for the cloth to fall so the reward was awarded. On the right, the agent learned to crumple the cloth so two diagonal corners become randomly aligned and agent receives the reward.

The observation is either low dimensional (low dimensional state as described above), high dimensional (84, 84, 3) RGB image or (4, 84, 84, 3) sequence of consecutive RGB images.

4.3.4 On the resourcefulness of agents in presence of ill defined rewards

The task of an agent in RL setting is to learn a behaviour that leads to the largest possible reward. The environment definitions we have outlined above are actually a result of a very long experimentation that finally yielded a desired behaviours. In this short section, we would like to describe some of the situations we have encountered when designing the environments and how we have resolved them.

One of the first successes of the trained agent, which we have celebrated in early March, was solving the Reacher environment in the presence of dense reward. The task was comparatively simple - the agent simply had to reach the blue sphere and it was receiving immediate feedback on its performance (the dense reward was growing as the gripper approached the blue sphere). However, the observation we have used at the time also included the depth render of the environment. We have discussed this with Stephen James and he expressed concern that depth cameras in real world are extremely noisy (the Kinova robotic arm has strong specular reflections and its dark colour makes it hard to register structured light necessary for depth vision). The noise in depth channel would be likely extremely confusing for the agent after the domain transfer, so we have decided to only use RGB observation. However, we found that reaching task becomes almost impossible in absence of depth channel. The only indication of target position is the size of the ball but the given the low resolution observations (84x84 pixels), the variation in size if quite small. The agent simply had no way of knowing where the sphere is along a specific ray from the camera.

After removing the depth channel from the observation, we were getting significantly worse average rewards (which, as we understand in hindsight, was caused by problem we explained in previous paragraph). However, after a couple of hours, the rewards started growing again and we optimistically thought that the agent has learned the reaching behaviour. When we visualised the final policy, we saw that this was not the case and the agent simply did a spinning motion at full speed, biased towards the area where the sphere was. By doing this quick spinning motion, it was guaranteed to hit the ball regardless of its actual position. After discovering how missing depth channel affects the learned policy, we have decided to instead iterate on the Pushing task - having a guarantee that both cube and target are in a single plane on the table allowed the agent to fully understand the state even without depth perception. In other words, there was a one to one mapping between object positions and camera rays in pushing tasks, which was not the case in reaching.

We had many more issues with cloth environments as those were first of their kind. When we have implemented diagonal folding environment for the first time, the only condition for success was to align the diagonal corners. We were excited to see that the agent almost immediately started getting very high rewards based on the learning curves. However, visualizing the learned policy revealed that the agent simply learned to crumple the cloth either by simply pushing it to create a small pile or by lifting it and letting it fall. The crumpled cloth was likely to have two diagonal corners aligned and the agent thus received the reward. We have resolved this by introducing a condition that adjacent cloth corners need to stay at 66% of their original distance from each other.

Similar situation occurred when we were testing the hanger environment. We initially specified that the cloth corners only need to stay 5 or more centimetres above ground for 10 simulation steps after the cloth is released from the gripper (we have experimentally verified that this is enough for the cloth to fall to the ground if hanging by the corner from the highest point the arm can reach). However, the agent learned to grasp the middle of the cloth and lift it quickly, so it continues going up for a brief moment after it was released. The agent learned to throw the cloth in the air so it takes more than 10 simulation steps for it to fall and hence the agent got the reward. We have simply solved this by increasing the number of time steps the cloth needs to stay above ground before we deem the agent successful.

4.3.5 Summary

Overall, we have implemented 3 different kinds of environments to help us with the development of the RL algorithm. The easiest environments to solve are 2D environments `Cartpole_v0` and `Pendulum_v0`. However, as those environments do not support pixels observation, we reimplemented them with OpenCV renders that can be passed to the agent as pixel observations.

More advanced environments require the algorithm to actually complete a robotic manipulation task by controlling a robotic arm. We were inspired by OpenAI gym robotics environments [20] based around Fetch [80] platform and simulated in Mujoco [79] and we re-implemented them in Pybullet [17] using the hardware available in the lab. In total, we have created 4 different environments for rigid object manipulation: Reacher, Grabber, Pusher and Slider.

We are not aware of any open-sourced environments for robotic manipulation of deformable objects and therefore we had to implement our own. More specifically, we have created environments for 3 different tasks: folding a square piece of cloth diagonally, folding a large towel up to a mark indicated by a black tape and draping a small rectangular towel over a small hanger.

We have encountered various challenges during the process of implementing the environments, mainly to do with importing the hardware model into the simulator, creating reasonable approximation of grasping in real world and specifying the reward criteria such that the agent is not able to cheat.

Chapter 5

Reinforcement learning

This chapter starts by describing our choices for programming languages, machine learning libraries and hardware. We will then discuss our initial experimentation with learning algorithms and justify our decision to use DDPG [22]. However, the central part of this chapter will focus on the implementation of various improvements to DDPG that ultimately led to an algorithm capable of solving the environments we have described in the previous chapter.

5.1 Research tools

The selection of correct tools is an essential prerequisite for every research project. We will briefly justify our choices and also outline any revisions we had to do over the course of the project.

5.1.1 Programming languages and tools

We have decided to write as much code as possible in Python programming language [81] for a variety of reasons:

- It is de-facto **industry standard** for machine learning projects. As a result, it has a massive ecosystem of libraries which is still growing (new libraries being build on top of older ones).
- It is **supported by our simulator** of choice. Bullet currently only supports Python, C and C++. However, we have decided to avoid C/C++ so we can do much faster prototyping and so we can (mostly) avoid dealing with memory management.
- There is a lot of **RL educational materials** written in Python. Most code examples, blog posts and reference implementations are written in Python.

We used standard Python tooling throughout the project. We installed all necessary packages using Pip into virtualenv to avoid conflicts between Python installations. We used Sublime text as our editor of choice as we did not feel that it is necessary to use an IDE for Python code. We used Vim whenever working with remote machines. As this is a research project, we did not feel it was necessary to set up any continuous integration pipelines, and we used software testing only on a small number of occasions.

Our codebase was version controlled using Git and stored on github.com. Our work required us to fork multiple repositories (Bullet physics [17], OpenAI baselines [77], OpenAI gym [19], KerasRL [82]) and we therefore extensively used git submodules to embed forked repositories in our main repository and to version control them (e.g. we knew which commit of Bullet works well with a specific commit of the main repository). We used a simple git work-flow with a master branch holding mostly stable code and feature branches used to track implementation of specific features. When necessary, we also used a "develop" branch to store new, untested code before a vital checkpoint (e.g. before a supervisor demo) to make sure the code on the master is in working condition.

In the second half of the project, we started using various headless machines with GPU acceleration for training instead of the personal computer, and we were therefore often interacting with machines via ssh. A tool that made this considerably easier is Mosh, which serves as a drop-in replacement of ssh. It, unlike ssh, does not disconnect on network change and therefore allows the users to continue working where they left off even after the commute home or more generally after any network outage/instability. We usually used it in tandem with tmux to enable terminal splitting and detachable sessions. We have scripted the necessary commands so connecting and reestablishing session on a specific machine in DoC, Dyson Lab, Azure or AWS only required issuing one command in the local shell of our laptop.

5.1.2 Important automations

We found that we are spending a significant amount of time doing the same workflows over and over again which prompted us to develop a set of automation to reduce the number of repetitive tasks:

- **Evaluation visualisation uploader** - as explained in section 4.3.4, looking only at the evaluation rewards the agent is receiving might be often misleading. We, therefore, needed a way to visualise its actual behaviour after each training epoch. We initially rendered and saved a video from the simulator on the remote machine and we always scp-ed the video to our laptop when necessary. We later decided to write a module that automatically uploads the videos to Google drive, and sorted them to folders by the name of the run. Google automatically compresses the videos to save storage space, and it offers a web player to play the video quickly. As a bonus, we could check the progress of learning from a mobile device when we did not have access to a computer. We estimate that we have collected around 1 million videos that are currently stored on the drive.

- **Lightly loaded GPU search** - in the second part of the project, we felt a need to start using GPUs to accelerate learning (more on this in section 5.1.3). One of the repetitive tasks was searching for a lightly loaded GPU among "graphic" machines in DoC. We automated this by writing a small script that repeatedly checks the GPU usage and reports it in the terminal.

- **Hyperparameter search** - in ML, it is often necessary to try different combinations of hyper-parameters to achieve good results. We automated this task by using hyperopt [83] package to start the searches and we uploaded the parameters and results to Google drive sheet (serving as a free database). However, we never had enough time and computing power to execute a large scale search.

5.1.3 Learning hardware

We have only been using a personal **MacBook Pro Early 2015** (2.7 GHz Intel i5, 8GB RAM 1867MHz DDR3 RAM) for all experimentation in the first half of this project. We did not find the hardware limiting because we mostly worked on improving the simulation environment. We have started working on learning algorithms in late February and we continued using the personal MacBook Pro for early experiments with low dimensional 2D environments. However, once we started learning from pixels, we found two aspects of the hardware limiting - firstly, it became impossible to store the replay buffer in memory as it required approximately 16 GB of storage even in small resolution. Secondly, training the network without GPU acceleration was incredibly slow.

We have first attempted to do training on AWS, more specifically on GPU **g2.2xlarge instance**. The instance has 1 CUDA-enabled K520 GPU and 15 GB of memory (which was enough to store the slightly smaller replay buffer). The main problem was the instance cost of $0.65 per hour, which was quickly depleting our AWS educate allowance of $100. After running out credits, we started using **Microsoft Azure NC6** instance with roughly two times stronger NVidia K80 GPU and 56GB of RAM. This instance allowed us to run multiple simultaneous experiments comfortably and at $0.9 per hour, this option was better value for money. However, we only had $100 worth of credit which we were mostly saving for emergency situations.

Apart from cloud solutions, we have looked at the hardware provided by the department. CSG kindly provided us with instruction on setting up Tensorflow [84] + CUDA on **"graphic" machines**. Those machines are equipped with a range of NVidia GPUs with NVidia 1080 Ti being the strongest and NVidia 970Ti being the weakest. The machines also have a variable amount of RAM, from 16GB on most machines up to 64GB on three machines. The machines are available on "First come, First serve" basis and are heavily utilised. We could not control how much memory other users use, which tended to be problematic because Linux kernel implements an OOM killer daemon which kills the largest process when exhausted RAM and Swap threaten the stability of the kernel. As our training was always the largest process due to the large replay buffer, our jobs were often killed.

Last hardware resource we extensively used during the project were machines in Dyson lab. More specifically, we initially had access to `bigdaddy` (64 GB RAM, 4x NVidia 1080 Ti GPU) and later we were also given access to `basalt` (256GB RAM, 5X NVidia Titan and 2X NVidia 980 Ti GPU) which we shared with other PHD students. Having access to those machines improved our iteration speed and efficiency because we no longer had problems with training jobs being killed and we had dedicated access to some GPUs for training.

Overall, we used a large variety of hardware during this project depending on our needs and the availability. We are genuinely grateful to the members of Dyson lab for sharing a significant amount of their computing power for the purposes of this project.

5.1.4 Deep learning libraries

The growing interest in artificial intelligence and machine learning motivated the development of many competing deep learning frameworks. At the time of writing, Tensorflow [84] was the most popular framework holding the strongest community with Keras [85] being slightly less popular according to a deep learning framework survey by Zacharias et al. [86] Keras itself is, however, a wrapper around Tensorflow - it provides higher level API and allows for faster prototyping at the cost of some expressive power.

We started the project with some early experimentation using Keras, but we later settled on using Tensorflow. The main reason for those choices was the availability of reference implementations of deep RL algorithm in OpenAI Baselines [77] and Keras-RL [82]. We wanted to avoid implementing the algorithms from scratch because deep RL is known to be unstable and it is hard to attribute poor performance to either a software bug, wrong hyper-parameter choice or merely a bad random seed. Having a correct and well-tested implementation as a starting point significantly reduces the likelihood of a software bug, and it also creates a baseline against which we can compare our further experiments.

5.1.5 Summary

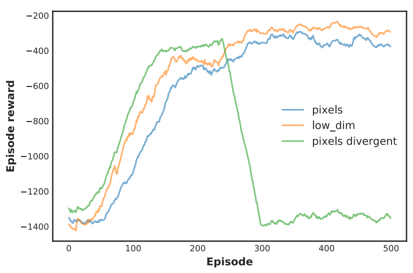
We have used Python programming language with standard Python tooling (pip, virtualenv etc.) for implementation of all RL components and we version controlled our codebase using Git. Our early experiments were implemented using Keras library, and we later moved on to use pure Tensorflow. We have extensively automated our work-flows to save time for actual research work. The need for computational power was increasing rapidly throughout the project implementation, and we responded to it by employing both cloud resources, DoC machines and Dyson lab machines.

5.2 First Experiments

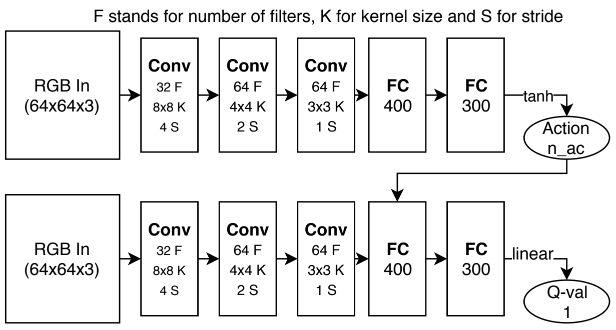
We have briefly experimented with multiple reference implementations of various RL algorithms before settling on a specific choice for this project.

5.2.1 DDPG - Keras

The first algorithm we have experimented with is DDPG [22]. It is a deep RL algorithm based on the actor-critic architecture. The actor is a neural network that tries to predict the optimal action in a given state from the partial observation it receives. The critic takes in observation and action from which it tries to predict



(a) Episode rewards on `pendulum_v0` with low dim observations and with pixels observations. Note the divergent behaviour occurring with bad hyper-parameters.



(b) Network architecture used for learning with pixels observations on pendulum environments. For 3D robot environments, we only increased the RGB resolution to 84x84.

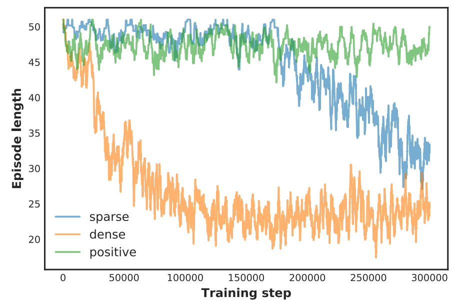
Figure 5.1: Episode rewards on pendulum environments and network architecture used.

the Q-value of that action in the given state. In other words, it predicts a measure of how good is the action given the state the agent is in. The critic is learning the Q-value by minimising the Bellman loss while the actor is merely optimising for the maximal output of the critic. We have covered DDPG more formally in Background chapter of this report (see section 3.4.2).

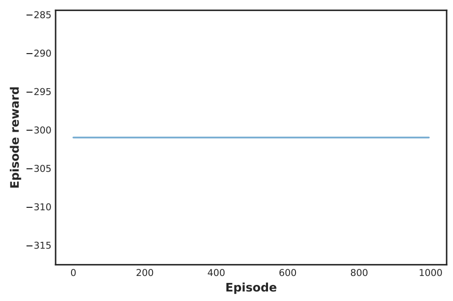
Our initial experiments with DDPG were based on reference implementation in Keras-RL [82]. We had to make only minimal adaptations to make the code work with our 2D environments and later with our 3D environments. The algorithm converged quickly for the low dimensional 2D pendulum. However, we had trouble making it converge for our pendulum environment with pixel observations. We experimented with multiple network architectures, and we found that it is necessary to have at least two fully connected layers after the convolutions to achieve good results. The network architecture we have used for learning from pixels in Keras experiments can be seen in Figure 5.1b.

