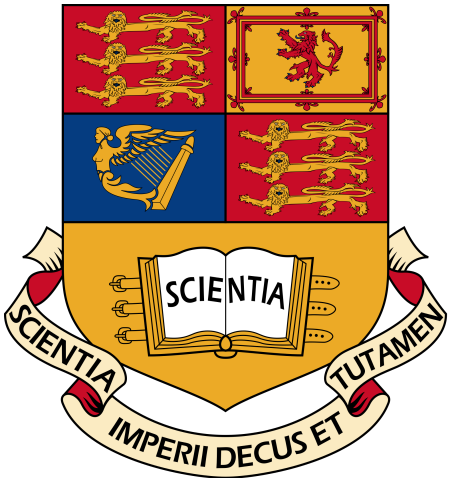
IMPERIAL COLLEGE LONDON

INDIVIDUAL PROJECT MENG

3D Simulated Robot Manipulation Using Deep Reinforcement Learning

Author:
Stephen JAMES

Supervisor:
Dr. Edward JOHNS



June 12, 2016

Abstract

Robots are increasingly becoming part of our lives, but despite their impressive repertoire of tasks, many of them will fail to adapt when presented to new and unfamiliar environments. Before robots can realise their full potential in everyday life, they need the ability to manipulate the changing world around them. Recent trends to solve this problem have seen a shift to end-to-end solutions using deep reinforcement learning policies from visual input. Building upon the recent success of deep Q-networks (DQNs), we present an approach that uses three dimensional (3D) simulations to train a six-joint robotic arm in an object manipulation task without any prior knowledge. Policies accept images of the environment as input and output motor actions. The high-dimensionality of the policies as well as the large state space makes policy search difficult. This is overcome by ensuring interesting states are explored via intermediate rewards that guide the policy to high reward states. Our results demonstrate that DQNs can be used to learn policies for a task that involves locating a cube, grasping, and then finally lifting. The agent is able to generalise to a range of starting joint configurations as well as starting cube positions. Moreover, we show that policies trained via simulation are able to be directly applied to real-world equivalents without any further training. We believe that robot simulations can decrease the dependency on physical robots and ultimately improve productivity of training object manipulating agents.

Acknowledgements

I would like to thank:

- My supervisor, Dr. Edward Johns, for his invaluable guidance and enthusiasm in this project.
- My family for their unconditional love and support.
- Lloyd Kamara from CSG for his assistance in machine allocation and running experiments without display.
- My flatmates, Andrew and Adam, for their understanding and support when I needed it the most.

Contents

Chapter 1

Introduction

Imagine a newly purchased robot assistant. You unpack it, power it on, and walk it to the kitchen. You begin cleaning the worktop, washing the dishes, and emptying the dishwasher. After a while of observing, the assistant joins you emptying the last few dishes. It has not seen your house before, nor has it seen your particular style of dishes before, yet it is capable of identifying that they are in fact dishes, and manages to put them away correctly.

Robot manipulation can be seen today in many industries across the world. They have the ability to lift massive loads, move with incredible speed, and perform complex sequences of actions with pin-point accuracy. Yet, despite this, robot kitchen assistants are still ideas of science fiction.

Despite their impressive repertoire of tasks, industrial robots will fail to adapt when presented a new and unfamiliar environment. This comes down to the dynamic and unpredictable nature of human environments which cannot be preprogrammed, but instead must be learned first hand by the robot.

Advances in robot technology has seen impressive feats of work from the likes of Boston Dynamic's Atlas [5] and Dyson's self cleaning vacuum [23]. In coming years, the number of robots inhabiting our living spaces is expected to increase dramatically. However, before robots can realize their full potential in everyday life, they need the ability to manipulate the changing world around them. Traditionally, robot manipulation has been solved by engineering solutions in a modular fashion using active vision [36] [32].

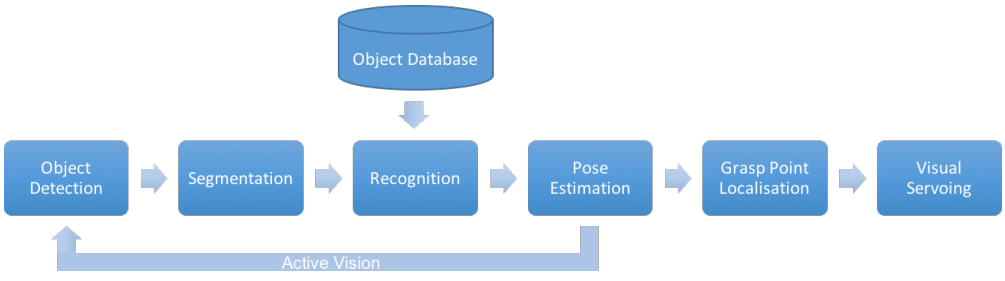


FIGURE 1.1: Traditional active vision pipeline for object manipulation.

Active vision [2] has had tremendous success over the years not only in object manipulation, but also visual motion, navigation and 3D recovery [63].

Summarised in figure 1.1, the process first consists of visually exploring the scene in order to find points of interest. The scene is then segmented in order to identify an object using data from an *object database* containing 3D models of known objects to produce a suitable pose alignment through pose estimation. Once the object identity is known, grasp point localisation is performed to find a suitable grasp configuration by using a box-based approach where the object shape is approximated by a constellation of boxes. Finally, visual servoing is applied to bring the robotic hand to the desired grasp position. Often it is not always the case that all segments are used in the pipeline.

In its aim to simplify the overall problem of manipulation, the approach above can result in loss of information between each of the modules and relies on a number of assumptions — such as the availability of complete knowledge of the object to be grasped. To overcome this, recent trends have seen robots visually learning to predict actions and grasp locations directly from RGB images using deep reinforcement learning — a combination of deep learning and reinforcement learning. However, these approaches are run on real-world robotic platforms which often require human interaction, expensive equipment, and long training times.

In addition to robotics, reinforcement learning has received great success in learning control policies to play video games with a Q-learning variant called deep Q-learning [65]. Deep Q-learning has been shown to learn policies directly by receiving only pixels and score from a range of Atari-2600 games — resulting in agents that are able to match human level control [64].

Given that these methods can be applied to video games, is it then possible to apply similar techniques to robot simulations? If this was the case, then future training for robot control could be done using life-like simulations and then be directly mapped to real-world hardware without the burden of training human interaction on the training process.

Building upon the recent success of deep Q-networks, we present an approach that uses three-dimensional (3D) simulations to train a six-joint robotic arm in an object manipulation task without any prior knowledge. Policies accept images of the environment as input and output motor actions. To overcome the high-dimensionality of the policies as well as the large state space, we ensure only interesting states are explored via intermediate rewards that guide the policy to high reward states.

Our results demonstrate that DQNs can be used to learn policies for a task that involves locating a cube, grasping, and then finally lifting. The agent is able to generalise to a range of starting joint configurations as well as starting cube positions. Moreover, we show that policies trained via simulation are able to be directly applied to real-world equivalents without any further training. To the best of our knowledge, this paper is the first to successfully learn robot control through deep Q-learning using a 3D robot simulation, and the first to transfer these policies to real-world hardware.

1.1 Objectives

Our goal was to research the feasibility of using 3D simulations to training a six-joint robotic arm in an object manipulation task without any prior knowledge, using only

images as input. Achieving this would open up the possibility of transferring knowledge gained from a simulation to a physical robot without any additional training. In order to accomplish this, we had to:

1. **Understand existing solutions.** The first aim was to get a good understanding of the underlying area of deep reinforcement learning. This required detailed knowledge of core concepts in both deep learning and reinforcement learning. Once confident with these core concepts, related state-of-the-art work needed to be examined and compared to our proposed work.
2. **Explore machine learning libraries and robot simulations.** A large proportion of time was spent developing and running experiments using different techniques. It was therefore necessary to explore a range of machine learning libraries and robot simulations that would be most beneficial to us.
3. **Implement a training framework.** A training framework that implements our deep reinforcement learning solution needed to be created. This framework needed to be developed such that swapping from the simulation to real-world required little effort.
4. **Experiment iteration.** We looked to evaluate how different reinforcement learning techniques are suited to solving the problem. This required us to iteratively run experiments that would either succeed or fail, evaluate the results to form a hypothesis, and finally improve the system based on these.
5. **Communicate with real-world robot.** Ultimately, we hoped that agents trained in simulation could be transferred to a physical robot. This required us to develop an interface to the real-world robot that could interpret commands from the simulation-trained network.

1.2 Challenges

The work surrounding this project was very challenging due to its focus on research and development of state-of-the-art deep reinforcement learning methods. The work entailed gaining a full understanding and appreciation of the field with almost no prior experience. During the course of the project, the biggest challenges encountered were:

1. **Long training times.** A single experiment would usually run for many days before producing any conclusive results. This makes progress slow, especially when experiments crash due to bugs or our machine being powered down unexpectedly. Moreover, due to the demand on machines in the college lab, it was not possible to reserve more than one machine — resulting in the inability to run multiple experiments to improve productivity.
2. **Experiment planning.** Due to the long training times, it was often challenging to decide what experiments would be most beneficial to run. Making large changes from one experiment to another would make it difficult to know why an experiment failed, yet on the other hand too little of a change would result in slow iteration progress.