We also found an interesting issue with the performance degrading if the agent is trained for more than 30000 timesteps. This behaviour seems to occur when critic network starts predicting extremely low Q-values (critic estimates started diverging). This is a common issue in Q-learning algorithms, as also noted by Mnih et al. [56]. We hypothesise that this might be caused by caused by a.) wrong setting of learning rate, b.) too fast propagation of weights to target networks, c.) too small replay buffer (changes of Q function cause large changes of policy which again causes changes in Q function) or d.) lack of regularisation on the critic, which then starts over-fitting. The behaviour went away with hyperparameter tuning (doubling replay buffer size to 1×10^6, decreasing target model update by 10x to 1×10^{-3}). Overall, learning from pixel observation achieved comparable performance to learning from low dimensional observation, as seen in Figure 5.1a.

We used the same algorithm to also experiment with Reacher environment. The learning unsurprisingly converged when using dense reward and low dimensional observation. It was, however, a straightforward task because the agent simply had



(a) Episode lengths on Reacher low dim environments (lower is better) with dense (distance to target), sparse (3 on success, -1 otherwise) and positive (100 on success, 0 otherwise) reward structures.



(b) Pusher never achieved success by random exploration with normal noise. Hence, the rewards are flat at -301 (-1 for each timestep in 301 steps long episode).

Figure 5.2: Learning plots from experiments with DDPG on 3D robot environments - reacher and pusher.

to learn to reduce a specific component of observation - relative gripper and target position. Moreover, we used dense reward, so the agent was immediately rewarded for decreasing the distance between the gripper and the target. Learning with sparse reward took considerably longer and showed only limited signs of progress, as seen in Figure 5.2a. Our final experiment was to try a multistage task with sparse rewards, such as Pushing. On figure 5.2b, we can see that the episode rewards are flat at minimum reward (-301) which means that the agent never reached a reward state and therefore could not learn the right behaviour. This is consistent with prior work by Vecerik et al. who report that pure DDPG cannot solve any of their four manipulation environments if the agent is awarded only sparse rewards. [7] We hypothesized that the exploration noise (a noise term added to the action at each timestep to make the agent explore different behaviors) is too low or not distributed correctly, so we tried normal noise with standard deviation of 0.2, OrnsteinUhlenbeck noise with $\theta = 0.15$ and uniform noise sampled from $[0.2, 0.2]$. OU noise seemed to be slightly more successful with two successes out of 300k timesteps, compared to 0 for normal noise and uniform noise. However, we did not experiment with multiple random seeds so we cannot show statistical significance of this result and we believe random effects might have caused it. Overall, we see that although DDPG can solve simple environments with dense rewards both learning from low dimensional observations and from pixels, it has trouble converging in the presence of sparse rewards.

5.2.2 DQN

As we previously mentioned, we had many difficulties when we were trying to make Keras implementation of DDPG work with pixel observations, and we had to experiment with multiple network architectures. There are not any reference implementations of DDPG build for environments with RGB observations available online, so we have decided to try an algorithm which achieved substantial successes in Atari environments. DQN uses only a single neural network to approximate the Q-value of all possible actions in a given state. This can be done in two ways - either by passing an action as network input and having Q-value as output (which would require a forward pass for each action to determine maximum Q-value) or by passing only the state as input and having a single output for each action (in this way, one forward pass through the network is sufficient to get Q-values of all actions and therefore get the maximum). The exploration is done by using ϵ-greedy policy, which takes action with highest Q-value with probability $(1 - \epsilon)$ or takes a random action with probability ϵ. We provide a more detailed description of DQN in background section 2.2.

In order to use DQN, we had to discretise the environments. For pendulum, we went with simple discretization with 5 discrete actions: (-1, -0.5, 0, 0.5, 1). For manipulation tasks in our 3D robot environments, this would lead to $5^4 = 625$ which would create a fairly large network. We instead went for a simpler discretisation scheme with only three values in each dimension (-0.5, 0, 0.5) leading to only 81 actions.

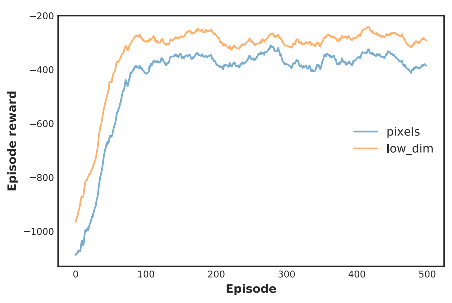
The results of learning on Pendulum environments can be seen on Figure 5.3a. Learning on both low dimensional and pixel observation converges, with pixel observation having slightly worse final performance. When compared to DDPG, DQN converges on fewer episodes than DDPG and achieves comparable final results.

The situation is however different with low dimensional Reacher environment, where DQN does not show any progress regardless of the reward structure (as seen in Figure 5.3b). We have visualised the learned policy, and the arm had fast erratic behaviour. We hypothesise that coarse discretisation of the environment does not allow the arm to move precisely enough the accomplish the task. We did not experiment with different discretisation schemes because we believed that cloth manipulation would require a very precise control which won't be achievable without an exponential increase in the number of actions. Moreover, we are worried that DQNs would show same limitations as DDPG when training with sparse reward. Overall, DQNs did show some promise on 2D environments, but we do not consider it a viable option because of the need to discretise high dimensional action space (4 for position control and 10 for velocity control).

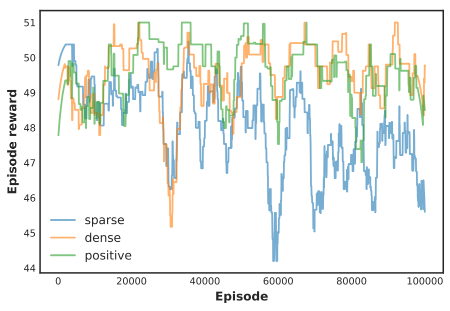
5.2.3 Other experiments

We have also experimented with Hindsight Experience Replay [63], TRPO [65] and PPO [64]. Each of those algorithms had some drawback that discouraged us from further work with it, and therefore we do not propose a more formal evaluation here.

Hindsight Experience Replay (HER) tries to address the problem of sample efficiency in sparse reward settings. In most RL algorithms, there is only very little an agent can learn from failed episodes (episodes with no positive reward). HER proposes to alter the goal of the agent when replaying a previous experience, so it achieves the goal. Let's take a pushing task as an example. Assume an agent randomly hits the cube and moves it somewhere, but not to the target position. Normally, there would be no reward for the behaviour, so the agent does not learn anything. With HER, the episode is artificially made informative by moving the



(a) Pendulum with low dim and pixels observations. Both converge with pixels having slightly worse performance.



(b) Episode lengths on Reacher low dim environments (lower is better). All curves fluctuate between 45 and 50, which means no convergence.

Figure 5.3: Learning plots from experiments with DQN.

goal to the location where the cube landed in hindsight (during training, when the transition is sampled), so that the agent will be rewarded. More generally, we always change the goal of an episode to be the final state, so the agent learns how to achieve different goals. The drawback of this algorithm is a difficulty to change the goals "in hindsight" for some of our cloth manipulation tasks. For example, in Hanger task, the robot needs to drape the cloth over a hanger, and this is the only way it can achieve the reward. There is no straightforward way to make a final result of a failed episode to be the goal for training because there is only one very specific goal in this task. We ended up not experimenting further with HER because of this fundamental limitation.

TRPO and PPO are both from the same family of algorithms and PPO is essentially a simplified version of TRPO. Both are said to be very robust to hyper-parameter selection which made them appealing to us. We tested the two algorithms on our pendulum environments, and we achieved comparable results to our experiments with DDPG and DQN. Both algorithms also worked well with dense reward Reacher task. However, we were not able to achieve convergence on sparse reward Pushing, most likely due to the agent failing to find high-reward states often enough. We ceased to experiment with those two algorithms because we found considerably less research on how to improve them as compared to DDPG. As we did not have a good road-map of things we could try to make the algorithms work with our more complex tasks, we have decided to continue our work on DDPG.

5.3 DDPG and implemented improvements

As we have seen in the last section, DDPG has shown promising results, but it did not work well when using sparse reward structure because exploration using random noise did not hit the successful states often enough. We, however, wanted to avoid using dense rewards because it would mean going down the path of reward shaping. This is difficult to do in practice on many complex manipulation tasks, and it can encourage the suboptimal behaviour. For example, one way to define dense reward in pushing task would be:

$$r = \lambda_1 * d_{grip/cube} + \lambda_2 * d_{cube/target}$$

where λ_1 and λ_2 are some positive parameters, $d_{grip/cube}$ is the Euclidean distance from gripper to the cube and $d_{cube/target}$ is the Euclidean distance between the cube and the target. The reward function will, therefore, encourage the robot first to reach the cube and then attempt to push it towards the target. However, we have noticed that an agent might learn to try to push the cube from the top and drag it to the target, which is much worse than pushing the cube from the back (it is unsafe for the robot fingers that can break and it is also more likely to loose the cube during the pushing motion). We believe that the agent learned this behaviour because the reward function discouraged it from going a bit further from the cube to get behind it. We could have solved this by introducing a more complicated reward function, but that would defeat the purpose of RL where the robot should explore the best behaviour and learn from its successes and mistakes. It is therefore essential to make sure DDPG can work well with sparse rewards. In the following subsections, we will describe various improvements to DDPG we have implemented to get the final learning algorithm used in this project. We defer sharing the experimental results and ablation studies of this algorithm to the Evaluation (Chapter 7).

5.3.1 Basic DDPG implementation

We initially considered using the DDPG implementation in keras-rl [82] that we have evaluated in the previous section. However, the library is written with multiple layers of abstractions that ensure that different learning algorithms expose almost the same API calls to the user. However, we were aiming to do in-depth work only on DDPG, so we thought that this design decision introduced a lot of unnecessary complexity for out project. We have therefore started looking for other reference implementation we could use as a basis for the future work.

One such implementation is available in OpenAI baselines repository [77]. At the time of writing, it was the most starred repository with RL algorithm implementations on GitHub which gave us a lot of confidence in implementation correctness. The code was using TensorFlow [84] instead of Keras and it was well-structured into multiple components:

- **main.py** - entry point responsible for argument parsing, environment initialisation and starting the training.
- **training.py** - module responsible for training the agent. It collects transitions in the environments according to current policy, calls training functions of the agent and evaluates the performance after each epoch.
- **memory.py** - implements the replay buffer storing the transitions collected in the environment.
- **models.py** - the code describing the actor and critic network architectures.
- **ddpg.py** - the code implementing the learning algorithm.

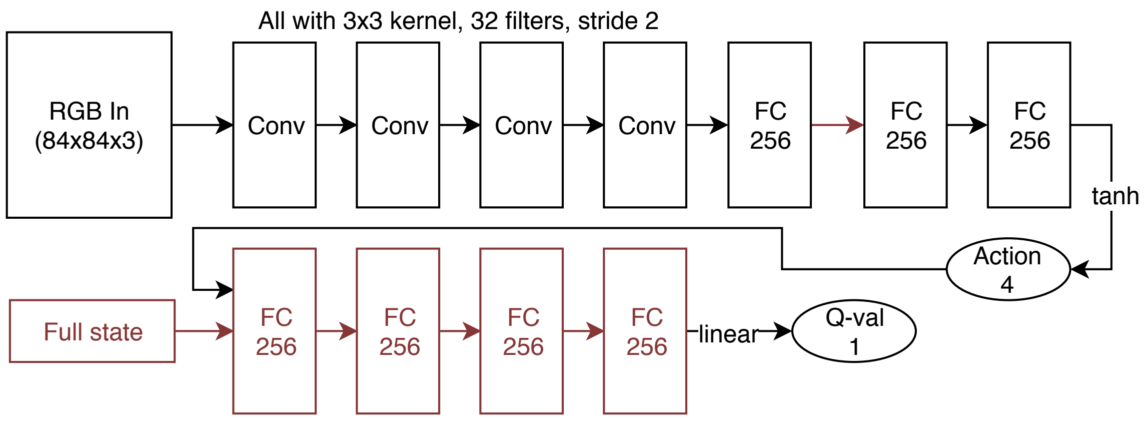


Figure 5.4: Network architecture with asymmetric actor and critic (most important change is highlighted in red).

- **noise.py** - implements the functions to generate exploration noise (Ornstein-Uhlenbeck noise and parameter perturbation noise).

We have decided to mostly retain this structure for the remainder of the project. We benchmarked the Keras-RL and Baselines implementations on pendulum pixels environments, and we did not see significant differences in performance of the trained agents or in the speed of learning. We have therefore decided to use the OpenAI baselines implementation.

5.3.2 Asymmetric actor-critic

The first improvement we implemented was using asymmetric inputs for the actor and the critic, as proposed by Pinto et al. [15]. When we are training the agent in simulation, we have access to the full state of the environment (positions and velocities of each object) which is not the case in the real world, where we only have a partial observation (e.g. RGB images from the camera). The idea of using asymmetric-actor-critic is to leverage the low dimensional full state during training of the agent but only use partial observation at test time (possibly in the real world). The critic learns to estimate the Q-function much faster because it is presented with complete information about the environment and it does not need to train the convolutional network. We show the network architecture at this step in Figure 5.4. We have experimentally found that slightly more fully connected layers seem to work better for our use case, which we have already reflected in the figure.

The ablation studies provided in the paper showed a significant improvement of using asymmetric critic over pure DDPG. The symmetric DDPG agent never converged on their set of tasks when learning from pixels, while the one using asymmetric DDPG did. The authors also show significant improvements in the learning stability. We have also experimentally verified the improvement both in the speed of learning as there are significantly fewer parameters that need to be optimised and in terms of the agent performance.

5.3.3 N-step returns

N-step returns are used to facilitate faster propagation of Q-function values through Bellman equation and they, therefore, increase the learning speed. In classic DDPG, the critic is trying to minimise Bellman loss defined as:

$$L_{1-step} = (Q(s_t, a_t) - r_t - Q^*(s_{t+1}, \pi^*(o_{t+1})))^2 \tag{5.1}$$

In other words, the predicted Q-value of taking action a_t in the current state s_t should be as close as possible to the awarded reward plus Q-value of taking action according to the current target actor $\pi^*(o_{t+1})$ in the next state s_{t+1}. Q^* is the target critic (second critic network updated at a slower pace to stabilise learning). N-step Bellman loss is similar:

$$L_{N-step} = (Q(s_t, a_t) - \sum_{i=0}^{N} \gamma^i r_{t+i} - \gamma^N Q^*(s_{t+N}, \pi^*(o_{t+N})))^2 \tag{5.2}$$

The main difference is that we are looking N steps into the future, so we would like to have Q-value of current action-state pair $Q(a_t, s_t)$ as close as possible to the discounted (by discount factor γ) Q-value of the state we reach after N-steps plus the discounted rewards we collect on the way there. N-Step returns have been empirically shown to improve performance on locomotion tasks significantly [62]. The ablation studies in their publication reveal that N-step returns are particularly useful when used in sparse reward environment. In sparse reward setting, the reward signal is distributed very unevenly - only a couple of states award reward on entry. It is therefore important to quickly propagate where those states are.

However, even though N-step returns have been empirically shown to improve performance, it is essential to select a correct value of N. With 1-step update, the training algorithm always considers a single transition at a time and therefore it does not matter which policy generated the transition. With N-step updates, the algorithm considers a sequence of N-transitions which were generated according to a specific policy. Therefore early training iterations will have different N-step returns than later iterations (which is not only due to environment randomness). As DDPG is an off-policy algorithm, this breaks some fundamental assumptions in its design and becomes detrimental as a value of N increases. We have found $N = 5$ and $N = 10$ to work reasonably well, but increasing it further made performance worse.

In terms of the implementation, we add L_{N-step} to the critic loss equation, so the critic loss becomes:

$$L_{critic} = \lambda_{N-step} L_{N-step} + \lambda_{1-step} L_{1-step} + \lambda_2 \tag{5.3}$$

where λ_{N-step} and λ_{1-step} are hyper-parameters that weight the contribution of two Bellman losses and λ_2 is the l2-regularization term.

5.3.4 Prioritized replay buffer

Replay buffer is the component that stores the transitions generated by the agent interacting with the environment, and hence it lets the agent reuse experiences from the past during training. Each transition contains information about the original state (low dimensional representations and pixel observation), action taken, reward awarded and information about the new state (the result of the action on the environment). During training, DDPG takes transitions randomly in mini-batches from the buffer, so it always sees a large temporally uncorrelated sample of previous states. This stabilises learning and decreases the likelihood of oscillations (agent "overwriting" old learned behaviours with new ones).

Some of those transitions are naturally more informative than the others. For example, imagine an agent acting in a sparse reward setting that has never been given any reward. Suddenly, it randomly executes the right action in the right state (thanks to the exploration noise), and it stores a transition that contains a reward. This transition is arguably more valuable than the other transitions in the buffer because it is the only one that describes the desired behaviour of the agent. However, suppose then that over time, the agent learns to always do the rewarded action in that specific state. This transition will, therefore, become less valuable because it is not bringing any new information to the agent. In general, transitions that are likely to teach the agent something new are the most valuable. We can measure this by looking at Bellman losses as described in 5.3.3 (they are also called temporal difference errors). Large TD errors mean that critic did not understand well the action-state pair in the current transition, so it miss-predicted the Q-value. The transition is therefore vital for further learning.

This idea was explored in a paper by Schaul et al. [87]. They propose a proportional sampling strategy, where transition i, which had TD error δ_i when last used for training, is sampled with probability:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5.4}$$

where α is a hyper-parameter controlling for the strength of prioritization (when $\alpha = 0$ there is no prioritization) and $p_i = |\delta_i| + \epsilon$. ϵ is a small positive constant that ensures a transition can be sampled even if it had zero TD error at some point during training. A new transition is always given maximal priority (currently highest priority value in the buffer) because we do not yet have any TD error information.

However, having a different probability for each transition introduces a bias that needs to be corrected. The authors of the paper cope with the problem by introducing importance weights:

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta$$

where β is a hyper-parameter and N is the number of transitions in the buffer. When training, we multiply the critic loss coming from a specific transition by its importance weight. If $\beta = 1$, multiplying the critic loss by weight fully offsets the increased probability of sampling the transition. In practice, we do not need to use $\beta = 1$, and we either use a lower value throughout the training or start with a lower value and anneal it towards one during training.

In terms of implementation, a sampling method linear in size of the replay buffer would not be sufficient because selecting transition will dominate the CPU usage of the algorithm. We instead implement an alternative method based on `SumSegmentTree` data-structure implemented in OpenAI baselines [77]. In our case, the tree stores transition probability values. We can then uniformly sample a float between 0 and 1 and query the segment tree to return a maximum index such that the sum of elements up to and not including the index is less than or equal to the number we have sampled. We are effectively searching the first element whose corresponding value in cumulative distribution is larger than a uniformly sampled value between 0 and 1. Segment tree allows doing this operation in $O(logN)$ time. Setting a value in a tree is also $O(logN)$ time which makes the data structure ideal for our mixed insert/query workload. However, even when we use algorithmically efficient implementation, the benchmarking revealed that roughly one-fourth of the execution time is spent sampling. We believe this could be improved if we implemented the memory module in C as a Python extension to get full control of the underlying data-structures and to reduce the overhead of interpreted code.

5.3.5 Demonstrations

As we have shown in Figure 5.2, DDPG has problems to converge on complicated tasks with sparse rewards because it is statistically very unlikely it will achieve a correct behaviour by random exploration. One way to overcome this issue is to seed learning with a small number (in our case 20) of demonstrations to show the agent what sort of behaviours lead to the rewards. This idea was explored by Vecerik et al. [7]

Generating demonstrations

One of the obvious downsides of using demonstrations is that we first need to create them. There are two main approaches to collecting demonstrations:

- **Real demonstrations** - involves a real robot which can be either a.) kinesthetically force controlled [7], b.) controlled directly by joystick/keyboard or c.) controlled more intuitively by a VR controler [88]. In this case, the demonstrator manually controls the robot to achieve the task. The main advantage is that it does not require any task-specific programming and it easily scales to a large number of different tasks. It is easy to imagine that in future it will be enough to show the robot the correct behaviour to achieve the task, it will learn to generalise it in simulation and then execute it in the real world. The drawback of this method is that we would need to construct a pipeline to control the robot, process the data and create a demonstration file. Moreover, we wanted to do experiments with changing environments and numbers of required demonstrations, which would mean we would need manually recollect the observation after every such change. The demonstration collected on real robot is translated to simulation and stored in replay buffer to be used in training.

- **Programmatical** - another option is to create a program that controls the robot to achieve the task in simulation and record the transitions. This is

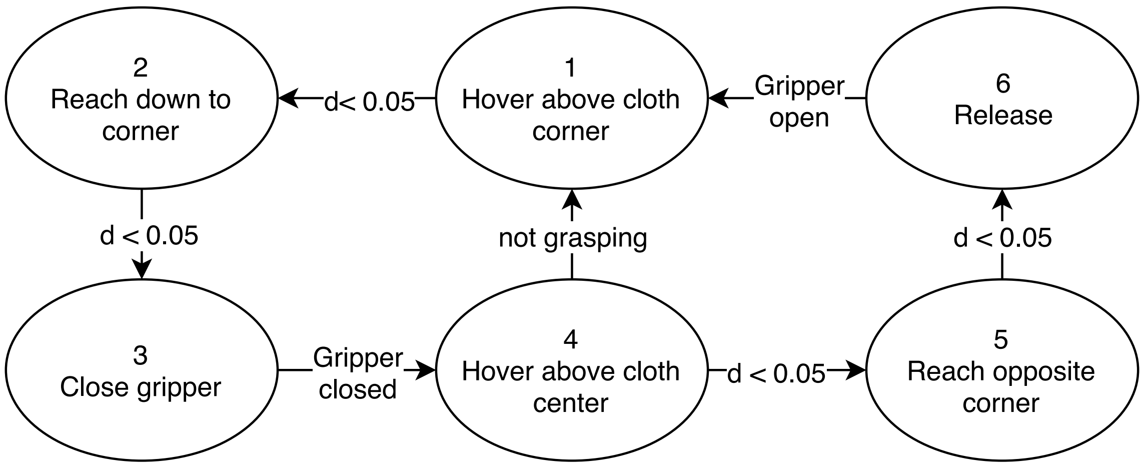


Figure 5.5: Example of demonstration state machine for diagonal folding task.

feasible because we can access the full state of the environment in simulation and therefore it is reasonably straightforward to estimate (possibly suboptimal) trajectory for the robot in the majority of tasks. This method does not require real robot at all.

In our case, we have decided to go for programmatical approach because it allows for fast iteration on environment specifications. One could argue that there is no point in using RL for those tasks if we can simply write code that solves the environment to collect demonstrations. This is not true for a couple of reasons:

- **Demonstration code uses full state** - one of the main reasons it is reasonably straightforward to write demonstration code is that we have access to the full state, including the exact positions of the gripper, cloth corners, tape, hanger etc. This allows us to write things such as: reach a corner, grasp it, hover over cloth centre point, reach down to the opposite corner, release. We would need to create a complex vision pipeline if we were to do the same thing only from pixel input (as the agent does at test time).
- **Demonstrations can be suboptimal** - we do not necessarily need to provide optimal demonstrations because their purpose is only to seed the learning. Once the agent learns the demonstration behaviour (or something close), nothing is stopping the agent from improving on it.
- **Demonstrations can fail** - sometimes it is difficult to achieve 100% success rate with demonstration code. However, when we are collecting demonstrations, we can throw away those that failed. In practice, we throw away approximately 25% of demonstrations, and this number was considerably larger in the past.

We can, therefore, see that the role of RL is still very important even when it is seeded with demonstrations. It can improve on the success rate, find more suitable trajectories and most importantly, learn to work with pixel observation.

We collect the demonstrations programmatically by examining the full state of the environment and implementing a finite state machine. In each state, the demonstration code executes an action (e.g. move toward cloth corner), and it can advance to the next state if it fulfils an advancement condition (e.g. the gripper is within 5 cm from the cloth corner). The demonstration code rarely handles errors explicitly, and it instead repeats the whole sequence of actions until it succeeds or the environment time limit is up (we wanted to avoid over-engineering the demonstration code because its purpose is only to seed the learning). We show our state machine for the diagonal folding task in Figure 5.5. The system stays in a given state unless the leaving condition is met. All states apart from 3 and 6 are implemented by sending action proportional to the vector between gripper position and goal. 3 and 6 close and open the gripper respectively. We store all transitions so we can use them later to seed learning (more on this in later subsections).

On the difference between demonstrations and reward shaping

We have presented a couple of arguments against reward shaping at the beginning of section 5.3 and we gave a practical example from our observations how shaping reward can lead to suboptimal behaviour. However, we claim in this section that engineering a task solution and showing it to the agent is the necessary and right thing to do. Even though both demonstration and shaped reward accomplish the same end goal to make the agent visit the right areas of the search space and ultimately learn the task, they have one fundamental difference.

Shaping reward constraints the agent to the behaviour which is described by the reward shape. Any deviation from the trajectory encouraged by this reward is immediately penalised, even though it might lead to better end outcome (consider the example with pushing from the introduction to section 5.3). The demonstration shows the agent how to achieve the reward but it does not discourage further exploration, so even though the demonstrated behaviour is suboptimal, it can learn the more optimal behaviour through more training episodes. In conclusion, both demonstration and shaping reward accelerate learning by leading the agent towards the right behaviours, but only suboptimal shaping reward function puts an upper bound on its final performance.

Prioritized replay buffer and demonstrations

After we successfully collect some demonstration transitions, we can store them in the replay buffer, and the agent will randomly sample them and use them for training (as proposed in the DDPGfD algorithm by Vecerik et al. [7]). DDPGfD also makes transitions more likely to be sampled by adding a small constant to the priority of demo transition. However, we found that tuning this constant is very difficult because TD errors tend to vary by multiple orders of magnitude between runs and also during training. We, therefore, suggest using a relative measure instead and increase the probability of sampling demo transitions by some fixed percentage of current maximal priority in the minibatch. More specifically, the priority will become:

$$p_i = |\delta_i| + \epsilon + \epsilon_d \max_{k \in minibatch} p_k \tag{5.5}$$

The sampling probability can then be computed from equation 5.4. ϵ_d is a hyper-parameter that controls the ratio between demonstration transitions and normal transitions. It would be an option to anneal this parameter down to 0 as learning progresses but we did not find it necessary as the TD error of demonstrations goes down and therefore their sampling becomes less likely even without this intervention. In practice, we use $\epsilon_d = 0.2$. DDPGfD also suggests adding a term proportional to the square of actor loss to the priority term, which we followed.

Finally, DDPGfD starts the training with some number of pre-training steps. This means that we sample a few hundred mini-batches from the buffer and train the networks before starting collecting new transitions. This behaviour leads to an agent that can immediately have good attempts at completing the task. However, too much pretraining can lead the agent to a local minimum that is difficult to escape.

Behavioural cloning

The demonstrations in DDPGfD algorithm do not get any special treatment when computing network losses. They are simply more likely to get sampled. Nair et al. [89] propose a method that allows the agent to leverage demonstrations more effectively by directly incorporating them to the actor loss computation. They propose adding another loss term, $L_{BC} = (\pi(o_i|\theta_\pi) - a_i)^2$, where π is the actor, θ_π are the actor parameters, o_i is the observation in transition and a_i is the action stored in transition. This loss term is called behavioural cloning loss. We add it to the loss only when an actor is trained on a sampled demonstration transition. It encourages the actor to compute the same action $\pi(o_i|\theta_\pi)$ as was done in demonstration transition a_i.

However, the authors point out that simply adding this loss each time would prevent the agent from improving on the demonstration behaviour because it will always just try to imitate the action from the demonstration. In order to overcome this limitation, they suggest only applying L_{BC} if the action proposed by the actor has worse Q-value than the demonstrations action (stored in transition tuple). They call this method Q-filter. The behavioural cloning loss then becomes:

$$L_{BC} = \begin{cases} (\pi(o_i|\theta_\pi) - a_i)^2, & \text{if } Q(s_i, a_i|\theta_Q) > Q(s_i, \pi(o_i|\theta_\pi)|\theta_Q) \\ 0 & \text{otherwise} \end{cases}$$

where Q is the critic, θ_Q are its parameters, and s_i is the state in transition tuple. Q-filter guarantees that the actor is imitating the demonstration behaviour only when the action it computes is worse (as judged by the critic). If it is better, the loss term is 0 and no imitation happens.

Reset to demonstration

One thing we did not address so far is the low probability of the agent randomly hitting the goal state in a complex manipulation task. Even with demonstrations to seed the learning, it is initially still tough for the agent to complete the task end to end to create its successful episodes. A method proposed by Nair et al. [89]

is to sometimes reset the environment to a state which was encountered during a demonstration, so the agent could start the experience collection closer to the goal.

For example, in the diagonal folding task, the robot always starts in some initial joint configurations with the cloth laying on the table. To achieve the reward, the robot would need to successfully reach it, grasp it, drape it over itself and release it at the right time. This is arguably a challenging sequence of subtasks. However, if we reset it to some state encountered in demo episode, we can significantly shorten this sequence. For example, if we reset it to a state where it is already grasping a corner of the cloth, it just needs to drape it over and release it which is more straightforward. Consequently, the agent will be able to start collecting own successful experiences earlier on in training and it will have the much more diverse set of transitions leading to reward.

Whenever an agent finishes a training episode, we reset the environment to a random demo state with probability `reset_to_demo_rate`. We start with high probability (0.9) and quickly anneal to low probability (0.1). Any demo state can be sampled apart from the last five steps before the demo episode termination. This is to avoid the overhead of many environment resets and to make sure agent needs to generate some of its transitions before the reward is given.

Implementation-wise, we store the state of simulation after each demonstration step in a file with a specific UUID (generated at the start of training run), and we reload it when we that specific state is sampled for reset. We also need to re-generate the constraints between soft bodies and rigid bodies which are not saved.

Summary

Adding demonstrations is one of the most complex improvements to DDPG we have used. We generate the demonstrations programmatically based on the full-state of the environments and store them in a replay buffer with increased priority. The actor also tries to imitate them whenever its actions are worse compared to demonstration action. Finally, they help the agent to reach reward states early in training by randomly resetting the environment to demonstration state where the agent is closer to getting the reward.

5.3.6 Distributed DDPG

The original version of DDPG is alternating periods of collecting new transitions and periods of training. According to our profiling results, it spent roughly the same amount of time on both tasks (in case of cloth environments, for others it spent more time in training as simulation was significantly cheaper). There is, however, no reason why transition collection and training cannot be happening in parallel. In fact, Barth-Maron et al. implemented DDPG with multiple simulator instances collecting transitions independently in parallel with training. [62]

We have also separated the acquisition of new experiences into a separate thread, so it is happening in parallel to training. We needed to synchronise the replay buffer data structure, so there are no concurrent reads (by the training thread) and writes (by the experience collector thread). We can also spawn multiple threads to collect experiences in parallel, but we did not find it necessary in practice. Having one thread doing training and one thread collecting transitions ensures roughly the same ratio of the number of new experiences to the number of training steps as in original sequential DDPG. Spawning multiple simulators will consume more RAM, which is the resource we are trying conserve to run multiple experiments on the same machine.

5.3.7 Auxiliary predictions

Sometimes it is difficult for the agent to learn the most informative set of features it should predict from the high-dimensional RGB input. One way to help it would be to make the agent predict the low dimensional features of the environment by adding another network output that estimates some specific features we believe might be necessary for the task. For example, in the case of cloth manipulation, the useful features might be the position of cloth corners, or in the case of pushing environment, it might be the position of the cube. Auxiliary predictions were used by James et al. to help to transfer the behaviour learned in simulation over reality gap. [13]

Another advantage of having auxiliary predictions is the ability to better understand what is going on under the hood of the network. Specifically, we could print and visualise the predicted positions of cube and gripper corners to make sure the network understands how to translate the RGB input into the low dimensional state. This becomes even more important when we attempt the domain transfer.