3. **Handling the large state space.** With approximately *4 quadrillion* possible robot configurations and a state space of $256^{64\times64}$ states, it is challenging to encourage agents to explore only the interesting areas of this vast space. We have to balance the exploration-exploitation tradeoffs correctly to ensure our agent learns policies to complete the task, but does not take too long to train.
4. **TensorFlow.** Google's new machine learning library — TensorFlow, was used to build and train our neural networks. As this is a new library, the small community is still growing and tutorials on specific features are lacking or missing. This made development more difficult than choosing a well established library that has been in production for years. Having said that, we are happy to have had the chance to test this developing library in an advanced setting.

1.3 Contributions

We have demonstrated that DQNs have the ability to be used for learning control policies for robot manipulation using images as input. Along the way we have tried many variations that we hope will be insightful to future work. Our key contributions are:

1. **Learning robot control policies through simulation.** We present a deep Q-learning system for object manipulation trained in a 3D simulator. We train and evaluate our agents in simulation and show that learning policies from 3D simulation is possible.
2. **Transferring from simulation to real-world.** As well as evaluating agents in simulation, we also evaluate them in the real-world without any additional training. Ours is the first work that attempts to apply a policy trained in a 3D simulation to the real-world and show that this is possible.
3. **Generalisation of agents.** Experiments are trained with variations in robot configuration and target positions. The agents are able not only to generalise to unseen states, but also to slight variations in the target dimensions and ignore the introduction of small amounts of clutter into the scene.
4. **3D Mico arm simulator.** We develop a 3D robotic arm simulation that replicates a 6-joint Mico arm. The simulation is fully customisable and allows for faster training than our earlier third-party simulator.
5. **Learning from images alone.** Related work in learning control policies concatenate joint angles into the final stages of the network. Our work uses images alone as input, and we show that adding in joint angles alongside the image does not make learning any faster than using images alone.
6. $\mathcal{O}$-$Greedy$ **algorithm.** We propose a novel learning algorithm to ensure the agents explores interesting areas of the state space. We show that this method reduces training time for the agent.

1.4 Accompanying Paper

A portion of the work presented in chapter 6 is also present in a paper by *S. James* and *E. Johns* that will be submitted to IEEE Robotics and Automation Letters (RA-L).

1.5 Outline of Contents

The rest of this report is organised as follows. Chapter 2 introduces the background material covering the fields of reinforcement learning, artificial neural networks, and robot simulation. We then cover relevant work in these respective fields. Chapter 3 discusses the options available when choosing the technologies to use in this project. We highlight tradeoffs of each technology, and make our decision based on these. In chapter 4, we define our work in terms of the reinforcement learning problem. We describe our state and action space, as well as giving an overview of the network and learning process.

Experimentation is split into two chapters. Chapter 5 contains our early work with a third-party simulator. Although not contributing as much as the following chapter, these experiments were an important part in gaining a deeper understanding of the field, and vital experience from which to build. Following that, chapter 6 introduces a custom-built simulator that was used in the remainder of the experiments. This chapter contains the majority of our contributions and results. Chapter 7 contains the evaluation of our project — giving a critical appraisal of our work followed by a comparison to existing solutions. Finally, chapter 8 concludes the paper with some lessons learnt and some suggestions for future work.

Chapter 2

Background

To fully understand deep reinforcement learning, one must be familiar with concepts in both deep learning and reinforcement learning — both of which are vast fields in themselves. In this chapter, we introduce core concepts that the rest of this paper is based on, followed by a detailed discussion of relevant work in the field.

2.1 Reinforcement Learning

Reinforcement learning [93] falls under the wider field of machine learning. Inspired by behavioural psychology, it allows an *agent* — the learner and decision maker, to autonomously discover optimal behaviour through trial and error interactions with its surrounding environment in an attempt to solve the problems of control. The *environment* is defined as everything outside of the agent that can be interacted with, while a learning *task* is the complete specification of the environment.

In a given time step t, both the agent and environment can be modelled as being in a state $s \in \mathcal{S}$, which contain all relevant information about the current situation, e.g. a position in a navigation task. From this state, action $a \in \mathcal{A}$ can be performed. Both s and a can be members of either discrete or continuous sets. Upon advancing to the next time step, the agent receives a reward $r \in \mathcal{R}$, and transfers to the next state. A mapping from states to actions is given by a policy π. Policies can be deterministic — where the exact same action is used for a given state, or probabilistic — where the action is chosen through drawing a sample from a distribution over actions for a given state. The reinforcement learning framework defined above can be summarized in the figure below.

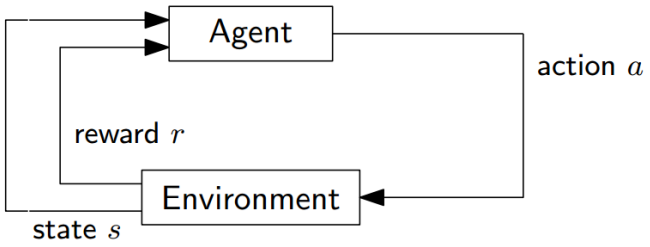


FIGURE 2.1: The reinforcement learning framework.

Within a task, learning is split into *episodes*. An episodic control task will usually use a finite horizon model, in which a task runs for a number of time-steps before being reset

and restarted. The number of time-steps may be arbitrarily large, but the expected reward over the episode must converge.

2.1.1 Returns

So far we have mentioned that upon advancing to the next time step, the agent receives a reward. Informally, the agent's goal is to maximise the cumulative rewards it receives in the long run. In general, the agent wishes to maximise its *expected return*, which in the simplest case is the sum of rewards:

$$R_t = r_{t+1} + r_{t+2} + ... + t_T, \tag{2.1}$$

where T is the final time-step. Here the final-time step is a natural break point at the end of a sequence which puts the agent in a special state called the *terminal state* — signalling the end of an *episode*. Following this, the agent is reset to a standard starting state or a state sampled from a standard distribution of starting states. Tasks such as this are called *episodic tasks*.

Conversely, we have *continuing tasks* where the agent-environment interactions go on continuously without any natural breaks. In tasks such as these, we have $T = \infty$, and by definition of equation (2.1), we could also have $R_t = \infty$.

As we can see, the expected return as defined in equation (2.1) may not be suitable for all tasks. Moreover, it seems that rewards gained now are worth just as much as rewards gained in the future. This brings us to the concept of *discounting*. We introduce a parameter γ that ranges $0 \leq \gamma \leq 1$, called the *discount rate*. This is used to determine the present value of future rewards the agent may receive. Adding the discount rate to our expected return, gives us the new *expected discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{N} \gamma^k r_{t+k+1}. \tag{2.2}$$

This equation tells us, that at any given time step k in the future, the reward is weighted only γ^{k-1} times what it would be worth if we were to receive it presently.

Great care must be taken when choosing an appropriate value for γ as it often qualitatively changes the form of the optimal solution [41]. As γ approaches 0, the agent becomes myopic, only choosing actions that maximize its immediate reward, leading to poor performance in long term tasks. On the contrary, as γ approaches 1, the agent becomes more far-sighted, leading to the problem that is cannot distinguish between policies that immediately gain a large amount of reward, and those that gain a reward in the future.

We are now able to unify the return as a sum over a finite number of terms from equation (2.1) and the return as a sum over an infinite number of terms from equation (2.2) to obtain a single notation that covers both episodic and continuing tasks. This is done by defining episode termination as entering a special *absorbing state* that transitions to itself and generates a reward of 0.

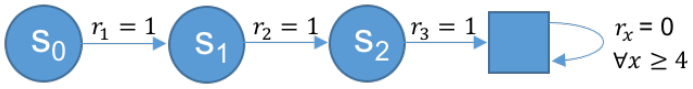


FIGURE 2.2: State transition diagram.

In the diagram above, states are represented via circles and the absorbing state is represented via a square. It shows us that starting from state s_0, we receive the reward sequence: $1, 1, 1, 0, 0...$, which upon summing over the first $T \geq 3$ rewards gives the same return as summing over the infinite sequence.

2.1.2 Markov Property

In reinforcement learning systems, we do not expect the state to inform the agent about everything in the environment. Moreover, we do not fault the agent for not knowing something that matters, but we do fault the agent for forgetting something. Therefore in an ideal system we would like a state signal that summarizes past sensations in a compact way, while retaining all relevant information. A state signal that preserves this information is said to be a *Markov Property*.

Formally, a state s_t is said to be *Markov*, if and only if:

$$P(s_{t+1}, r_{t+1}|s_t, a_t) = P(s_{t+1}, r_{t+1}|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ..., r_1, s_0, a_0). \tag{2.3}$$

This states that the environment's response at $t+1$ depends only on the state and action at time t. That is, the future is independent of the past given the present state s_t, such that s_t captures all information of past agent events.

2.1.3 Markov Decision Process

A reinforcement learning task that satisfies the Markov property in a continuous state and action space is said to be a *Markov Decision Process (MDP)* whilst one in a finite state and action space is said to be a *finite Markov Decision Process (finite MDP)*.