Implementation-wise, we added loss terms to the actor for predictions of all auxiliary features - object configuration (this is either cloth corners position or cube position), target configuration (this is either target position for pushing, hanger position for hanging or tape y-position for folding up to tape) and gripper position. The loss terms were simply weighted square errors between the prediction and ground truth. The ground truth was already part of the low-dimensional state, so we did not need to do any further work with environments or replay buffer. The changes to the network are highlighted in Figure 5.6 in blue.

5.3.8 Combining low-dimensional and pixel actor input

We have noticed that it is difficult to predict the gripper position from the pure RGB input accurately. Without the depth perception, the agent can theoretically infer it only from the scale of the gripper (which varies minimally on low-resolution image) or from the overall position of the whole arm (extremely difficult). However, having an accurate position of the gripper is essential for successful manipulation of cloth because the agent needs to reach it and grasp it with a sub-centimetre accuracy (on the z-axis).

We mitigate this problem by providing the agent with additional pieces of information that are readily available at test time anyway - joint angles and gripper position (real and IK goal). Joint angles can be read from real robot sensors at test time, IK goal is available in the environment state and gripper position is computed by forward kinematics from joint angles (which is already implemented in the robot firmware). Those pieces of information are concatenated with a flattened output from last convolutional layer. The agent no longer needs to infer gripper po-

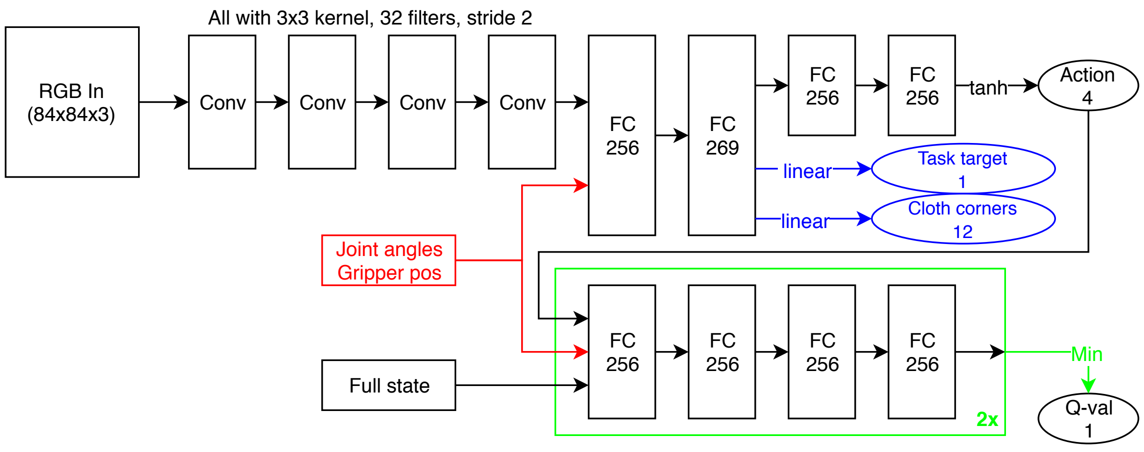


Figure 5.6: Final network architecture. Improvements from section 5.3.7 are highlighted in blue, section 5.3.8 in red and section 5.3.9 in green.

sition from scale and it can instead learn forward kinematics or just use the gripper position. The architectural change is highlighted on 5.6 in red.

5.3.9 Twin critic

The final improvement we have implemented was taken from the work by Fujimoto et al. [61]. The authors prove that actor-critic methods are prone to overestimating of the Q-values and also show empirical results from locomotion environments to support their claim. We have also empirically observed the overestimation of Q-values in our tasks.

They propose 3 different (not mutually exclusive) methods to fix the overestimation bias:

- **Twin critic** - maintain two critic networks to estimate Q-values and always use the minimum estimate. This method is similar to double-Q learning proposed by Van Hasselt et al. [90]. The critic with higher estimated Q-value servers as an upper bound for the full estimate, so this method will never overestimate Q-values more than classic DDPG.

- **Regularize target critic estimates with random noise** - two action/state pairs that are very close to each other should have similar Q-values. This behaviour is implicitly achieved by using a neural network as function approximator, but it is possible to require it explicitly by using a noisy version ϵ_r of target actor with standard deviation in bellman loss (updated form of equation 5.1):

$$L_{1-step} = (Q(s_t, a_t) - r_t - Q^*(s_{t+1}, \pi^*(o_{t+1}) + \mathcal{N}(0, \epsilon_r)))^2 \qquad (5.6)$$

  This regularisation would reduce the variance of Q-value estimates. We also clipped the noise normal distribution to avoid randomly getting very high/low values.

- **Delay target network updates** - the last improvement proposed by the authors is to delay target network updates and update them only once every n steps. Although this is similar to merely reducing τ (the parameter that controls how quickly target networks follow actor and critic), the authors argue that delaying the update by a couple of steps would improve the quality of update (do less good quality updates rather than many poor quality updates).

We have implemented all 3 improvements but we have found that regularizing target critic estimates with random noise significantly decreases performance so we stopped using it in final algorithm. For the other two improvements, we did not notice significant improvement but we left it in as a baseline for ablation studies. The architecture changes are highlighted on Figure 5.6 in green.

5.4 Summary

We have decided to use DDPG as the learning algorithm for this project. However, in its original state, it was not able to solve any of the more advanced manipulation tasks with sparse reward. We have justified the need to work with sparse rewards, and we have implemented a large number of improvements to DDPG, most of which are current state-of-the-art research in deep RL. The majority of the papers we have used as a resource for implementing those improvements were published in the last year (during the ongoing project work). The combination of those improvements allowed us to train the agent and achieve very good performance on all manipulation tasks. We will analyse the performance in more detail in evaluation (Chapter 7). In the next chapter, we will discuss our work on crossing the "reality gap".

For completeness, we also list the final equations we used. The critic loss was computed as follows (with passing action a_t from transition tuple):

$$L_{critic}(a) = \lambda_{nstep} L_{nstep}(a) w_i + \lambda_{1step} L_{1step}(a) w_i + \lambda_{L2} L^{C}_{reg}(\theta^Q),$$

$$L_{nstep}(a) = (Q(s_t, a) - \sum_{i=0}^{N} \gamma^i r_{t+i} - \gamma^N min_{i=1,2} Q^*_i(s_{t+N}, \pi^*(o_{t+N})))^2,$$

$$L_{1step}(a) = (Q(s_t, a) - r_t - min_{i=1,2} Q^*_i(s_{t+1}, \pi^*(o_{t+1})))^2$$

The actor loss was computed as follows:

$$L_{actor} = -L_{critic}(\pi(o_t)) + \lambda_{BC} L_{BC} + L_{aux}$$

$$L_{BC} = \begin{cases} (\pi(o_i) - a_i)^2, & \text{if } Q(s_i, a_i) > Q(s_i, \pi(o_i)) \text{ and } i \text{ is demonstration} \\ 0 & \text{otherwise .} \end{cases}$$

Chapter 6

Transferring policies learned in simulation to the real world

Transferring policies learned in simulation into the real world is a challenging task, but there are multiple significant benefits:

- **Fast acquisition of training data** - simulations usually run significantly faster than real hardware. Moreover, it is easy to spawn multiple simulators to collect experiences in parallel (see section 5.3.6).
- **Robot safety** - the robots tend to do very unsafe actions during the first few epochs of training before they learn the correct behaviours. This can often become damaging to the robot (e.g. breaking the gripper). Any damage in the simulation is not a concern because it can be simply reset.
- **Cheap data acquisition** - only the largest labs with strong funding can afford large robot farms necessary to collect data for still very sample inefficient RL algorithms. However, getting data in the simulation is significantly cheaper as it requires only compute power so almost all researchers can do it.

Considering those significant benefits, it is only natural that we also attempted to transfer the policy learned in simulation to the real world. In this chapter, we will start by showing the changes we made to the simulation environments to facilitate the transfer, we will then describe our setup for real-world evaluation, and we will finish by listing some of the challenges we encountered during transfer attempts and how we dealt with them.

6.1 Domain randomisation in simulation

The technique we have used to facilitate the transfer of the policy over the "reality gap" is called domain randomisation. The idea is to randomise as many simulation parameters as possible, so the learned policy becomes robust to some subtle changes of the environment properties. James et al. [13] have successfully used supervised learning to train an agent on artificial data generated in simulation (with strong randomisation) to accomplish Grabbing task in the real world, without ever seeing

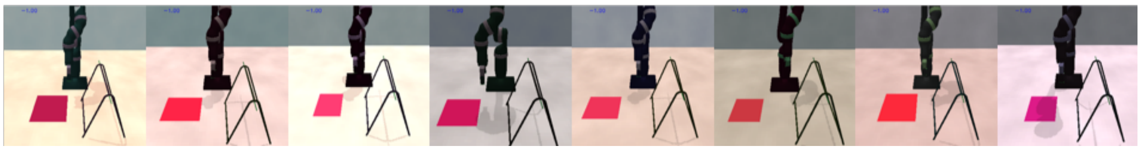


Figure 6.1: Examples of domain randomisation for hanger environment. Please note the changing textures, colours, light position, camera position and orientation, cloth size and position, hanger size and position, initial arm position and size of arm base. The values are sampled from either normal or uniform distributions around our best estimate of the value in the real world.

real data. Similarly, Tobin et al. [14] used domain randomisation to create a vision component of grasping algorithm (estimate object positions) that successfully transferred over the reality gap. Most relevant previous work is by Pinto et al. [15], who used domain randomisation exclusively for visual elements of simulation to transfer their policies learned with Asymmetric DDPG.

We used strong randomisation of many simulation parameters. More specifically, we randomized:

- **Textures** - the random textures were generated using Perlin noise [91], more specifically we were using the implementation by Casey Duncan [92]. We first sampled the base colour for the table and the wall from a normal distribution centred at the real colour of the objects (as measured by examining pixels from the real camera image). We clip the sampled colour, so each component stays in $[0, 255]$ range to create "base colour". We then generate a colour palette (map from an integer to colour) which maps all integers from 0 to 255 into colours centred at base colour and varying by up to 10 in each direction. The noise generator then creates a 2D noise array, which is mapped into colours using the palette to generate the final texture.

- **Camera parameters** - we are randomising a variety of camera parameters. Perhaps most importantly, we are sampling the camera position from a normal distribution centred at its real position (according to our noisy measurement) with a standard deviation of 1 cm. We then randomise the "look at" position (the point in space where the camera aims) and the field of view (FOV) from a normal distribution around the noisy measurement of FOV of the camera. We measured this using a simple printout of a protractor and marking last lines visible on the image.

- **Lighting and shadows** - we randomise the position of the light by setting it to a uniformly sampled coordinate above the scene ($z \in [70, 100], x \in [-20, -5] \cup [5, 20], x \in [-20, -5] \cup [5, 20]$). Note the absence of samples with the light directly above the scene, which we had to do because values close to 0 caused segmentation faults in the tinyrenderer shader. We were not able to identify the root cause of this behaviour. The hue of the light is sampled from a uniform distribution close to white, and all lighting coefficients (ambient, specular, diffuse) are sampled from uniform distributions.

- **Object randomisation** - we have randomised the colours and sizes of all objects present in all environments (cube, target, cloth, hanger, tape). All values were sampled from normal distributions.
- **Arm randomisation** - we randomise the spawn position of the arm in z-direction because, in the real world, the arm sits on a small metallic stand. We also randomise the initial joint positions from a normal distribution around the rest poses, and we randomise the colours of all components.
- **Cloth parameter randomisation** - we randomise the angular stiffness, linear stiffness, damping and mass of the cloth (from uniform distributions). We have already mentioned that we also randomise the size and colour of the cloth. Cloth position is of course also randomised as the part of the task, not only for the purposes of the transfer.

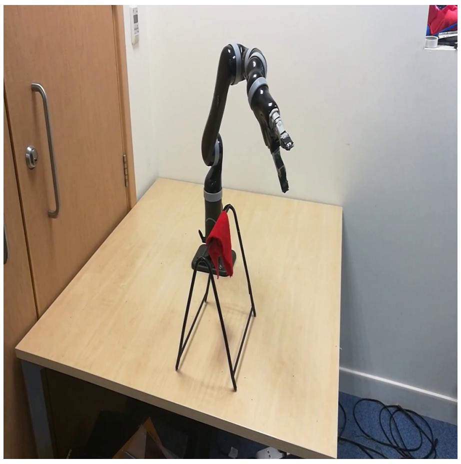
Overall, we have attempted to randomise as many parameters as possible. We, unfortunately, did not have an opportunity to do ablation studies of different randomisations to verify how much randomisation is necessary for successful transfer, but it is an interesting suggestion for future work.

We also implement an interesting interaction between the reset to demonstration feature of the learning algorithm (see section 5.3.5) and domain randomisation. Whenever we reset the environment to a state sampled from a demo, we re-randomise as many environment characteristics as possible. For example, the arm spawn position needs to stay the same as it was during a demonstration because otherwise the gripper position would change and it would no longer be grasping the cloth. However, other things such as textures, camera position and lightning can be easily sampled again. Consequently, even though we are resetting the agent to a low-dimensional state which is already in the replay buffer, we are getting entirely new set of associated RGB observations. It would be an interesting future research direction to augment the collected demonstrations by this technique before the start of learning.

We also tried more aggressive randomisation of textures, where we sampled the colours from uniform distribution across all colours instead of normal distribution centred at real noisy measurement. The learning algorithm still converged to reasonably good policy. However, this policy did not transfer as well. We were working on the pushing task at the time, and we have noticed that the policy trained with uniform textures is much more likely to miss the cube and only very rarely completes the task in the real world, compared to the good performance of the agent trained with "normal textures". We therefore decided to only employ "normal textures" at the cost of some environment generalizability.

6.2 Real world setup

The setup we used in real-world was very close to the simulation environments we presented in previous sections. All real-world experiments were done in Dyson lab at Imperial College campus. The integral part of our setup was Kinova Mico 6DOF robotic arm. The hardware of the arm is built from carbon fibre, and it can lift


(a) Sideview of the real world setup

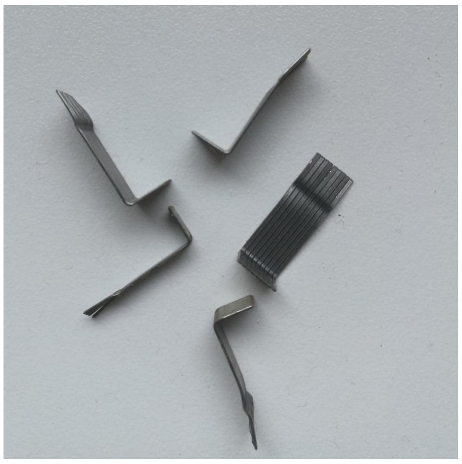

(b) Fingernails used before we received new grippers.

Figure 6.2: Setup for evaluating the policy in the real world.

payloads of up to 2 kilograms. The arm is equipped with a two finger gripper. Unlike most of the arm, the gripper did not have a particularly robust build, and it was prone to breaking.

We did a few hardware alternations of the gripper to make it less likely to get damaged by hitting the table during an experiment. Firstly, we added a small ring made out of wire around gripper fingers that helped to stabilise them and ensured the finger bone would not tear through the plastic coating. Secondly, we secured the wires with duct tape, so they do not slip out of place during operation. Finally, we added "fingernails" to the tips of gripper fingers. The fingernails considerably helped to stabilise the cloth grasping, and they also helped the gripper fingers to slide on the table. The arm usually grasps the cloth by placing the fingertips of an open gripper on the table and the closing it. The fingertips have a rubber coating that has very high friction with the table, and they, therefore, make this approach not feasible. Adding the fingernails allows the fingers to slide and moreover they also grab the cloth from underneath, which makes the grasp stronger. These changes were no longer necessary after we received replacement finger from the manufacturer which had much better dynamics.