Finite MDPs are defined by their state, action sets, and the one-step dynamics of the environment. For any given state s, and action a_t, the *transition probability* from s_t to s_{t+1} is defined as:

$$\mathcal{P}^a_{ss'} = P(s_{t+1}|s_t, a_t). \tag{2.4}$$

Moreover, given state s_t, action a_t, and next state s_{t+1}, the expected value of the next reward is defined as:

$$\mathcal{R}^a_{ss'} = E(r_{i+1}|s_t, a_t, s_{t+1}). \tag{2.5}$$

2.1.4 Value Functions

When an agent enters into a new state, it must know how valuable it is to be in this state. The value of a state can be measured in two ways — *state-value functions* and *action-value functions*. The *state-value function* for a policy π is the expected return when starting in a state s and following π thereafter and is defined as:

$$V^{\pi}(s) = E_{\pi}(R_t|s_t = s) = E_{\pi}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s). \tag{2.6}$$

Here, E_{π} is defined as the expected value of following policy π.

The *action-value function* for a policy π, is the expected return when starting in state s, taking action a, and following π thereafter, and is defined as:

$$Q^{\pi}(s,a) = E_{\pi}(R_t|s_t = s, a_t = a) = E_{\pi}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a). \tag{2.7}$$

A fundamental property of value functions is the ability to satisfy a set of recursive consistency equations called the *Bellman equation*:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}(R_t|s_t = s) \\ &= E_{\pi}(r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s) \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma E_{\pi}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s')] \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')]. \end{aligned} \tag{2.8}$$

The final equation in the derivation above expresses a relationship between the value of a state and the value of the successor state. It allows us to average over all the possibilities, weighting each by its probability of occurring. Moreover, we can see that the value of the first state must equal the reward for making a transition to s' together with the discounted value of s'.

2.1.5 Optimal Value Functions

Recall that a policy π is a mapping from states to actions, and that agents attempt to maximize their expected return. We therefore seek to find a policy that gives us a set of actions that maximizes our expected return. If we find a policy π' that gives an expected return greater than or equal to that of π for all states, then π' is said to be better than or equal to π. Formally:

$$\pi' \geq \pi \text{ iff } V^{\pi'}(s) \geq V^{\pi}(s). \tag{2.9}$$

We define the *optimal policies* π^*, as the policies that are better than, or equal to all other policies. This now gives us the ability to define *optimal value functions*, which assign to each state, or state-action pair, the largest return achievable by any policy. This allows us to define the *optimal state-value function*:

$$V^*(s) = \max_\pi V^\pi(s), \tag{2.10}$$

and the *optimal action-value function*:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{2.11}$$

Therefore, any policy that is greedy with respect to the optimal value function V^*, is in fact an optimal policy.

Given the Bellman equation in (2.8), and that V^* is simply a value function for a policy, we are able to show that V^* constancy condition can be written in a special form without reference to any specific policy. This gives rise to the *Bellman optimality equation* for V^*:

$$\begin{aligned} V^\pi(s) &= \max_a Q^{\pi^*}(s,a) \\ &= \max_a E_{\pi^*}(R_t|s_t = s, a_t = a) \\ &= \max_a E_{\pi^*}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a) \\ &= \max_a E_{\pi^*}[r_{t+1} + \gamma E_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s, a_t = a)] \\ &= \max_a E[r_{t+1} + \gamma V^\pi(s_{t+1}|s_t = s, , a_t = a)] \\ &= \max_a \sum_{s}^{\prime} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')]. \end{aligned} \tag{2.12}$$

Similarly, it follows that the *Bellman optimality equation* for Q^* is defined as:

$$Q^\pi(s,a) = \sum_{s}^{\prime} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')]. \tag{2.13}$$

Given an environment with N states and known dynamics, we are able to use the Bellman optimality equations to solve the value of V^* for each state. From V^*, we have shown that it is relatively easy to determine an optimal policy.

2.1.6 Dynamic Programming

Reinforcement learning optimisation methods can be divided into two broad classes: *model-based* and *model-free*. Model-based methods assume there exists an internal model

whose parameters must be estimated. Appropriate actions are then chosen by searching or planning in this estimated model. Model-free methods use experience to learn simpler quantities, such as state or action values, without estimating the model.

We now focus on *Dynamic Programming (DP)* — a model-based method, specifically on that of *policy iteration* and *value iteration*. These two methods can be used to reliably compute optimal policies and value functions, and are both obtained via alternating between the two computations of *policy evaluation* and *policy improvement*.

We may first apply a round of policy evaluation, which determines the iterative computation of the value function for the current policy. We then switch to a round of policy improvement, which determines the computation of an improved policy by greedily selecting the best action according to the value function. The above process then repeats until the policy does not change any longer.

In policy iteration, the policy improvement is run only once the policy evaluation step has run. In contrast, value iteration only runs a single iteration of policy evaluation in between each policy improvement step.

We use *general policy iteration* to refer to the general idea of policy iteration, which can be summarised in Figure 2.3.

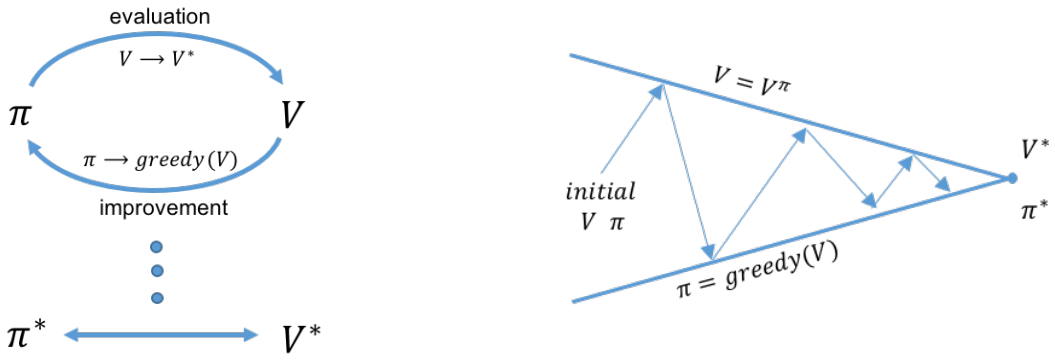


FIGURE 2.3: Generalised policy iteration.

DP allows for a technique called *bootstrapping*. Bootstrapping allows us to update estimates of the values for each state based on the estimates of successor states.

2.1.7 Monte Carlo Methods

All of the methods discussed so far require us to have complete knowledge of the environment, which is not the case for our project. Instead, we look at *Monte Carlo (MC)* methods which are model-free methods that allow us to learn from sample sequences of states, actions, and rewards from the environment. This works well for our use-case as we require no prior knowledge of the environment's dynamics.

MC methods work by averaging the sample returns from the environment on an episode-by-episode basis. Notice that this is different to dynamic programming methods, in that they update on a step-by-step basis. MC is best demonstrated through the code below:

LISTING 2.1: First-vist MC.

```
Let π be the policy to evaluate
Let V be an arbitrary state-value function
Let Returns(s) be an empty list, for all states
Repeat forever:
    Generate an episode from π
    For each state s appearing in the episode:
        R ← return following first occurence of s
        Append R to Returns(s)
        V(s) ← average(Returns(s))
```

Maintaining sufficient exploration is an issue in MC control methods. The agent must consider the *exploration-exploitation tradeoff* in order to gain information about the rewards and the environment. It needs to explore by considering both previously unused actions, and uncertain actions that could lead to negative rewards. The decision must be made to play it safe and stick to well-known rewards, or risk trying new things in order to discover even higher rewards.

In general, two approaches that can be used to solve this are *on-policy* methods and *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, while still exploring. Off-policy methods on the other hand, attempt to learn a deterministic policy that may be unrelated to the policy used to make decisions.

One common form of on-policy methods are ϵ-greedy policies. These are a form of *soft* policy ($\pi(s,a) > 0, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$) where most of the time an action that has maximal estimated action value is chosen, but with probability ϵ, we choose an action at random. This is summarised in the equation below:

$$\text{Let } a^* = \arg\max_a Q(s,a)$$

$$\pi(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = a^* \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq a^* \end{cases} \tag{2.14}$$

2.1.8 Temporal-Difference Learning

Temporal-difference (TD) learning is a combination of DP — the ability to learn through bootstrapping, and MC — the ability to learn directly from samples taken from the environment without access to the MDP.

Unlike MC methods, TD does not have to wait until the end of the episode to update the value function. Instead, TD methods only wait until the next time-step by using *temporal errors* to inform us how different the new value is from the old prediction. This update has the general form of:

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]. \tag{2.15}$$

The simplest TD method is known as TD(0), and is defined as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{2.16}$$

TD methods are a powerful combination of the sampling of MC and the bootstrapping of DP. Their model-free nature offers a big advantage over DP methods, whilst their on-line, fully incremental updates improve upon MC methods. This is especially important when dealing with long, possibly infinite episodes, so delaying updates until an episode's end would not be ideal.

We discuss 2 important TD control algorithms below.

2.1.9 Sarsa

Sarsa is a on-policy TD control algorithm which stands for State-Action-Reward-State-Action. This name is derived from an experience (s, a, r, s', a'), in which the agent starts in state s, performs action a, receives a reward r, transfers to state s', and then decides to do action a'. This experience is used to update $Q(s, a)$ using the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.17}$$

The general form of the algorithm is given below:

LISTING 2.2: Sarsa on-policy TD control algorithm.

```
Initialize Q(s,a) with random values
For e in episodes:
    Initialize s
    Choose a from s using policy derived from Q (e.g. ϵ-greedy)
    For each step of episode:
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
        Q(s,a) ← Q(s,a) + α[r + γQ(s',a') − Q(s,a)]
        s ← s'
        a ← a'
```

2.1.10 Q-learning

Conversly to Sarsa, Q-learning [100] is an off-policy TD control algorithm which directly approximate Q^* independent of the policy being followed. An experience is defined as (s, a, r, s'), in which the agent starts in state s, performs action a, receives a reward r, and transfers to state s'. The update to $Q(s, a)$ is then performed by getting the maximum possible reward for an action from s' and applying the following update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma \text{max}_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{2.18}$$

Q-learning has been proven to eventually find an optimal policy given any finite MDP as long as there is no bound on the number of times it tries an action in any state. The Q-learning algorithm is given below:

LISTING 2.3: Q-learning off-policy TD control algorithm.

```
Initialize Q(s,a) with random values
For e in episodes:
    Initialize s
    For each step of episode:
        Choose a from s using policy derived from Q (e.g. ϵ−greedy)
        Take action a, observe r, s′
        Q(s,a) ← Q(s,a) + α[r + γmax_a′ Q(s′,a′) − Q(s,a)]
        s ← s′
```

A variant to Q-learning is used heavily in our work and is discussed in detail in section 2.4.1.

2.1.11 Comparing Sarsa and Q-learning

We have just discussed 2 control methods: Sarsa (on-policy) and Q-learning (off-policy). The differences between these two methods are quite subtle. As Sara is an on-policy method, it means that it follows a control policy when making moves that will be used to update the Q-values. Q-learning on the other hand, is a off-policy method which assumes that the optimal policy is being followed, and so always takes the best action. To summarise, the main difference is in the way the future rewards are found.

The difference between these two methods is illustrated well in the example from Sutton and Barto's book [93].

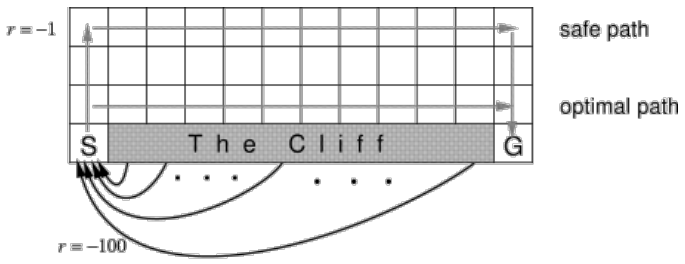


FIGURE 2.4: Grid-world of the cliff-walking task from [93].

The grid-world shown in figure 2.4 is part of a undiscounted ($\gamma = 1$), episodic task. The goal of the task is to go from the start state (S) to the goal state (G) using the actions up, down, right, and left, without falling off the cliff.

The agent receives a reward of -1 on each state transition except for when it enters the region marked "The Cliff", where it instead receives a reward of -100 and then is sent back to the start state. While performing the task, the agent follows a ϵ-$greedy$ action selection, with a constant epsilon set at $\epsilon = 0.1$.

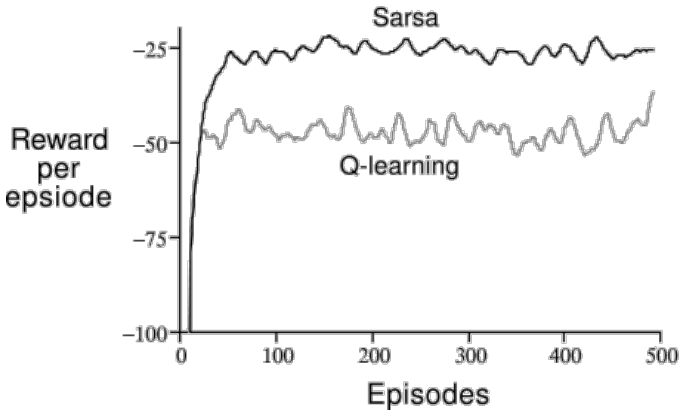


FIGURE 2.5: Results of the cliff-walking task from [93]. The graph shows the total reward for each episode when following a Q-learning control method and a Sarsa control method.

After a short time, Q-learning manages to learn the optimal policy, which involves travelling along the edge of the cliff — despite sometimes leading to a random action that would push the agent off the cliff by following the ϵ-$greedy$ action selection. Conversely, Sarsa takes this into account by considering the action selection method , resulting in a policy that follows a longer but safer path away from the cliff. Despite Q-learning finding the optimal policy, its performance is worse than Sarsa's, though both would converge to the optimal policy if ϵ were gradually reduced to 0.

2.1.12 Function Approximation

Many of the challenges of reinforcement learning become apparent when applied to robotics. This is mostly due to the inherently continuous states and actions of most robots. When dealing with tasks such as this, we face the *curse of dimensionality* [10]. This tells us that the number of states grows exponentially with the number of state variables.

All of the methods discussed so far have had the value functions represented as table with one entry for each state. The question is: what happens when we are given a large, or even continuous state space, such as the case with our robot arm? With a continuous state space, comes large, memory consuming tables that require a large amount of time and data. Therefore, the only way to learn in situations like these, is to generalise from previous states to ones that we have not seen before.

This brings us to the idea of *function approximation*, a sub-field of supervised learning that allows us to take samples from a function, just like the value function, and generalise an estimate to the entire function. We cover a specific type of function approximation called *artificial neural networks* in section 2.2.

With function approximation, we aim to represent the value function at time t, V_t, not as a table, but as a parametrised functional form with parameter vector $\vec{\theta_t}$. For example, $\vec{\theta_t}$ could represent the vector of connection weights in a artificial neural network that approximates V_t.

This concludes our discussion of reinforcement learning, and brings us to the next topic of artificial neural networks. We refer the reader to Sutton and Barto's book [93] for a complete review of reinforcement learning.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are quite possibly one of the greatest computational tools currently available for solving complex problems. These biologically-inspired models can be defined as a collection of densely interconnected processing units called neurons, that work in unison to perform massively parallel computations.

What makes ANNs particularly appealing is the possibility of mimicking many of the desirable characteristics of the human brain. These include massive parallelism, distributed representation and computation, learning ability, generalization ability, adaptivity, inherent contextual information processing, fault tolerance and low energy consumption [39].

The biological neuron, or nerve cell, is one of 100 billion in the human brain. Its main features consist of a cell body (soma) with branching dendrites (signal receivers) and a projection called an axon, which conduct the nerve signal. Dendrites receive electro-chemical impulses from other neurons which pass over the soma and then travel along the axon until reaching the axon terminals. Finally, the electro-chemical signal is then transmitted across the gap between the axon terminal and a receiving cell's dendrites — these gaps are called synapses.

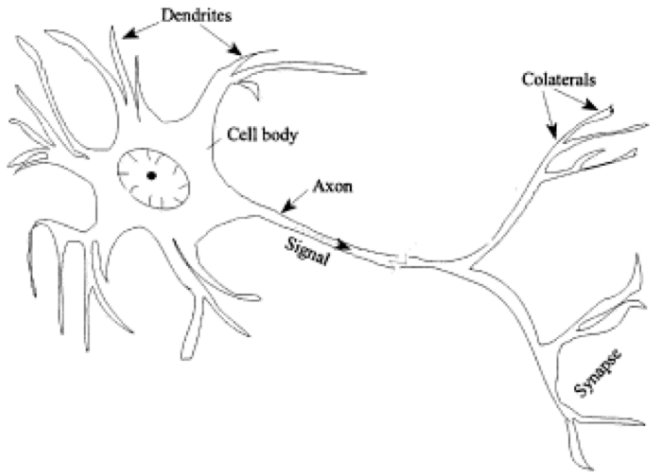


FIGURE 2.6: Schematic of a biological neuron [7].

Due to the large number of dendrites and synapses, it is able to receive many signals simultaneously. These signals alter the cells membrane potential which is governed by the intensity of the signal and the synaptic strength. These synapses can be adjusted by the signal passing through it, allowing them to learn the activity in which they participate.

Like its biological counterpart, the artificial neuron shown in figure 2.7, is a model with many inputs and one output.

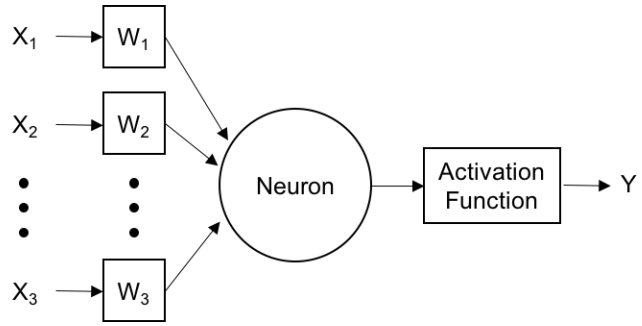


FIGURE 2.7: Artificial neuron model.

The neuron computes a weighted sum of its inputs to give:

$$\sum_{i=1}^{N} x_i w_i. \tag{2.19}$$

This can be seen as modelling the function of the dendrites. The soma is modelled via an activation function that determines when the neuron should fire. Common activation functions are step functions, sign functions, sigmoid function, and linear functions.

Neurons are arranged in a weighted directed graph with connections between the inputs and outputs. The two most common networks are feed-forward networks — which contain no loops, and recurrent networks — which contain feedback connections causing loops. The networks are organized according to layers, where each neuron's output in one layer is connected to every neuron's input in the next layer. Between the input and output layers exist the hidden layers, as illustrated in figure 2.8 — note that input layers do not count as a layer. These hidden nodes do not directly receive inputs or send outputs to the external environment.