We used a standard low-cost Genius C170 webcam to collect the RGB observations in the real world. The camera was mounted on a static tripod at a fixed location that we measured and attempted to replicate in the simulation. We also measured the field of view of the camera. We had to slightly crop the camera image because the edges of the table were visible on the picture and we wanted to avoid the need to model them in the simulation. They were still visible even after cropping, but they were less intrusive and did not seem to cause trouble in our experiments. We noticed that reading the camera image via OpenCV [78] sometimes returns an outdated image by up to a couple of seconds. We suspected that this is due to buffering either on the device, in the driver or in OpenCV itself. We did not investigate the root cause, and instead, we moved the code to acquire the image from the camera to a separate thread. This thread is always querying the device and stores the image to an in-memory buffer protected by a lock. The actor then reads this buffer to get the observation. Stephen James suggested this implementation.

We used Robot Operating System (ROS) [71] to facilitate the communication with the robot. We have created a new OpenAI gym [19] environment `SimRealEnv-v0` , which implemented the same APIs as the simulation environments used for training. This environment, however, provided RGB observations taken from the real camera and real joint angles as measured by arm encoders. The full state of this environment returns an array of zeros because we do not have access to this information. The realEnv is therefore usable only at test time for evaluation.

Implementation-wise, we use Kinova Mico ROS packages [18] available free of charge from the supplier. `SimRealEnv-v0` imports a `MicoReal` python object which implements the same APIs as the object representing simulated arm, so the control is roughly the same. The `MicoReal` class contains a ROS node that communicates with arm driver via message passing, which is a standard way of communication in ROS. We have also decided to render a simulated arm along with the real arm. This helps us with debugging as we can visualise auxiliary predictions (cloth corner position, gripper position, tape position etc.) in the simulator along with the arm, so we have some insight about how the agent understands the scene. We have also decided to use IK solver from Pybullet package instead of the firmware IK solver because the results the two solvers gave varied considerably in terms of resulting joint angles and the firmware solver was not able to plan for positions starting with the initial configuration we used for training. We have decided not to retrain the model but we instead simply used simulation solver for real robot as well.

The pieces of cloth used in the real world were ordinary low-cost towels we have bought for this purpose. Because of the arms limited reach, we could not fold the towels at its full size but we instead "pre-folded" it, so all corners are within easy reach of the arm. The cube we used for rigid object tasks was made from paper, it had the side of 6 cm and was used in previous work by James et al. [13]. The target was a blue plastic ball taken from Lego Mindstorms NXT kit with a radius of 26 mm.

6.3 Transfer challenges

We encountered a couple of challenges when attempting the domain transfer and we will now describe the most serious ones and how we resolved them.

We often encountered problems with insufficient domain randomisation. For example, we trained a couple of models for folding up to a tape task. Those models converged successfully in simulation, but the robot in real-world always folded the cloth up to a point roughly in the middle between all possible tape positions. The auxiliary predictions indeed showed that the agent believes the tape is at $y = -0.20$ which is almost exactly the average of the tape spawn points ($y = -0.075, -0.225, -0.3$). Employing randomisation of tape colour and tape size-resolved this problem - even though the predictions were still slightly biased towards the mean, they were always accurate to a couple of centimetres. The takeaway was always to try to randomise all possible parameters.

Another challenge we encountered was the camera alignments and camera settings. Our policies did not transfer because we had the camera in significant misalignment with the view in the simulation. We viewed the camera image and simulation render side by side and overlaid them on top of each other to roughly align the main features (robotic arm stand, edge between table and wall). As we have already mentioned, we have also encountered significant problems with the field of view of the camera. In the simulation, we were training with a massive field of view (90 degrees) and with a camera located very close to the scene. At first sight, the real and simulated scenes were similar, but they differed significantly when the robot got to the edge of the frame. We have then decided to measure the camera distance and FOV, adjust the parameters of the simulation and retrain the models.

We have also encountered issues with robot safety. Some policies tended to take the gripper too low and hit the table with the fingers (which were already damaged by previous experiments). Even though we employed some hardware measures to prevent further damage (see section 6.2), we were still not confident it is safe to operate the robot mainly given the very high prices of the components. We have therefore decided to restrict the reach of the real robot, so it always stops a couple of millimetres above the table and does not push it with full force. It still sometimes happened that it touched the table in fast movement (due to sensor inaccuracy and inertia) but it happened less often, and the impact was not as threatening. Our final policies that we talk about in Evaluation (Chapter 7) did not need this protection so we disabled it. We also operated the robot at reasonably low speeds, as suggested in DDPGfD paper [7].

6.4 Summary

Overall, we successfully managed to employ domain randomisation to transfer the policies trained in simulation to the real robot (see section 7.4). We randomised a large number of simulations parameters, such as textures, lighting, scales and colours, to ensure that the agent is robust to domain changes. We then developed a sim-to-real environment that allows us to evaluate the algorithm on the real robot and we successfully resolved many hardware challenges. In the next section, we will finally evaluate our work and discuss the results in detail.

Chapter 7

Evaluation

We can roughly subdivide our work on this project to 3 different stages (which had some large overlaps):

1. **Working on the simulator and creating the environments** - the goal of this stage was to create believable deformable object simulation and a set of RL environments where we can test the learning algorithms. This work was explained in detail in Chapter 4.

2. **Exploratory research of learning algorithms and learning algorithm implementation** - in this stage, we wanted to review existing state-of-the-art algorithms, find the most suitable option and adapt it to our needs. More details on this are available in Chapter 5.

3. **Transfer from simulation to real world** - after we successfully trained the agent in simulation, we wanted to use the same policy in the real world without further training. The details of techniques used and our real world setup can be found in Chapter 6.

We will now critically evaluate our work on each stage, and we then summarise our results.

7.1 Simulator work

We will evaluate our work on simulation using a qualitative approach because it is almost impossible to reduce our work in this domain into numbers. We will aim to answer the following questions: a.) is the simulation of deformable objects believable representation of real-world behaviour? b.) is the simulation stable enough for RL purposes? c.) can the simulation run in real-time or faster?

7.1.1 Deformable object simulation quality

Overall, we subjectively consider the cloth simulation to be believable and accurate enough for the purposes of the task. However, our contribution in this area was limited to a parameter search and anchor code so we will focus on those two aspects.

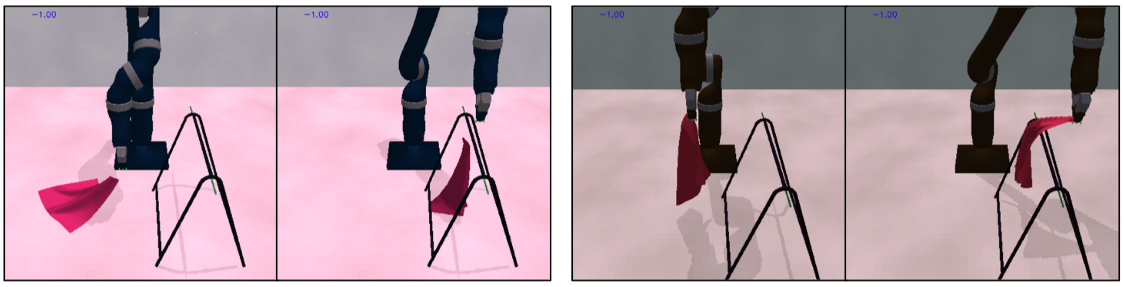


Figure 7.1: Cloth anchored to the gripper to create fake grasp before our code changes (left) and after (right)

The selection of cloth deformation parameters is essential for achieving a believable cloth behaviour. We made sure it does interact well with other rigid objects (little to no tunnelling and falling through), it is stable (the cloth does not do waving motion when left on the ground, and it does not explode when touched) and it deforms comparably to real towel (not too stiff or too elastic). All those choices were a trade-off with respect to some other properties. For example, to avoid tunnelling through a very narrow object (e.g. hanger), we had to increase the collision margin which makes the cloth float in the air. We also had to increase the number of iterations of all constraint solvers, which traded off performance. Overall, we believe that the cloth behaviour is now believable in most scenarios.

The anchor code was not suitable for our purposes before the changes we implemented. The cloth was hanging a couple of centimetres below the gripper which made all operations visually very different from real behaviour. We present the comparison of anchoring before and after our code changes in Figure 7.1. The gripping now looks reasonably accurate.

On the other hand, the simulation quality was still extremely restricting concerning the tasks we could accomplish. Perhaps most importantly, the cloth is still unstable when left crumpled on the table. It simply unfolds over time until it lays flat or mostly flat. This prevented us from trying any de-wrinkling, unfolding or re-configuration tasks. Secondly, the cloth simulation requires unnaturally large collision margins (1 to 3 cm) which causes the cloth to float over the table. This can be falsely learned by the agent that attempts the grasps a couple of centimetres above the table and hence often fails. Thirdly, the simulation does not support rendering textures on the cloth which limits both the domain randomisation aspect and the variety of clothes we can train on. Lastly, the lack of self-collisions in cloth simulation is also restricting us to select only simple manipulation tasks, and it would probably be a hurdle if we were to attempt large tasks, such as clothing assistance.

Overall, we consider our work on cloth simulation quality to be successful as it enabled us to simulate the small set of tasks we have selected for this project. However, there is much further work to be done, and we believe that cloth simulation quality remains a significant bottleneck in future RL robotic manipulation research.

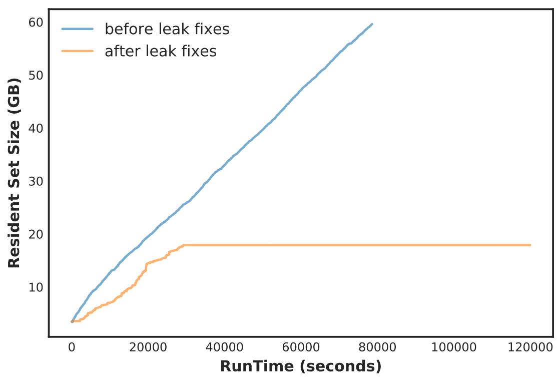


Figure 7.2: Comparison of memory used by training process (Resident Set Size) in GB before and after we worked on memory leak fixes. The initial ramp up is caused by filling the replay buffer.

7.1.2 Simulation stability

Even though we experienced a couple of problems during our early experimentation, we found the simulation to be becoming stable, and we did not experience any segmentation faults or fatal errors during the last month of project work (during which we were running at least ten simulation instances non-stop). Our work in the area included debugging intermittent memory leaks in the large codebase, which were extremely difficult to identify with usual tools and also debugging rare segmentation faults. Altogether, the fixes we have identified contributed to a significant decrease of simulation memory usage, and it allowed us to run indefinitely long experiments (see figure 7.2). The fixes were also contributed back to the library for the benefit of the community.

Overall, we found that even though the library maintainers do not yet support soft bodies, their implementation is reasonably stable. We fixed a few remaining issues, and the current version is stable enough to run for prolonged periods necessary for RL experiments.

7.1.3 Simulation performance

We did not find the performance of the library to be limiting, so we did not attempt to optimise the existing code-base. We, however, made sure that the code we added to the library does not add too much computational overhead, and we compiled the library to use native architecture optimisations on each machine we used.

Cloth simulation, however, is still significantly more expensive than a simulation with exclusively rigid objects (as we show in Figure 7.3). All times were measured on DoC machine with Intel Xeon E5-1630 (3.70GHz). For the sake of cloth stability, we are running the simulation at very low discrete time step (1/240s), and each environments step corresponds to 5 simulation steps. Therefore the simulation time elapsed per environment step is roughly 0.02s. As we can see from the figure, median wall time for an environment step with rigid objects is 0.05s while it is 0.2s for a cloth environment. The simulation is therefore not running real-time even with rigid objects, and this difference is even more pronounced with cloth (wall-time

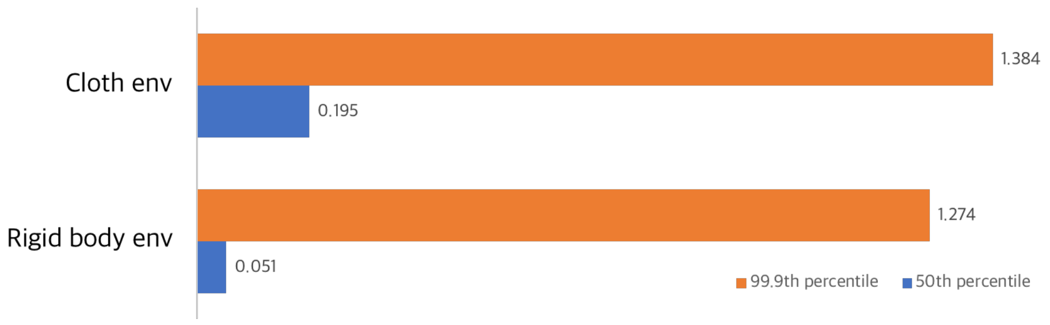


Figure 7.3: Comparison of wall time of execution (in seconds) of one simulation step between Hanger environment (cloth env) and Reacher environment (rigid body env).

is 10x larger than the time elapsed in simulation). Even though we see that the simulation is quite slow, it is still faster than the training of the networks. As both are happening in parallel on different threads, we did not have an incentive to improve on this.

7.1.4 Summary

Overall, we consider the simulation stage of this project to be successful. We went a long way from the first simulation experiments in V-Rep up to the randomised cloth environments we used for training. We are glad we achieved sufficient cloth simulation quality and simulation engine stability to be able to train our models and achieve the transfer. However, there is still some work to do to improve the simulation performance and unlock the potential for simulating more complex tasks with deformable objects.

7.2 Learning in simulated environments

The core objective of this project was to develop an RL algorithm that can learn to do deformable object manipulation tasks in the simulation. We have addressed this problem using DDPG with a large variety of improvements. Overall, we believe that this objective has been fulfilled. We will substantiate this claim by first showing the evaluation results on the three different cloth manipulations tasks, and we will then discuss the learning plots to explain the progress of the algorithm. We will also perform a series of ablation studies to show which features of the algorithm were the most crucial to achieving the final success rate.

7.2.1 On the variance in results

There is recently a growing concern about the large variance of all RL results [93]. The variance is caused by a variety of things inherent to RL: the environments are non-deterministic, the learning algorithms use a large number of stochastic elements,

	Seed1	Seed2	Seed3	Average
Folder	0.9	0.9	0.9	0.9
Hanger	0.8	0.9	0.6	0.77
Tape	0.7	1	0.9	0.86

Table 7.1: The success rates of our algorithm on the 3 environments with full randomisation.

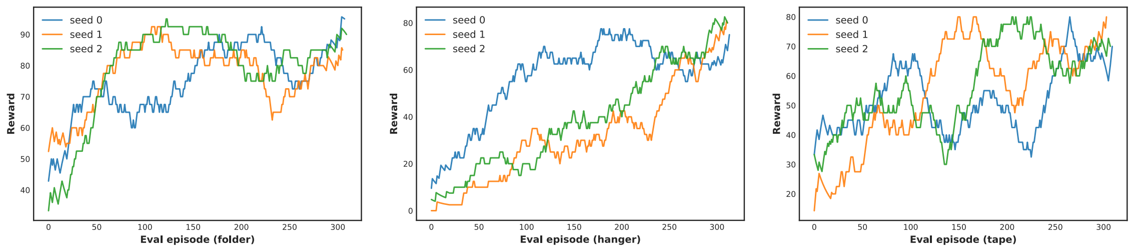


Figure 7.4: Evaluation rewards on the Folder(left), Hanger(middle) and Tape(right).

and the exploration of the agents is based on chance - either they find a high reward strategy or not.