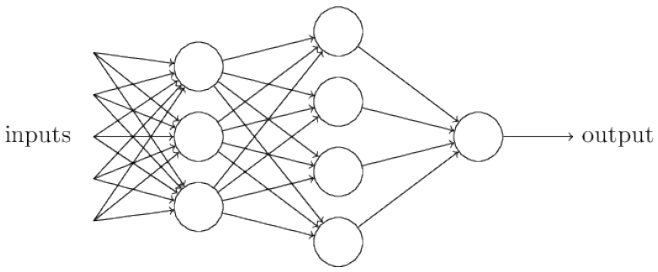


FIGURE 2.8: A 2 layer neural network.

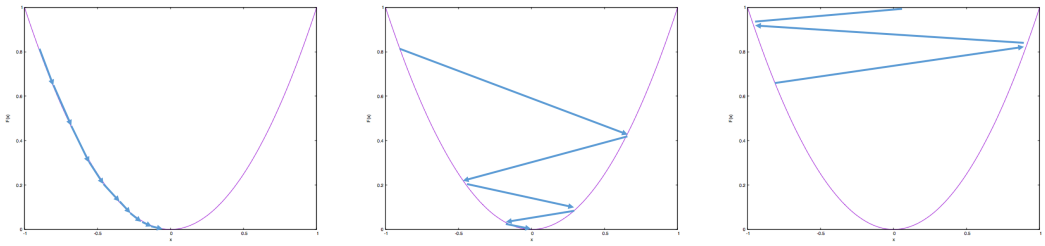
2.2.1 Gradient Descent

Gradient descent is a first-order optimisation algorithm and one of the most common ways to optimise neural networks. It is a way to find local minimum of a function by starting with an initial set of values and then iteratively stepping the solution in the negative direction of the gradient until it eventually converges to zero. We define such a function as a *loss function*.

Consider a loss function $F(w)$ with parameters w, each iteration we wish to update the weights in the direction that reduces the loss in order to minimise F. Taking the gradient of the curve will point to an increase in F, so we instead negate the inverse to lower the function value:

$$w_{i+1} = w_i - \eta \frac{\partial F}{\partial w_i}, \tag{2.20}$$

where η is the *learning rate* which determines the size of the steps taken to minimise the function. Great care must be taken when choosing a learning rate — too large will result in divergence, but too small will result in a long convergence time. Figure 2.9 illustrates the effect that different learning rate values have in comparison to the learning rate $\eta_{optimal}$ that would result in reaching the minimum in one step.



(A) $\eta < \eta_{optimal}$ (B) $\eta_{optimal} < \eta < 2\eta_{optimal}$ (C) $\eta > 2\eta_{optimal}$

FIGURE 2.9: The effect learning rate (η) has on convergence — where $\eta_{optimal}$ is the value of η that would result in reaching the minimum in one step.

Below we discuss three variants of gradient descent which differ in the amount of data needed to compute the gradient of the loss function. A tradeoff must be made depending on the amount of data available and time taken to perform an update.

Stochastic Gradient Descent

Stochastic gradient descent performs weight updates after the presentation of each training example. Through these frequent updates, local gradients are used to determine the direction to step, resulting in a fluctuating loss function that will on average move in the direction of the true gradient [102]. Stochastic gradient descent tends to be computationally fast due to its ability to hold its data in RAM. Moreover, it has the ability to follow curves in the error surface throughout each epoch, allowing the use of a larger learning rate which results in less iterations through training data.

Batch Gradient Descent

Batch gradient descent uses the entire training dataset to compute the gradient of the loss function. This method tends to be slower than stochastic gradient descent, especially with large training sets where it is often orders of maginitude slower [102].

Mini-Batch Gradient Descent

Mini-Batch gradient descent offers a hybrid of both stochastic and batch gradient descent, performing an update every n training examples. This method reduces the variance of weight updates which results in a more stable convergence than in stochastic gradient descent. Altering the batch-size n decides whether it will behave more like its stochastic version or its batch version — setting $n = 1$ results in stochastic training, while $n = N$ results in batch training, where N is the number of items in the training dataset.

Each of the 3 methods mentioned above have their strengths and weaknesses, but ultimately a choice should be made based on the expected loss function. As batch gradient descent uses the entire dataset, it is most applicable to situations where we expect a convex or smooth loss function, where we are able to move almost directly to a local or global minimum. Both stochastic gradient descent and mini-batch gradient descent (providing n is small enough) perform better when applied to functions with many local minima. The smaller number of training samples has the ability to nudge the model out of a local minima and potentially into a more optimal minima. Mini-batch gradient descent has the capability of averaging out the noise of stochastic gradient descent by reducing the amount of jerking that is caused by single sample updates. This makes mini-batch gradient descent an ideal balance between avoiding the local minima and finding the global minima.

2.2.2 Deep Learning

Recent trends has seen a move to deep neural networks. *Deep learning* [11] displays state-of-the-art performance in a number of fields and tasks. Such tasks include visual recognition [92] [51], face detection [73], audio recognition [53] [66], pedestrian detection [90], and natural language processing [17]. Deep learning's popularity comes from its power to learn useful features directly from supervised and unsupervised learning, avoiding the need for cumbersome and time-consuming hand-engineered feature selection.

Deep learning learning architectures have been categorised into three broad classes: generative, discriminative, and hybrid [21]. According to Li Deng, these are defined as follows:

Generative deep architectures are intended to characterize the high-order correlation properties of the observed or visible data for pattern analysis or synthesis purposes along with characterise the joint statistical distributions of the visible data and their associated classes.

Discriminative deep architectures are intended to directly provide discriminative power for pattern classification, often by characterizing the posterior distributions of classes conditioned on the visible data.

Hybrid deep architecture are intended to achieve discrimination but assisted with the outcomes of generative architectures via better optimization.

In our work, we focus on discriminative deep architectures, specifically convolutional neural networks.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) [52] use many of the same ideas of neural networks and have shown to eliminate the need for hand-crafted feature extraction in image analysis. CNN's differ in that of ANNs by taking into account the spatial structure of images — meaning that pixels which are far apart are treated differently than pixels that are close together. Today, deep convolutional networks, or some close variant, are used in most image recognition applications [92].

The architecture of CNNs are best explained through an example. A 10×10 image is a grid of pixel intensity values. In a CNN, we think of these as a grid of 10×10 neurons, as in figure 2.10

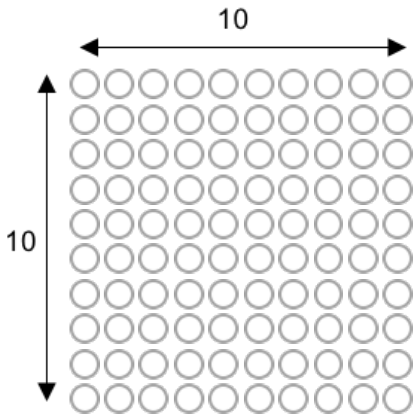


FIGURE 2.10: 10×10 input neurons.

In a standard ANN, this would be treated as a network with 100 (10×10) input neurons, with every neuron in the network being connected to every neuron in an adjacent layer. As mentioned above, this has the disadvantage that the spatial structure of the images is not represented. Instead, we only make connections in small, localized regions of the input image — called the *receptive field*. This can be treated as a small window, where each of the neurons in the receptive field is connected to one hidden neuron.

The receptive field is slid across the input image with each position of the receptive field mapping to a different hidden neuron. This continues until hitting the bottom right of the image. The amount the receptive field is slid each time is defined by the *stride length*. Figure 2.11 shows an example of a 3×3 receptive field with a stride length of 1.

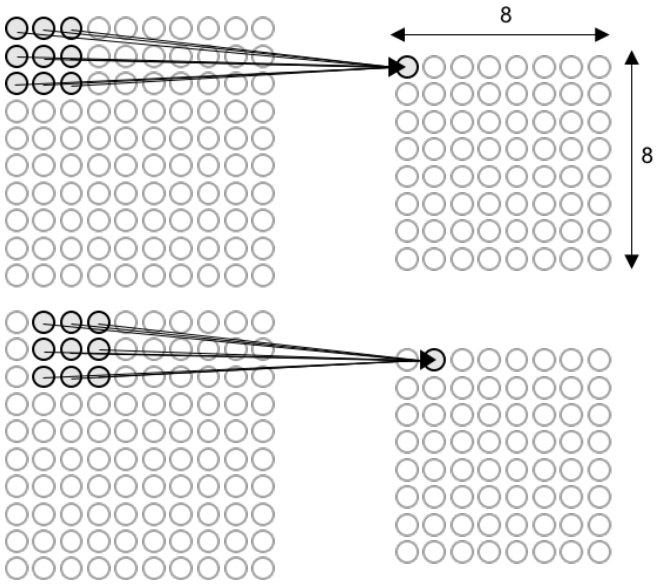


FIGURE 2.11: 3×3 receptive fields.

With a 10×10 input image and a 3×3 receptive field, we get an 8×8 hidden layer. Just like an ANN, each of the connections has an associated weight, meaning that each of the hidden neurons in an CNN has 9 (3×3) weights (and a bias) connected to it from its corresponding receptive field. Unlike an ANN, these weights and bias will be the same for each of the hidden neurons, giving rise to the names *shared weights* and *shared bias*. The shared weights and bias are often said to define a *kernel* or *filter*. This gives each hidden neuron an output of:

$$\sigma(b + \sum_{i=1}^{3} \sum_{j=1}^{3} w_{i,j} a_{k+i,l+j}) \tag{2.21}$$

Where σ is the activation function, b is the shared bias, $w_{i,j}$ is the shared weights of the 3 x 3 receptive field, and $a_{x,y}$ is the input activation at position (x, y).