The RL community usually addresses this issue by running each experiment multiple times with a large number of different random seeds, and they then report the mean and the standard deviation bounds. However, in our case, running a single experiment to completion can take up 48 hours and given the limited computer power, we are not able to run all experiments with a large number of seeds. However, all the results we present are a result of running with two or three different seeds, and we comment on the variance we have observed. We find two seeds to be a reasonable trade-off between data reliability and the number of ablation experiments we would like to complete with a limited amount of time until the deadline.

7.2.2 Success rates in a simulation of baseline runs

We report the simulation success rates of the algorithm with all improvements as described in section 5.3 in table 7.1. We call those runs "baselines" because we will compare against them in ablation studies. All runs (even across tasks) were run with the same learning algorithm and same hyper-parameters. We got the numbers by examining last ten evaluation episodes of training and looking at the number of successes. The environments were running with full randomisation, evaluation environments were never seen before, the training took 34 hours (using shared GPU), and each agent has seen roughly 80k simulation steps. It should be noted that this can be considered excellent sample efficiency. For example, state-of-the-art Rainbow DQN [94] takes roughly 18M frames to achieve human-level performance, even though the comparison is not entirely fair because Rainbow does not use demonstrations.

The learning graphs corresponding to the same training runs can be seen in Figure 7.4. The graphs are smoothed with an average window of 20 to make them more legible. They show how the reward awarded for evaluation episodes evolved during training. Particularly in case of Hanger environment, we can see that the rewards yet did not plateau and there was some potential for further improvement. We, however, ended the experiment to make sure we can reasonably keep all experiments running for the same amount of time in ablation studies to allow for fair comparison.

These are not the best success rates we have achieved throughout the project. For example, we were able to get a success rate of 95% on Tape task (19 successes out of last 20 evaluations) in a couple of experiments. However, we were using a reward structure that was awarding -1 for each step before the success, while experiments in this section use positive structure of +100 on success and 0 otherwise. We found that negative structure worked better in terms of final success rate, but it was significantly less robust to random seed changes (e.g. out of 3 runs with same parameters, one converged to almost 100% success rate, and others failed).

We also looked at failure cases of each model to understand why the success rate is not reaching 1. The analysis is listed below:

- **Tape failure cases**:
  - **Repeated failed grasp** - even tough the agent has learned to re-grasp the cloth if the grasp failed, in some episodes, it repeatedly fails until time runs up. The most likely cause of failure is trying to grab above the cloth. We hypothesise this might be due to an outlier in camera spawn position or arm spawn position.
  - **Grasp crumpling the towel** - grasp often crumples the towel. When it is released (even directly over the tape), it immediately flattens out, and the corners fall out of the reward threshold.
  - **Towel twisting in mid flight** - even though we employ multiple anchors with our "fake grasp", the grasp is still not stable, and the towel sometimes rotates when grasped, which causes corner misalignment when released. We do not believe this failure would happen in real world.
- **Hanger failure cases**
  - **Grasping at the wrong place** - The hanger agent is likely to miss the cloth corner - the gripper sometimes grasps the cloth too far to the right outside of the cloth edge. This might be due to the agent learning to "cut corners" - after the grasp, the next move is to drape the cloth over the hanger which is always on the right. It has therefore learned to grasp the cloth as close to the hanger to reduce the trajectory length, but sometimes is miss-predicts how far it can go from the cloth and still get a successful grasp.
  - **Draping the towel too far** - much rarer but still present failure case is that hanger pulls the cloth over the hanger too far, so it falls.
- **Folder failure cases**
  - **Crumpling the cloth** - The most likely problem with Folder is the cloth becoming crumpled when the diagonal folding motion is executed inaccurately or if the robot moves too fast.

- **Grasping issues** - similar grasp issue as with Tape environment is present in Folder.

Overall, we consider the simulation results to be satisfactory. The agent was able to learn competent policies even despite heavy domain randomisation. Even though the training took a long period of time, the sample efficiency was outstanding. We have also found the algorithm to be robust to random seed changes as all our baseline runs converges to similar final results. We will now look under the hood of those training runs to better understand the behaviour of the algorithm.

7.2.3 Plot analysis

The presence of demonstrations in our algorithm moves the originally pure RL problem to a grey area between RL and supervised learning. We would like to understand how much of the training is supervised and how does the ratio between supervised learning and RL evolve.

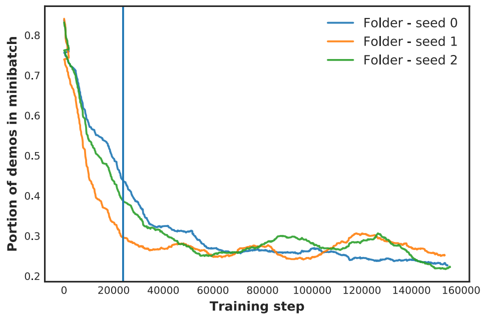
We show the relevant plots in Figure 7.5 (both curves smoothed with a window of 50). The first plot shows how the number of sampled demonstrations decreases over time in Folder environments. The vertical line marks the moment in training when the replay buffer achieves its maximal size and when it starts evicting old experiences. The sharp drop at the start can, therefore, be attributed to the replay buffer simply being diluted - when more transitions are stored, it is less likely that demonstrations are sampled because their number remains fixed. However, the percentage of demonstrations in minibatch keeps decreasing afterwards. We can conclude that the agent is learning from the demonstrations repeatedly and it begins to remember them. Hence, the loss of demonstrations decreases which in turn decreases the temporal difference error and sampling priority. The percentage of demonstrations plateaus towards the end, which can be attributed to fixed demo-epsilon that keeps the priority non-zero even if TD-errors are low.

The second plot shows us that Q-filter removes approximately 50% of demo transitions as being inferior to the actor policy. We intuitively expected the curve to follow a decreasing trend as we hoped the Q-filter would eliminate more and more demonstrations from Behavioural Cloning. However, the curve line seems to stay fixed. The presence of priority replay can explain this. The sampling is biased towards the demonstrations that still provide a learning opportunity, and therefore the proportion of demonstrations that rejected might not necessarily drop.

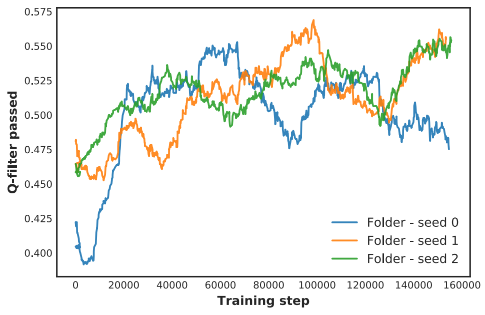
Overall, it seems that the learning gradually progresses from heavily supervised phase to RL phase as the percentage of sampled demonstrations drops from 100% during pre-training to roughly 23% at the end. These are the desired dynamics - we want the agent to initially understand the right behaviour from supervision and then start exploring other options on its own.

7.3 Ablation studies in simulation

We have done a series of ablation experiments where we removed (or changed) some feature of the learning algorithm, and we observed the results. To reduce the



(a) The portion of demos sampled in a minibatch. The vertical line shows the training step when replay buffer was saturated.



(b) The portion of demos passing the Q-filter (demos that are "rated better" by critic then actor policy).

Figure 7.5: Demo sampling rate and Q-filter rate

number of necessary experiments, we have decided to analyse only the Folder task. We always compare the ablation result to the mean of the three baseline runs (agent with all improvements).

7.3.1 N-step returns

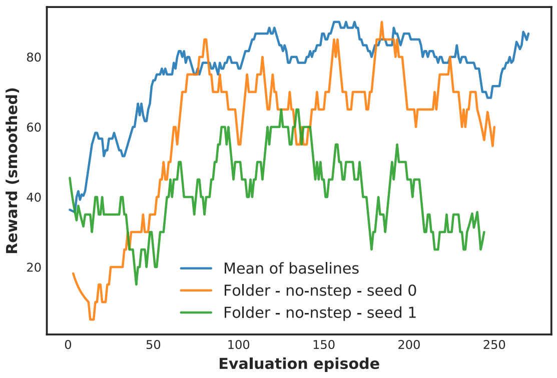


Figure 7.6: Comparison of evaluation reward development with and without N-step returns.

We present the result of removing the N-step returns from the critic loss equation in Figure 7.6. The mean of baseline rewards is consistently higher than both runs without N-step returns. We find this result consistent with the existing literature [62] and also with our preliminary experimentation. We use quite high $N = 10$ in presented runs, but we have also tried training with $N = 5$, and we did not see any significant increase or decrease of performance as compared to $N = 10$.

We understand that removing a term from critic loss equation results in decreasing the magnitude of the critic loss, which in turn might disrupt the balance of loss terms and decrease performance. We accounted for this possibility by doubling the 1-step loss multiplier in those experiments, so the change in critic loss magnitude is less significant.

7.3.2 Twin critic

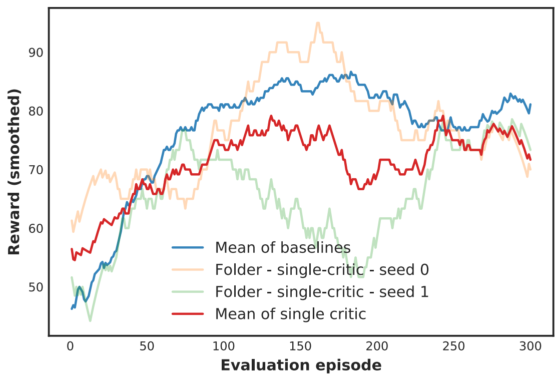
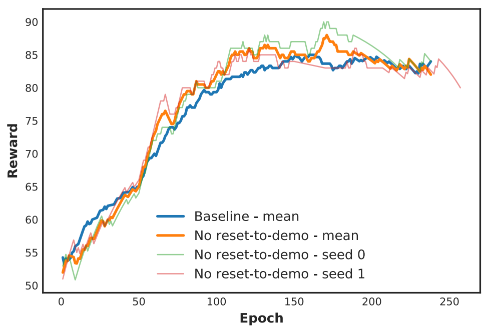


Figure 7.7: Comparison of evaluation reward development with and without twin critic.

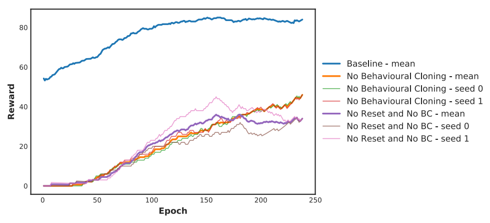
Twin critic is one of the most debatable additions to our algorithm because it is very new research result not yet verified by multiple studies. We present the comparison of experiments with only a single critic and baseline experiments in the figure 7.7.

It seems that the mean of experiments with only a single critic stays consistently below the mean of baselines. However, one of the two runs stayed very close or even above the baseline line for a large portion of the experiment. Overall, it seems like having twin critic might be useful, but we cannot conclusively rule out the possibility that the difference in mean is due to only random seed selection. Even if the difference is statistically significant, it is not large enough to justify the extra computational cost (extra 1 % for asymmetric critic but much more for symmetric critic). We would need to run many more experiments with different seeds and ideally also do a hyper-parameter sweep to get a more reliable result.

7.3.3 Reset to demonstration



(a) Ablating reset to demonstration does not decrease the model performance.



(b) Ablating both reset to demo and BC has similar effect as ablating only BC.

Figure 7.8: Reset to demonstrations ablations.

Perhaps the most surprising result was that reset-to-demonstration does not increase the performance of the agent. This finding is in sharp disagreement with our early experiments when we could not get a model to converge without this feature. We hypothesise that reset-to-demo stopped being necessary after the introduction of behavioural cloning loss and prioritised treatment of demonstrations in replay buffer.

This hypothesis states that after introducing BC loss coupled with pre-training, the agent learned the basics of the behaviour very early in training (which we can see on some evaluation plots - some runs have a strong evaluation episode in the first couple of epochs). Therefore the agent could accomplish the full sequence of necessary movements on its own and does not need the reset to a demonstration to get exposed to high rewards states. This was shown to be false by ablating both reset-to-demo and BC loss - the results were comparable to the results of ablating only the BC loss. Hence reset-to-demo was not useful even in this case.

Another hypothesis is that frequent resets to demo might cause the agent to over-fit for the demo configuration (e.g. cloth position) or that some of the states from demonstrations might be suboptimal for the current policy of the agent, so it might be better off learning without being forced to visit them. We also believe that prioritised sampling of demo transitions made reset-to-demo obsolete. If the agent has enough diverse demo transitions to learn from already, it might not be that useful for it to reset into the same state it has already seen in saved demonstration data.

7.3.4 Number of provided demonstrations

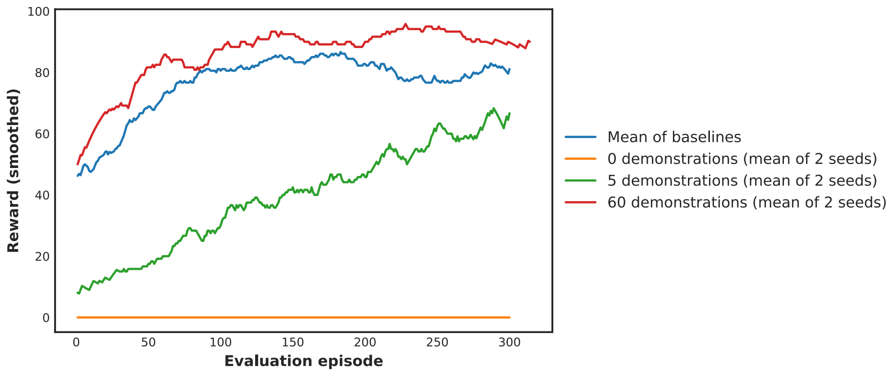


Figure 7.9: Comparison of agent performances when given different number of demonstrations.

One of the main criticism of imitation learning and RL methods based on demonstrations is the amount of work required to collect the data. We were therefore interested to know if the number of provided demonstrations has strong impact on performance. We present the results on Figure 7.9.

As we have expected based on our initial DDPG experiments (see section 5.2.1), an agent with no demonstrations in the presence of sparse reward does not finish the task because it is unlikely it would be able to grasp the cloth corner and drape it diagonally without crumpling by purely random action.

We are more interested in the comparison of performance when the agent is given 5, 20 (baseline) and 60 demonstrations. Unsurprisingly, an agent with more demonstrations behaves better than an agent with fewer demonstrations. However, we can see that the advantage of having more demonstrations is diminishing - there is a significant difference throughout the training between 5 and 20 demonstrations, but the difference between 20 and 60 is much smaller. Moreover, the differences are closing towards the end of the training.

Overall, we can conclude that using demonstrations is necessary for successful completion of the task. However, providing more demonstrations gives only a slight benefit in final performance, and the only substantial difference is the speed of training (the runs with many demonstrations seem to plateau after 100 evaluation episodes while the run with five demonstrations takes three times longer).

7.3.5 No auxiliary predictions

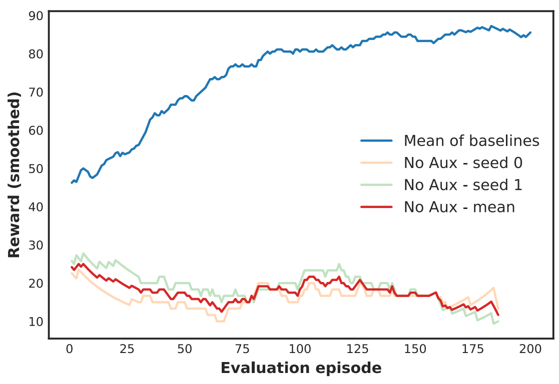


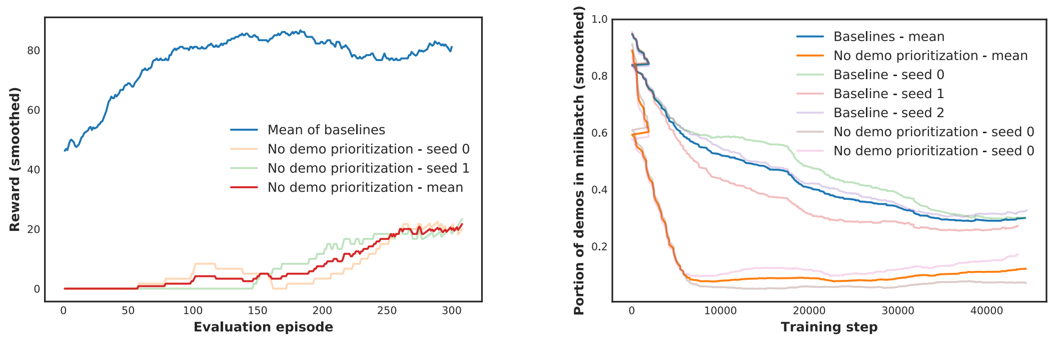
Figure 7.10: Comparison of agent performances with and without predicting low dimensional state components (eg. cloth corners).