Keeping the weights and bias the same means that the neurons in the hidden layer detect the same features at different locations in the image. This brings us to one of the fundamental properties of CNNs — *feature maps*. Feature maps are the map from one layer to the next layer, allowing us to learn features from the image instead of defining features manually as we would in an ANN.

Generally we would want to extract more than one feature from an image, requiring more feature maps. Thus, a convolutional layer itself consists of several feature maps, as shown in figure 2.12.

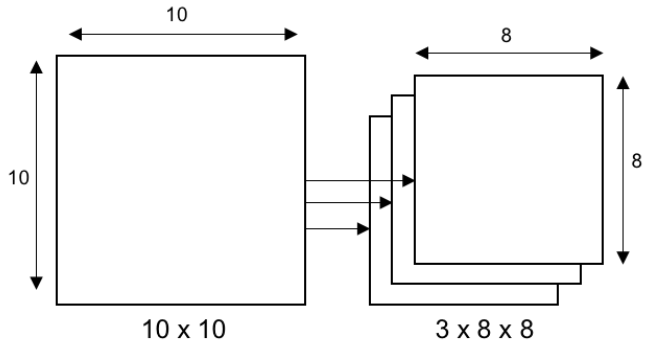


FIGURE 2.12: Multiple feature maps.

When sharing weights and biases we get a reduction in the number of parameters involved in a CNN in comparison to an ANN. In our example, each feature map needs 9 (3×3) shared weights and a shared bias, giving a total of 10 parameters for each feature map. Suppose we use 12 feature maps giving a total of 120 (12×10) parameters for the first convolutional layer. Now suppose using an ANN with 100 (10×10) input neurons representing the image, and 30 hidden neurons. With a fully connected first layer we would end up with 3,000 (30×100) weights, with an additional 30 biases, giving a total of 3,030 parameters — over 25 times the number for a CNN. This allows for faster training and construction of deep convolutional networks.

Pooling layers usually follow the convolutional layers and are responsible for taking each of the feature map's output's to create a condensed feature map. These pooling layers are similar to the convolutional layers except that they take input from the hidden layers and summarise a region of pixels, such as the 2×2 region in figure 2.13.

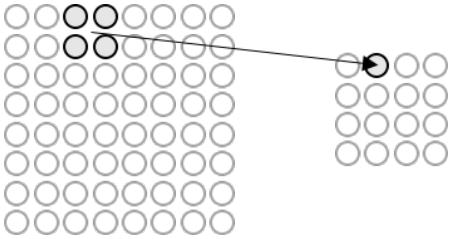


FIGURE 2.13: 2×2 pooling layer.

One example of a pooling layer is max-pooling, which involves taking the maximum activation in an input region (like a 2×2 region in figure 2.13). Max-pooling is useful in highlighting whether a feature is found anywhere in a region of the image. As with the convolutional layers, usually there are many feature maps, and so max-pooling is applied to each feature map separately.

The convolutional network usually ends with at least one fully connected layer. Every neuron from the final pooling layer is connected to every one of the fully connected layer — which could be the output layer at this point. Adding a pooling layer followed by a fully connected output layer to our ongoing example gives the final CNN in figure 2.14

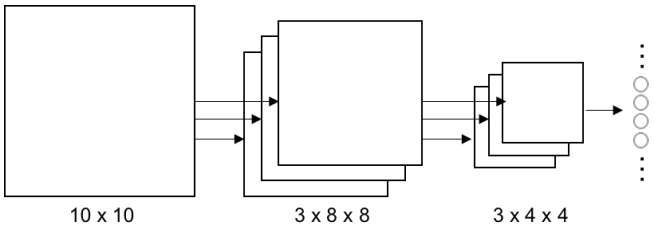


FIGURE 2.14: Convolutional layer, followed by a max pooling layer, and finally a fully connected layer.

In a typical network, there may be several layers of convolution and pooling before the final fully connected layers.

Overall, the performance of computer vision systems depends significantly on implementation details [16]. Generally, deeper networks perform better than shallow networks at a cost of more data and increased complexity of learning. Having said that, the dimensionality of the output layer can sometimes be reduced significantly without adverse effects on performance, as pointed out in a recent paper that found reducing the output layer from 4096 to 2048 actually resulted in a marginal performance boost [15] .

Tradeoffs must be made when deciding on the size of a filter. Generally small filters are used to allow for capturing very fine details of an image and to preserve the spatial resolution, while choosing large filters could miss out on these finer details.

Understanding how convolutional networks perform can give indications to how they might be improved. In figure 2.15, we show feature visualisations from a trained network by Zeiler and Fergus [106] on the *ImageNet* classification benchmark [37]. Layers 2 to 5 shows the top 9 activations in a random subset of feature maps across the validation data — revealing the different structures that excite a map and showing its invariance to input deformations. For example, looking at the collection of images on the right of the second layer, we see a group of images that contain people. The corresponding activations of the left then show excitement surrounding the face which allows the network to classify the image as containing a person with a high degree of certainty. Similarly in the 4th layer, we can see that the nose and tongue of a dog cause the highest amount of excitement when given an image of a dog.

FIGURE 2.15: Visualization of features in a fully trained model from Zeiler and Fergus [106]. The figure shows the top 9 activations in a random subset of feature maps across a validation set.

2.4 Related Work

There have been a number of developments in the fields relating to our work. This section aims to outline, analyse and compare these bodies of work, organising them according to the methods used. We first look at notable pieces of work in the reinforcement learning field in general before homing in on reinforcement learning in the context of robot learning. We discuss the details of robot grasping and manipulation, from traditional methods to current state-of-the-art. Finally we look at progress in reinforcement learning applied to robot simulations, before concluding with a brief mention of related miscellaneous work.

2.4.1 Reinforcement Learning

Over the past few decades, reinforcement learning has been an ideal solution for learning control policies to play video games. These type of tasks are well suited for reinforcement learning because of their episodic nature and reward structure in the form of game scores. Moreover, work in this area has recently been progressing to capturing state via images alone — something we also try to accomplish but in a different context.

In 1992, reinforcement learning was successfully applied to a backgammon playing program — TD-gammon [96]. Following the success of TD-gammon, additional attention was brought to the field and it remains one of the best-known success stories of reinforcement learning.

Following a number of unsuccessful attempts to re-produce TD-gammon's success, it was believed that its approach was a special case that only worked due to the co-evolutionary structure of the learning task and the dynamics of the backgammon game itself [80]. Progress was made when reinforcement learning was used to train an agent to play games on an Atari-2600 emulator, first with linear approximation and generic visual features [9] and then with the HyperNEAT evolutionary architecture [34].

2.4.2 Deep Q-learning

Recently, state-of-the-art results have been achieved by DeepMind Technologies, who successfully applied deep learning models to 49 Atari-2600 games, outperforming all previous approaches [64]. These agents were trained directly from RGB images using deep neural networks. The success of this approach was down to a novel variant of Q-learning specified in their earlier paper — *deep Q-learning* [65]. Parametrisation of Q was done using a convolutional neural network, which when combined with their approach, produces a so-called *deep Q-network*.

The Deep Q-learning algorithm presented below, utilises a technique known as *experience replay* [57]. With this technique, the agents experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is taken at each time-step t, and stored in a data-set $\mathcal{D} = e_t, e_{t+1}, ..., e_n$ called the *replay memory*. Training is then performed using mini-batch gradient descent by taking samples of experience drawn at random from this replay memory.

LISTING 2.4: Deep Q-learning with experience replay [65].