Auxiliary predictions should help the agent understand how to translate high dimensional RGB observation into low dimensional features. In our case, we are predicting cloth corners and task target (irrelevant for Folder task and therefore set to 0), and in experiments without explicitly passing in gripper position, we also predict it (see 7.3.8). We wanted to understand how predicting those values helps the agent performance, so we set the losses on those network outputs to zero, effectively ablating this algorithm feature. Results can be seen in Figure 7.10. It should be noted that this experiment ended 10 hours early due to the machine running out-of-memory, but we believe we have gathered sufficient data for this ablation.

We can easily see that auxiliary predictions have a sizeable positive impact on learning performance. We are hinting the agent how to extract some critical features while we are not preventing it from recognizing the others, should it need to do so (this is in contrast with bottleneck approach [95] which constraints a whole actor layer to predict low-dim state so no other information can get through to fully connected layers). The downside of adding this information is the need to define the "useful" features that we want to help the agent discover, which reduces the generalizability of the algorithm and increases the amount of hand engineering.

Overall, this algorithm improvement is beneficial for both final performance and training time, but it does require domain-specific knowledge to implement.

7.3.6 Prioritized demonstration sampling



(a) Results of ablating demo priority boosting (setting ϵ_d to 0).

(b) Portion of demoes in minibatch after ablating ϵ_d.

Figure 7.11: Results of ablating demo prioritization.

Prioritized demo sampling increases the probability the demonstrations are sampled from the replay buffer. The rationale behind this design is that demonstrations are known to show right behaviour that we want the agent to know about. The result of ablation experiment can be seen in Figure 7.11.

We can see that without demo prioritisation it takes much longer for first successful evaluation episodes to appear and overall performance is significantly behind the baseline runs. We can see the strength of demonstration priority boosting on the right - the baseline runs are approximately eight times more likely to sample a demo around step 10000 (which is quite shortly after the pre-training phase visible as a dent in the graph).

Overall, it seems that boosting the demonstrations shifts the RL/supervised learning balance a bit towards supervised learning which is beneficial for the performance, especially early on. We would be interested to see if experiments without demo-priority boosting would catch up in terms of performance later on in training (after a couple of days) but we are not able to execute the experiment due to time constraints.

7.3.7 Pre-training

As we would expect, not running pre-training on demonstration transitions causes the agent to start getting first successes in evaluation episodes later in the training process which delays plateauing of training by a couple of epochs. The policy in our experiments also converged to slightly better final rewards, but we do not believe this result is beyond the scope of statistical error given the low number of seeds we

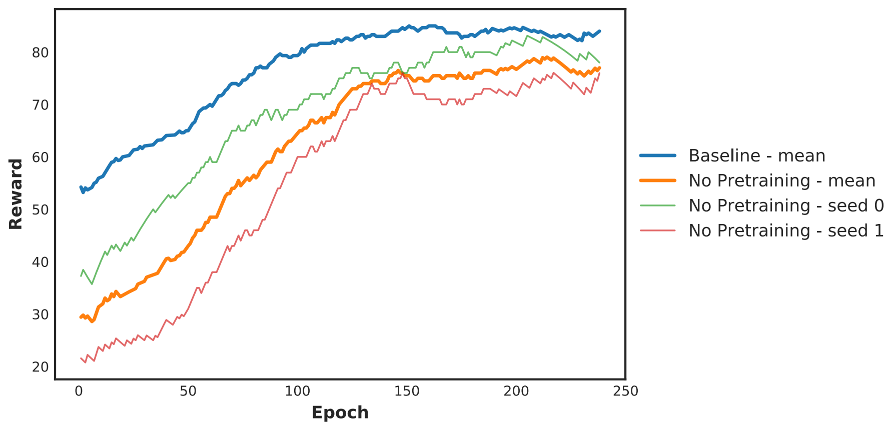
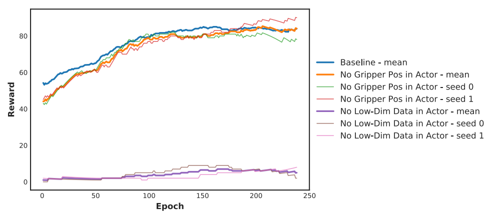


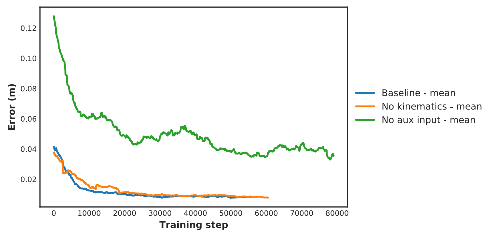
Figure 7.12: Comparison of agent performances with and without running 2000 pre-training steps (1̃0 minutes) at the start of training.

use. Overall, it seems that pre-training is worth approximately 10 minutes at the start of the training to achieve the overall speed-up.

7.3.8 Low Dimensional state in the actor



(a) Comparison of using no low-dimensional state; only joint angles; and baseline (joint angles along with gripper position).



(b) The distance between auxiliary prediction and ground truth of gripper position.

Figure 7.13: Results of ablating demo prioritization.

Adding low dimensional state to the actor might be considered a controversial choice because we are no longer inferring the motor policy purely from visual observation. However, from the practical point of view, both joint angles and gripper position are available at test time in robot API so leveraging them does not require any further engineering work.

We can see in Figure 7.13 that removing the gripper position from the state has a negligible impact on the performance. In those experiments, we added the gripper position to the set of auxiliary outputs. The loss on gripper auxiliary quickly dropped and the test distance plateaued at around 1 or 2 cm from ground truth, so we can conclude the agent is capable of confidently inferring the gripper position from joint angles and RGB images. Hence, it can learn forward kinematics. As we would expect, it took it a bit longer to plateau then baseline run which just needed to propagate the gripper position from input to output.

Hanging task	
Vicinity	100%
Grasp	76.6%
Drape over	70%
Full success	46.6%

Diagonal folding task	
Grasp	66.6%
Not crumpled	66.6%
d $\leq$ 0.15m	53.3%
d $\leq$ 0.1m	40%
d $\leq$ 0.05m	20%

Tape folding task	
Grasp	90%
d $\leq$ 0.15m	90%
d $\leq$ 0.1m	76.6%
d $\leq$ 0.05m	43%

Table 7.2: The success rates for each environment in the real world. Note that these are run in the real world without any additional training. For the hanging task, *vicinity* means the gripper being within 5cm from the cloth, *drape over* means the cloth is touching the top part of the hanger and *full success* is achieved if the cloth does not fall after it is released. For diagonal folding, *not crumpled* means that adjacent corners are more than 15cm from each other and the d is the distance between diagonal corners (lower is better). For tape folding, d is the distance between towel edge and the tape mark.

We have also tried fully ablating all low dimensional inputs (not giving gripper position nor joint angles). In this case, the smoothed error of auxiliary gripper prediction stayed above 3 cm, which is not sufficient for our fine manipulation tasks and the evaluation rewards were significantly affected. We hypothesise that it is very hard for the network to predict the position along a ray from the camera (consistently to our findings from Reacher environment explained in section 4.3.4).

7.4 Real world experiments

We evaluated our agent in real world on the experimental setup as described in Chapter 6 and we report the success rates based on 30 trials for each task in Table 7.2.

We subdivided each task into multiple subtasks so we can measure the agent performance with better granularity. For hanging, *vicinity* means that the gripper was within 5 cm of the cloth, *grasp* means that the robot lifted any part of the cloth to the air (not necessarily on the first attempt), *drape over* means that the robot started the draping motion (so the cloth touched top of the hanger) and *full-success* means that the robot draped the towel over the hanger and it did not fall.

In diagonal folding, we first looked if the agent managed to grasp the cloth. If it did, we looked at the result of the manipulation. The cloth is crumpled if adjacent corners are within 15cm from each other; otherwise we measure the alignment of diagonal corners and sort the trials into 3 buckets - aligned within 5 cm (very good alignment), within 10 cm (satisfactory alignment), 15 cm (some alignment). For tape task, we employ a similar method, but we measure the distance between the middle point of cloth edge from the tape.

We consider the presented results to be exceeding expectations. In all tasks, we achieve at least two thirds of successful grasps, which is very good given the low acceptable margin of error on z-axis when the cloth lays flat on the table. Going too low can damage the gripper while going too high results in failed grasp above the cloth. For tape task, the agent achieved a fold within 15 cm from the mark in 27 out of 30 trials which shows the robustness of our method to domain transfer.

Some results were to some extent disappointing, particularly in the diagonal folding task. Because of the large variation of cloth spawn positions, the agent achieved much lower grasp success rates and we noticed that it has more problems if we place the cloth on the edge of spawn position distribution. Moreover, it achieved a very good alignment only in 20% of the trials, compared to 90% in simulation. We have noticed that the agent tends to lift the cloth too high, which leads to deformation. This behaviour does not happen in training due to simulator inaccuracy, so the agent could not learn to avoid it. We will now discuss failure modes in more detail.

7.4.1 Failure modes

As we have already hinted, the most common failure mode was the failure to grasp the object. The agent needs to get into exactly the right z-axis position to make an efficient grasp. This success rate varied considerably from model to model, with only some training runs achieving the results we gave in the last section. A related problem is a weak grasp. Sometimes the gripper tries to grasp the cloth too close to the edge, and it only catches a couple of millimetres of the fabric. This grasp is not strong enough, and it fails when cloth touches the hanger.

Another common failure mode is movement imprecision. For example in diagonal folding, the ideal trajectory would most likely lead straight from one corner to another with little deviation. The agent, however, learned to do a suboptimal trajectory which deviates to a side. Moreover, it learned to lift the cloth a little too high. Those imprecisions often resulted in cloth crumpling. We investigated this failure mode in more detail by looking at simulation renders. The agent does the same motion, but the cloth in simulation behaves differently. It has lower linear stiffness, so it stretches when pulled up, and hence it does not crumple. This highlights that main limitation of this method is simulation accuracy, and further work will be needed to train more complex tasks with deformable objects.

7.5 Summary and Comparison with other work

Overall, we have successfully shown that DDPG with our implemented improvements can learn control policy to solve the three long horizon manipulation tasks we have defined. Moreover, we were also able to transfer the policy to the real world and achieve satisfactory success rates for the tasks.

To the best of our knowledge, this is the first application of Deep RL with sim-to-real-transfer on cloth manipulation and we, therefore, do not have a strong benchmark to compare against. We have designed one of our tasks to be similar to previous task defined by Lee et al. [21]. We do not achieve as good results as the original work. The authors do not report a failure rate and we, therefore, assume they have achieved 100% success rate in grasping the towel and folding it. Our method has achieved only 90%. The Root Mean Square error of successful folds is also better with their method (3.8cm) while ours is 6.8cm. The authors of this paper also compared their method with previous state-of-the-art - thin plate spline (TPS) [96] method and coherent point drift method (CPD) [97]. Both those methods achieve significantly worse results than ours, with RMS of 13.6cm or 26.9cm respectively.

However, we are using a task agnostic training algorithm that has learned the control from RGB (without depth) images and sensor data by itself, so those are not entirely fair comparisons. Moreover, the robotic platform they have used has a parallel gripper which allows for a wider margin of error in z-axis when top-down grasping and they do their experiments on a soft surface, which increases robot safety and further increases acceptable error in the z-axis. Overall, even though we did not achieve state-of-the-art results on this task, we achieve better scores than pre-2015 methods even tough we use task agnostic algorithm.

Looking at sim-to-real results, we can most closely compare our work to Pinto et al. [15]. Their methodology is however different. Firstly, they use a different set of tasks only dealing with rigid objects - picking up a cube, pushing the cube to a target location and moving the cube to a target location. We briefly experimented with those tasks during the project and we found them to be simpler because pushing or grasping the cube allows for significantly larger acceptable error during manipulation, and it is easier to recover from problems. Secondly, they did not use demonstrations to seed learning, although they did use a feature similar to "reset to the demonstration" when they sometimes reset the environment to a state where the agent was already holding the cube. Finally, they use a different robot platform with a parallel gripper. Their best method, Asymmetric Hindsight experience replay can achieve 100% success rate (out of 5 trials) on each task. The baselines they compare against (assym DDPG, sym HER, BC and DAgger) achieve success rates close to 0. We have achieved consistent results when experimenting with ablations. Given the larger difficulty of our tasks, we believe that the best methods presented in both works are comparable, but it is difficult to select a better one without same experimental set-up.

Chapter 8

Conclusion and Future work

We have successfully applied Deep Reinforcement Learning to the problem of soft object manipulation or more specifically to the manipulation tasks including 2D deformable objects - clothes and towels. As a first step, we have done a substantial amount of work on open-source physics engine Pybullet so we could simulate cloth with reasonable accuracy, stability and speed. We have then designed and implemented robotic environments to simulate three cloth manipulation tasks: folding a towel up to a mark (Tape), folding a face towel diagonally (Folder) and draping a face towel over a hanger (Hanger).

In the next phase, we evaluated a number of Deep RL algorithms on simple environments we have implemented based on OpenAI Gym suite, and we settled on using Deep Deterministic Policy Gradients (DDPG) for this project. The algorithm in its basic form, however, was not sufficient to learn the complex cloth manipulation tasks, so we extended it with multiple improvements described in current state-of-the-art publications. The final algorithm can achieve 80%+ success rate on all our proposed cloth manipulation tasks robustly, and some specific runs achieved success rates over 95%. We evaluated the contribution of specific improvements through an extensive series of ablation studies.

The final step of our project was to attempt to use the policies learned in simulation in the real world. We achieved reasonably good overall success rates of 66%(Folder), 46% (Hanger) and 90% (Tape) if we do not consider alignment quality. The problems we have encountered most often are failed grasps (the robot has only minimal tolerance in z-axis which makes the grasp difficult), weak grasps and imprecise movement caused by simulation inaccuracy.

Overall, the results show that using Deep RL is a viable way to tackle the problem of manipulating deformable objects both in simulation and in the real world. However, even though the tasks we have proposed are challenging given the current state-of-the-art, they are still very far from being applicable on a real-robot in uncontrolled settings, such as in-home assistance. The project has opened multiple avenues for further research we would like to discuss.

8.1 Future work

As with the rest of the project, we can subdivide the Future work into three domains - the simulation, the learning algorithms and the domain transfer. There is a vast number of questions that remain open after this project and equally many opportunities for additional research. We select a couple of them that we subjectively consider to be the most interesting, are large in scope and could serve as inspiration for a final year project or a publication.

8.1.1 Deformable object simulation and environments

Pushing the limits of the simulators

We are concerned that even though there are many simulators often used in RL community, only PyBullet currently has some limited implementation of deformable object simulation and even this implementation is not yet officially supported by the library maintainers. We believe that more work on implementing support for deformable object simulation into popular physics engines can enable future researchers to consider manipulating deformable objects along with their rigid counterparts in all future manipulation research.