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialise sequence s_1 = (x_1) and preprocessed sequenced φ_1 = φ(s_1)
    for t = 1, T do
        With probability ε, select a random action a_t
        otherwise select a_t = max_a Q*(φ(s_t), a; θ)
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set s_{t+1} = s_t, a_t, x_{t+1} and preprocess φ_{t+1} = φ(s_{t+1})
        Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
        Sample random minibatch of transitions (φ_j, a_j, r_j, φ_{j+1}) from D
        if terminal φ_{j+1}:
            y_t = r_j
        else if non-terminal φ_{j+1}:
            y_t = r_j + γmax_{a'} Q(φ_{j+1}, a'; θ)
        Perform a gradient descent step on (y_j Q(φ_j, a_j; θ))^2
    end for
end for
```

The approaches used above have many advantages over the simpler Q-learning algorithm. The use of experience replay allows the potential for experiences to be used in more than one update, resulting in greater data efficiency, and tackles the re-learning problem highlighted in [57]. Moreover, randomly sampling from the replay memory breaks the correlation of experiences, therefore reducing the variance of updates.

Two years following the success of deep Q-learning, the DeepMind team presented the first massively distributed architecture for deep learning [70]. This architecture was used to implement the deep Q-network described above and managed to surpass the original paper in 41 out of 49 games, with speed-ups in the order of magnitude for most games.

2.4.3 Robot Manipulation And Grasping

For years, reinforcement learning has been heavily applied in robotics. Usually these tasks range from locomotion — including both bipedal [95] [24] [27] and quadrupedal [48], to sports — such as darts and table tennis [46]. In this section, we focus specifically on tasks involving object manipulation.

Object manipulation is said to be one of the oldest problems in the field of robotics [78]. Motives for success in this field vary — from a block stacking task done by an affordable robotic system that can play with children [20], to a water serving task that could offer assistance to the elderly [75].

Generally, object manipulation is split into two sub problems — *arm planning* and *grasp synthesis*. Arm planning refers to determining the best path that results in a collision free motion of the arm, though this path may not be ideal for grasping an object. Grasp synthesis refers to the problem of finding a grasp configuration that satisfies a set of criteria relevant for the grasping task [12]. This is a challenging problem because not only does it depends on the pose and configuration of the gripper, but also the shape of the object. Moreover, it does not take into account the feasibility of achieving a configuration suitable for the best grasp.

Advanced physical simulations — such as Graspit! [62], have been used for visualising 3D models of objects that can be used with the desired contact points for the gripper

to allow for grasp optimisation [26]. Another piece of work uses spaces of graspable objects that are mapped to new objects to discover their grasping locations [79]. In general, robots will not have access to the 3D model, but instead will rely on input from sensors, such as cameras, giving rise to limited and noisier data. To compensate for this, earlier work made simplifying assumptions such as assuming that objects are made of a set of primitive shapes [13] [77], or are planar [67].

In 1996 it was shown that learning approaches could be used to solve the grasping problem through vision [42]. Since then, advances in machine leaning have allowed for grasping objects that have not been seen before [87], have been partially occluded [28], or put in an unknown pose [22].

One impressive feat of robot grasping was the demonstration of a method in a household scenario in which a robot emptied a dishwasher [86]. A simple logistic regression model was trained on large amounts of synthetic, labelled training data to predict good grasping points in a monocular image.

In an attempt to increase the amount of training data available, an end-to-end self-supervising staged curriculum learning system has been developed which uses thousands of trial and error runs to train deep networks. These are then used to collect greater amounts of positive and negative data which helps the network learn faster [78]. In addition to this, the work presents a multi-stage learning approach where a convolutional neural net is used to collect hard negatives in subsequent stages.

Recent work has seen a shift away from focusing on one component of the overall control pipeline to instead focus on end-to-end solutions. Sergey Levine et al. introduced the first known method of training deep visuomotor policies to produce direct torque control for complex, high-dimensional manipulation skills [56]. This method succeeded to learn complex manipulation tasks such as block insertion into a shape sorting cube, screwing on a bottle cap, wedging the claw of a hammer under a nail, and placing a coat hanger on a clothes rack. The solution comprised of a 7 layer convolutional neural network with 92,000 parameters that was used to represent the policies. Monocular camera images are fed into 3 convolutional layers followed by a fourth spatial soft-max layer, with the responsibility of mapping pixel-wise features to feature points. The points are then concatenated with the robot configuration (consisting of joint angles, end-effector pose, and their velocities) and fed into the final 3 layers of a fully connected neural network to produce the resulting torque values. This can be summarised by their figure below:

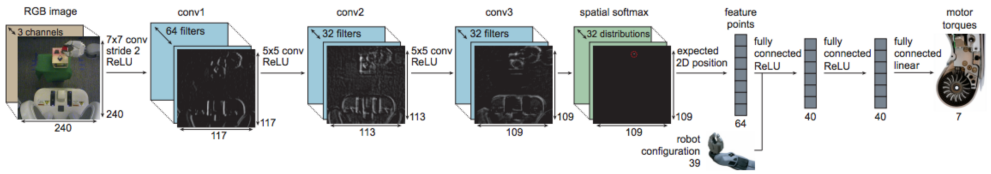


FIGURE 2.16: Visuomotor policy architecture from Sergey Levine et al. [56]

Training is done via partially observed guided policy search, which is an extension of earlier work on guided policy search — decomposing the problem into a trajectory optimization phase that generates training data for a supervised learning phase to train a policy [54].

The work of Sergey Levine et al. shows that significant improvements can be seen by using end-to-end training of deep visuomotor policies compared to using fixed vision layers trained for pose prediction. One of the downsides of this method, is its instrumental training set-up that is required when training on objects. In order to initialize the vision layers, the objects must be moved through a range of random positions. The recorded pose and camera images are then used to train a pose regression CNN. This data collection and training took 60-90 minutes, in comparison to the overall 3-4 hour training time for each visuomotor policy.

Further work improved on the downside highlighted above by removing the need for instrumental training set-up while still being able to learn a variety of manipulation skills that require hand-eye coordination [25]. These tasks consisted of pushing a free-standing toy block, picking up a bag of rice with a spatula, and hanging a loop of rope on a hook. This was achieved by using unsupervised learning with deep spatial auto encoders to learn a state representation of feature points, which correspond to object coordinates. These points are then combined in a trajectory-centric reinforcement learning algorithm. The proposed method consists of three stages. The first stage trains an initial controller using a combination of linear-Gaussian controllers and guided policy search [55] [46] [56]. This controller is then used to collect a data-set of images that are used to train a deep spatial auto-encoder. The encoder portion of this produces a vector of feature points that can then be concatenated with the velocities of the feature points, joint angles, and end-effector positions, to form the final state space. The final stage uses this new state space to train a vision-based controller using the same algorithm as in the first stage.

We are aware of only one attempt at mapping a deep Q-network from a robot simulation to real-world [107]. The paper's aim was to train a three-joint arm using a 2D simulator and then apply this trained network to a real-world scenario. Although the agent performed well when the network was tested in simulation, the network failed to transfer to the real world. Following this failure, the authors replaced real-world images with images from the simulation and sent the generated actions to the real-world robot — though we believe this is a trivial mapping that would naturally succeed if successful in simulation testing.

2.4.4 Robot Simulation

Simulations for learning policies for robots have many advantages over their real-world alternative. Within simulations, we are able to model continuous states and can train what would be an infeasible amount of iterations for a real-world system.

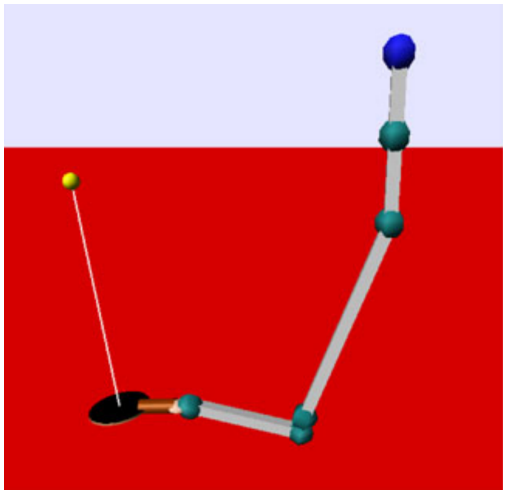
In an ideal setting, our approach would learn the required behaviour in a controlled 3D simulation, and then transfer the behaviour to a real robot. Unfortunately, this is far more challenging than it first seems. Differences between the simulation and real-world accumulate over time, meaning a small error in the model can lead to the simulated robot diverging rapidly from the real system. This can be mitigated by the use of a short horizon, or a very accurate simulation.