There are multiple aspects of simulation that would benefit from further work. Perhaps most important would be the grasp stability. Currently, we were only able to simulate grasping by creating artificial constraints that do not accurately resemble the real behaviour. The simulator would ideally support stable grasping only based on physical interactions of the objects so it can learn the grasping policies that transfer seamlessly to the real world. It would also be useful to improve the collision accuracy between the soft-body and rigid body (avoid penetrations) and between two soft-bodies (enable self-collisions). Finally, even the simulation of small soft bodies with couple hundred nodes is computationally expensive and does not run in real time. As a large part of the soft body simulation algorithm is parallelizable, it would be interesting to study how to migrate it to GPUs.

Creating standard environments for deformable object manipulation

The introduction of OpenAI Gym Robotics environments [20] finally created a widely accepted set of benchmarks for robotic manipulation. However, those environments do not use deformable objects which again limits the number of researchers investigating the problem in the future.

It would be ideal to create a standard set of tasks where robots manipulate deformable objects and implement them as open-source environments. This would allow researchers to easily reproduce published work and compare the achieved results across multiple methods, research teams or even institutions. Furthermore, the environments should be accompanied by a standardised real-world dataset that the institutions can buy and use for testing domain transfer of their policies. The set of environments should be also tested with standard RL algorithms, and the results should be made available to serve as baselines for further experiments.

8.1.2 Learning in simulation

Explaining DDPG performance

Reproducibility of the performance achieved by RL agents is a significant open problem in the research community. DDPG, in particular, is notorious for low robustness to initialisation (and hence to random seed) and hyper-parameter selection. This becomes problematic when trying to attribute a performance change to an implementation change. For example, an experiment might result in considerably worse performance after network architecture change, but it is impossible to say if this is due to randomness in results or it is a genuine regression of the algorithm. It would be useful to do a large-scale study of various DDPG improvements that can conclusively explain what algorithm features are beneficial in which circumstances. This study would essentially be a massively scaled up version of our ablation experiments, which would include a hyperparameter sweep for each ablation and experiments with a large number of random seeds and a large number of diverse environments and tasks. However, this study would require immense computing power by design, so it can only be done by either large institutions (Google, Facebook, OpenAI...) or by a broad community. If the study was to be done by the latter, it would require an implementation of an intuitive tool that RL enthusiasts could download and let their GPU machines use unused cycles for research purposes. Similar initiatives exist in other domains (e.g. Atlas@Home for analysing data from CERN, SETI@Home for searching extraterrestrial intelligence or POEM@Home to study protein folding with applications in cancer research).

Further investigating DualArm tasks

We have done some very limited experiments with Dual Arm folding during this project that were not successful. We have seen that the two arms tend to collide in early learning stages and they therefore never converge. It would be interesting to analyse why the learning with two arms fails and find some possible remedies, maybe by addition of some explicit loss that will discourage collisions and promote collaboration. This would unlock many more cloth manipulation tasks that require dual manipulation (as discussed in the Background section).

8.1.3 Domain transfer

Automatic environment creation

We have shown that learning in simulation and subsequent domain transfer are a viable way to cope with sample inefficiency of RL algorithms. However, there is still a fair amount of engineering involved in creating the learning environments that roughly correspond to real-world setup. If we were to use this approach in future, for example with home assistance robots, we would not be able to hand-engineer a 3D model of each home so the robot can learn in there. It would be however possible if the robot could automatically generate an environment to learn from its observation or from externally recorded sensor data.

It can be an exciting application of SLAM algorithms. The robot would start by exploring the area and generating a surfel map, then converting it into a mesh representation that can be loaded by a physics simulator and finally it could export the generated data as an RL environment. The robot can then experiment and learn to accomplish any externally defined task. The reward structure would still need to be hand engineered, but future research could reveal a user-friendly way of doing so (possibly based on demonstrations).

Understanding domain randomisation

Our project made excessive use of domain randomisation to enable the transfer of policy learned in simulation into the real world. After a couple of failed experiments, we have learned a simple rule of thumb - randomise everything. Although this approach is simple, it is not always optimal.

We have seen that excessive randomisation is detrimental to both the speed and performance of learning in the simulation. Intuitively this makes sense because environments with a higher degree of randomness require more robust and generalizable policy. However, we have also seen that too strong randomisation is detrimental to domain transfer when real-world experiments trained with texture colours sampled from normal distribution worked better than experiments trained with uniformly sampled colours. Overall, it seems that there is a golden middle way when implementing randomisation.

It would be interesting to do an in-depth study into randomisation and establish a.) which properties are necessary to randomise, b.) which seem to be beneficial and c.) which do not provide a further improvement or even cause harm. We believe it would be possible to partly do this study without the need for real hardware by attempting transfers between simulation domains with different parameters. The preliminary results could then be verified in the real world.

Bibliography

[1] S. Miller, J. van den Berg, M. Fritz, T. Darrell, K. Goldberg, and P. Abbeel, "A Geometric Approach to Robotic Laundry Folding," in *Household Service Robotics*, 2014.

[2] M. Laskey, C. Powers, R. Joshi, A. Poursohi, and K. Goldberg, "Learning Robust Bed Making using Deep Imitation Learning with DART," 2017.

[3] Y. Gao, H. J. Chang, and Y. Demiris, "Iterative path optimisation for personalised dressing assistance using vision and force information," in *IEEE International Conference on Intelligent Robots and Systems*, 2016.

[4] T. Tamei, T. Matsubara, A. Rai, and T. Shibata, "Reinforcement Learning of Clothing Assistance with a Dual-arm Robot," 2011.

[5] B. Thananjeyan, A. Garg, S. Krishnan, C. Chen, L. Miller, and K. Goldberg, "Multilateral surgical pattern cutting in 2D orthotropic gauze with deep reinforcement learning policies for tensioning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[6] J. Schulman, J. Ho, C. Lee, and P. Abbeel, "Generalization in Robotic Manipulation Through The Use of Non-Rigid Registration," *International Symposium on Robotics Research (ISRR)*, 2013.

[7] M. Večerík, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards," 2017.

[8] K. Huebner, K. Welke, M. Przybylski, N. Vahrenkamp, T. Asfour, D. Kragic, and R. Dillmann, "Grasping Known Objects with Humanoid Robots: A Box-Based Approach," *International Conference on Advanced Robotics*, 2009.

[9] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates," *Proceedings - IEEE International Conference on Robotics and Automation*, 2016.

[10] J. Peters, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, 2008.

[11] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An Application of Reinforcement Learning to Aerobatic Helicopter Flight," 2007.

[12] X. B. Peng and G. Berseth, “DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning,” *ACM Trans. Graph*, 2017.

[13] S. James, A. J. Davison, and E. Johns, “Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task,” 2017.

[14] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World,” 2017.

[15] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, “Asymmetric Actor Critic for Image-Based Robot Learning,” 2017.

[16] L. Pinto, J. Davidson, and A. Gupta, “Supervision via competition: Robot adversaries for learning tasks,” 2017.

[17] E. Coumans and Y. Bai, “PyBullet, a Python module for physics simulation for games, robotics and machine learning,” 2016.

[18] Kinova Robotics, “Official ROS packages for Kinova robotic arms,” 2018.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” 2016.

[20] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba, “Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research,” 2018.

[21] A. X. Lee, A. Gupta, H. Lu, S. Levine, and P. Abbeel, “Learning from Multiple Demonstrations using Trajectory-Aware Non-Rigid Registration with Applications to Deformable Object Manipulation,” 2015.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” 2015.

[23] A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” 2017.

[24] A. Deshpande, “A Beginner’s Guide To Understanding Convolutional Neural Networks,” 2018.

[25] D. P. Kingma and J. L. Ba, “Adam: a Method for Stochastic Optimization,” *International Conference on Learning Representations 2015*, 2015.

[26] M. Inaba and H. Inoue, “Hand Eye Coordination in Rope Handling,” *Journal of the Robotics Society of Japan*, 1985.

[27] A. Remde, D. Henrich, and H. Wörn, “Picking-up deformable linear objects with industrial robots,” 1999.

[28] M. Saha and P. Isto, "Manipulation Planning for Deformable Linear Objects," *IEEE Transactions on Robotics*, 2007.

[29] H. Wakamatsu, E. Arai, and S. Hirai, "Knotting/Unknotting Manipulation of Deformable Linear Objects," *The International Journal of Robotics Research*, 2006.

[30] B. Kahl and D. Henrich, *Virtual robot programming for deformable linear objects: system concept and prototype implementation.* 2002.

[31] K. Hirana, T. Suzuki, S. Okuma, K. Itabashi, and F. Fujiwara, "Realization of skill controllers for manipulation of deformable objects based on hybrid automata," 2001.

[32] S. Yue and D. Henrich, "Manipulating deformable linear objects: sensor-based fast manipulation during vibration," 2002.

[33] G. Nair, I. Daut, V. Kumaran, M. Irwanto, Y. M. Irwana, and M. Zambak, "Photovoltaic Powered T-Shirt Folding Machine," 2013.

[34] D. Lee, "This machine can fold an entire load of laundry in four minutes," 2018.

[35] N. Fahantidis, K. Paraschidis, V. Petridis, Z. Doulgeri, L. Petrou, and G. Hasapis, "Robot handling of flat textile materials," *IEEE Robotics & Automation Magazine*, 1997.

[36] F. Osawa, H. Seki, and Y. Kamiy, "Unfolding of Massive Laundry and Classification Types by Dual Manipulator," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 2007.

[37] J. Maitin-Shepard, M. Cusumano-Towner, J. Lei, and P. Abbeel, "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2010.

[38] P. C. Wang, S. Miller, M. Fritz, T. Darrell, and P. Abbeel, "Perception for the manipulation of socks," 2011.

[39] A. Ramisa, G. Alenya, F. Moreno-Noguer, and C. Torras, "Using depth and appearance features for informed robot grasping of highly wrinkled clothes," pp. 1703–1708, IEEE, 2012.

[40] C. Bersch, B. Pitzer, and S. Kammel, "Bimanual robotic cloth manipulation for laundry folding," in *IEEE International Conference on Intelligent Robots and Systems*, 2011.

[41] M. Cusumano-Towner, A. Singh, S. Miller, J. F. O'Brien, and P. Abbeel, "Bringing clothing into desired configurations with limited perception," *Proceedings - IEEE International Conference on Robotics and Automation*, 2011.

[42] Y. Kita, T. Ueshiba, E. S. Neo, and N. Kita, "A method for handling a specific part of clothing by dual arms," 2009.

[43] "Maya | Computer Animation & Modelling Software | Autodesk."

[44] J. Van Den Berg, S. Miller, K. Goldberg, and P. Abbeel, "Gravity-based robotic cloth folding," in *Springer Tracts in Advanced Robotics*, 2010.

[45] Y. Yamakawa, A. Namiki, and M. Ishikawa, "Motion planning for dynamic folding of a cloth with two high-speed robot hands and two high-speed sliders," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2011.

[46] V. Petrík, V. Smutný, P. Krsek, and V. Hlaváč, "Single arm robotic garment folding path generation," *Advanced Robotics*, 2017.

[47] Y. Li, Y. Yue, D. Xu, E. Grinspun, and P. Allen, "Folding Deformable Objects using Predictive Simulation and Trajectory Optimization," 2015.

[48] L. Sun, G. Aragon-Camarasa, S. Rogers, and J. P. Siebert, "Robot Vision Architecture for Autonomous Clothes Manipulation," 2016.

[49] K. Sun, G. Aragon-Camarasa, P. Cockshott, S. Rogers, J. P. Siebert, L. Sun, G. Aragon-Camarasa, P. Cockshott, S. Rogers, and J. P. Siebert, *A Heuristic-Based Approach for Flattening Wrinkled Clothes.* 2013.

[50] I. Lenz, H. Lee, and A. Saxena, "Deep Learning for Detecting Robotic Grasps," 2013.

[51] Cornell University Computer Science Department, "Cornell Grasping Dataset."

[52] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine, "Combining self-supervised learning and imitation for vision-based rope manipulation," 2017.

[53] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours," 2016.

[54] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, "Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection," 2016.

[55] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-Real Robot Learning from Pixels with Progressive Nets," 2016.

[56] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[57] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning,” *PMLR*, 2016.

[58] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014.

[59] R. J. Williams, “Simple statistical gradient-following methods for connectionist reinforcement learning,” *Machine Learning*, 1992.

[60] G. E. Uhlenbeck and L. S. Ornstein, “On the theory of the Brownian motion,” *Physical Review*, 1930.

[61] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” 2018.

[62] G. Barth-Maro, M. W. Hoffma, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, T. Lillicrap, and D. London, “Distributed Distributional Deterministic Policy Gradients,” 2018.

[63] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. Mcgrew, J. Tobin, P. Abbeel, and W. Z. Openai, “Hindsight Experience Replay,” 2017.

[64] J. Schulman, F. Wolski, and P. Dhariwal, “Proximal Policy Optimization Algorithms Background : Policy Optimization,” *CoRR*, 2017.

[65] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “ High-Dimensional Continuous Control Using Generalized Advantage Estimation,” *ICML*, 2015.

[66] NVidia, “The NVIDIA PhysX Physics Engine.” http://www.nvidia.com/object/physx_new.html.

[67] Havok, “Havok Physics Engine,” 2018.

[68] Coppelia Robotics, “V-Rep - Coppelia Robotics.”

[69] Blender Online Community, “Blender - a 3D modelling and rendering package.”

[70] Morse Online Community, “Morse - Modular OpenRobots Simulation Engine.”

[71] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, “ROS: an open-source Robot Operating System,” *Icra*, 2009.

[72] Phobos Online Community, “Phobos.”

[73] R. Kaestner, “Blender- URDF.”

[74] Bullet Physics Online community, “Bullet Physics Engine.”

[75] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2015.

[76] R. Featherstone, *Robot dynamics algorithms*, vol. 25. 1989.

[77] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "OpenAI Baselines." https://github.com/openai/baselines, 2017.

[78] OpenCV, "OpenCV," 2018.

[79] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *IEEE International Conference on Intelligent Robots and Systems*, 2012.

[80] Fetch-Robotics, "Fetch Mobile Manipulator." https://fetchrobotics.com/, 2018.

[81] G. Rossum, "Python Reference Manual," tech. rep., Amsterdam, The Netherlands, The Netherlands, 1995.

[82] M. Plappert, "keras-rl." https://github.com/keras-rl/keras-rl, 2016.

[83] J. Bergstra, D. L. K. Yamins, and D. D. Cox, "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures," *Icml*, 2013.

[84] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, and G. Brain, "TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[85] F. Chollet, "Keras as a simplified interface to TensorFlow: Tutorial," *The Keras Blog*, 2016.

[86] J. Zacharias, M. Barz, and D. Sonntag, "A Survey on Deep Learning Toolkits and Libraries for Intelligent User Interfaces," 2018.

[87] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," 2015.

[88] T. Zhang, Z. McCarthy, O. Jow, D. Lee, X. Chen, K. Goldberg, and P. Abbeel, "Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation," 2017.

[89] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming Exploration in Reinforcement Learning with Demonstrations," 2017.

[90] H. V. Hasselt, A. C. Group, and C. Wiskunde, “Double Q-learning,” *Nips*, 2010.

[91] K. Perlin, “An image synthesizer,” *ACM SIGGRAPH Computer Graphics*, 1985.

[92] C. Duncan, “Perlin noise library for Python,” 2016.

[93] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep Reinforcement Learning that Matters,” *CoRR]*, 2017.

[94] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining Improvements in Deep Reinforcement Learning,” 2017.

[95] Fangyi Zhang, J. Leitner, B. Upcroft, and P. Corke, “Vision-Based Reaching Using Modular Deep Networks: from Simulation to the Real World [arXiv],” *arXiv*, 2016.

[96] F. L. Bookstein, “Principal Warps: Thin-Plate Splines and the Decomposition of Deformations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1989.

[97] A. Myronenko and X. Song, “Point set registration: Coherent point drifts,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.