Transferring behaviours from simulations to real systems are easiest when we are dealing with macro-actions and a small state space, whilst highly dynamic tasks or systems are far more challenging. Tasks that involve manipulation and grasping, such as ours,

add to to these challenges especially when attempting to deal with soft body objects that are particularly hard to simulate — things like cloth, fluids, and elastics.

Scenarios with complex contact forces perform better when trained directly on a real-world robot [76] [47]. A study found that the learning task for a locomotion agent on rough terrain was significantly harder to train in a simulation, and the policy could not be transferred [76]. This would suggest that the same problem could occur when attempting to correctly simulate real world friction when grasping and picking up objects.

Another study showed that energy-absorbing scenarios (systems that absorb energy from the actions) in simulations are highly unpredictable [47]. This was shown via a paddle-ball system (illustrated in figure 2.17), where the aim was to keep a ping-pong ball attached to an elastic string in the air by repeatedly hitting it from below. The simulation was defined in great detail and used friction models, restitution models, dampening models, and air drag — but ultimately failed to reproduce the learning speed of a real-world system. The paper concludes that for energy-absorbing scenarios, a simulation may only transfer in a few cases, and therefore is only recommended for feasibility study and software debugging.



(A) Paddle-ball simulation

(B) Paddle-ball real-world system

FIGURE 2.17: Paddle-ball system in [47] which ultimately failed to reproduce the learning speed of a real-world system.

In addition to energy-absorbing scenarios, the paper explains that borderline scenarios, such as its ball-in-a-cup experiment, can use simulations to learn how to learn [47]. This means that the policy doesn't necessarily transfer from simulation to real-world, but parameters — such as learning and exploration rates, can instead be tuned using simulations.

In order to make the transition from simulation to real-world as simple as possible, studies have shown that simulations should be more stochastic than the real system [71], and secondly should be learned to make transferring the results easier [88].

2.4.5 Miscellaneous

Methods from computational neuroscience have also been explored for learning robot control. One target-reaching task was trained using a neural network model of the cerebellum (a region of the brain largely responsible for motor control) based on integrate-and-fire spiking neurons with conductance-based synapses [14]. The results showed that the cerebellum-based system successfully adapted to different contexts.

Chapter 3

Research Choices

Three important decisions needed to be made in the early stages of the project. We needed to choose a robot simulation tool that fulfilled all of our requirements, a machine learning library to construct and train our neural network, and a language that we would use to implement our reinforcement learning algorithms.

3.1 Robot Simulation

At this early stage in the project, we saw that there were a number of third-party simulations available and decided not to develop a simulation of our own in order to focus our time and effort on our main objectives. In the later stages of the project, we abandoned our original choice in simulation and built our own instead — the reasoning for this is discussed in section 6.2. We present our original search for a simulation below.

Ideally, we were looking for a simulation that:

- Offered flexibility in the choice of robots and environments.
- Allowed simulation parameters to be tweaked, such as the speed of the simulation and the physics of objects (e.g. mass, friction, etc).
- Was still supported by developers.
- Had an active community.
- Offered common language API's.
- Had a free license.

Microsoft Robotics Developer Studio

This free 3D simulated environment for robot control includes easy access to sensor and actuator data, a visual programing tool, and web-based interfaces [61]. The tool ships with several simulated environments, including an apartment, factory, house, and outdoor scenes. Being a Microsoft tool, programming is done in C#, which is generally not as popular for machine learning and reinforcement learning in comparison to Python and R. As of September 2014, support for the tool was suspended following Microsoft's restructuring plan.

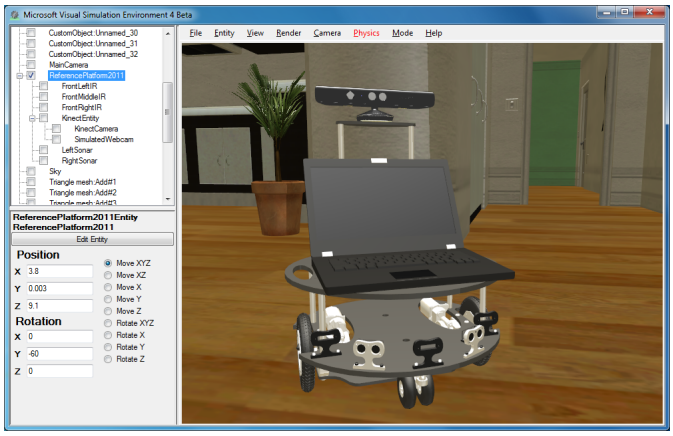


FIGURE 3.1: Microsoft Robotics Developer Studio.

V-REP

V-REP [99] describes itself as the Swiss Army Knife of robot simulators. This free tool has a large range of features with its own integrated development environment. V-REP is very versatile in that it allows objects to be individually controlled via internal scripts or in C/C++, Python, Java, Lua, Matlab, Octave, or Urbi. V-REP allows its users to change the physics engines used during simulation, offering great flexibility. It main uses are: fast algorithm development, factory automation simulations, fast prototyping and verification, robotics related education, remote monitoring, and safety double-checking, among others. Additionally, V-REP has good documentation and a fairly active community surrounding it.

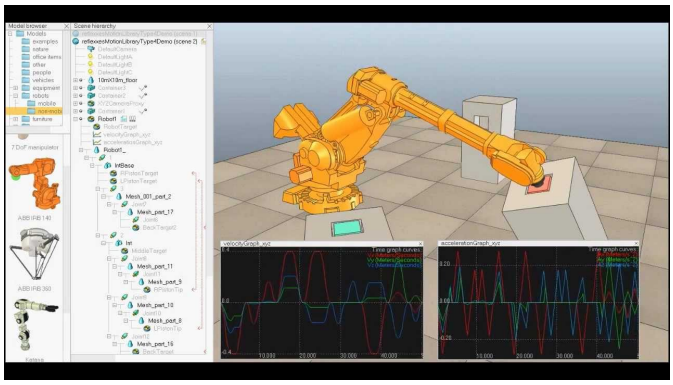


FIGURE 3.2: V-REP.

Robologix

Primarily used for teaching, Robologix [82] enables programmers to write their own movement sequences, modify the environment, and use the available sensors on a five-axis industrial robot. Unfortunately, this robot is the only one that is offered, severely damaging the flexibility for our use. Additionally, this tool seems more suited to younger audiences who are perhaps new to programming due to its simple programming model.

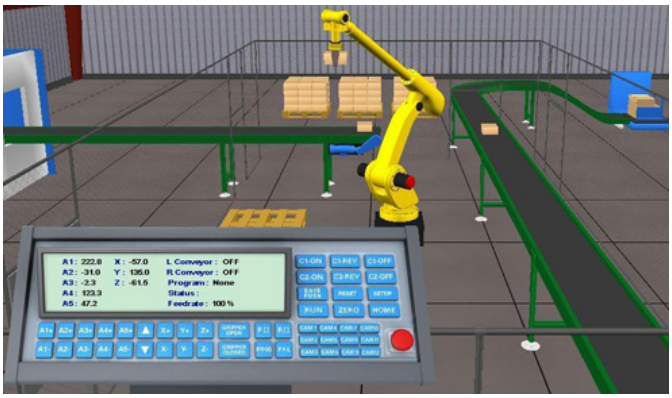


FIGURE 3.3: Robologix.

AnyKode Marilou

AnyKode Marilou [3] simulates environments for mobile robots, humanoids, articulated arms, and parallels robots operating in real-world conditions. They offer an engine that reproduces the behaviour of sensors and actuators with respect to real properties in a physical environment with an extremely high level of reality. They offer bindings in C/C++, VB, J#, and C#. Unfortunately, the education version comes with a license fee.

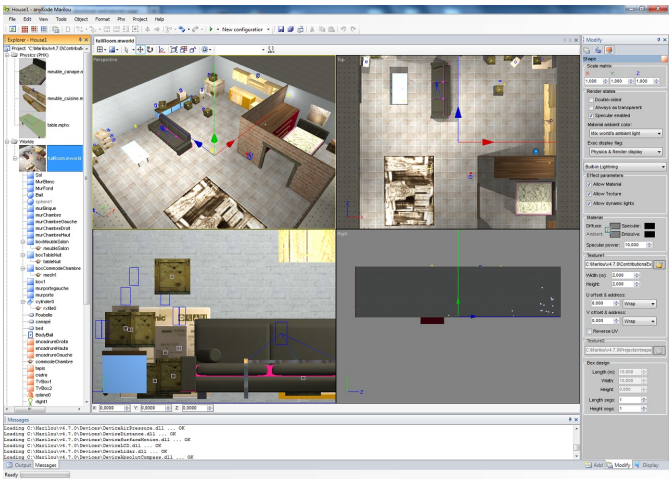


FIGURE 3.4: AnyKode Marilou.

Webots

Webots [101] is a commercial robot simulator which allows mobile robots can be modelled, programmed, and simulated in C, C++, Java, Python, Matlab, or URBI. It is one of the most popular simulations in use, catering more than 1200 companies, universities, and research centres worldwide for education and research. Although the software is free, robots and objects must be purchased separately.

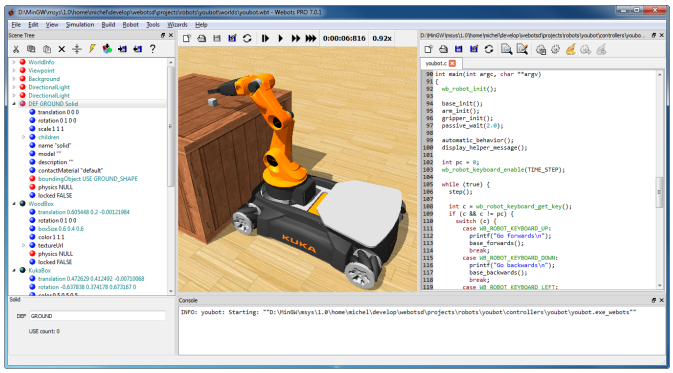


FIGURE 3.5: Webots.

Miscellaneous

Graspit!

Graspit! [62] is a tool designed for grasping research. It is a virtual environment for simulating robotic grasping tasks accompanied by a number of analysis and development tools. As the name suggest, there is more of an emphasis on grasping than an end-to-end manipulation task and the selection of models available seem to focus more on the hand rather than specific robots like the Mico arm. Moreover, the tool seems more directed towards more traditional approaches of grasping using active vision.

MuJoCo

MuJoCo [69] is a physics engine that aims to facilitate research and development in areas where fast and accurate simulation is needed. It is designed for model-based optimisation, and optimisation through contacts. This would be ideal to use as the physics engine if we were building a simulation form the ground up, but not as a stand-alone tool that covers both physics and life-like graphics.

Simulation discussion

V-REP, Microsoft Robotics Developer Studio, and Webots are the three that stand out from our review. Microsoft's tool would be a great choice considering its backing by such a big company, but since the tool is no longer supported, the community has begun to move to other tools, making it difficult to get support from others when running into problems.

Webots' popularity among researchers and companies makes it an attractive option, but this comes at a high licensee cost. Ultimately, V-REP meets all of our requirements, making it our choice over Webots, and thus the simulation we used going forward.

We have only discussed a small number of simulation tools, but during our reserach we came across a number of other publicly available tools that we have not discussed, such as ROBOGUIDE [81], MotoSim [68], Robot Expert [83], Robot Studio [84], Work

Space [104], Work Cell Simulator [103], AX On Desk [6], Robo Works [85], and Aristo Sim [4].

3.2 Deep Learning Libraries

With the increasing popularity of deep learning, there are now many machine learning libraries on offer. In this section we will discuss some of the more popular options available, which include Caffe [40], CNTK [105], TensorFlow [1], Theano [8], and Torch [97].

Caffe

Convolutional Architecture for Fast Feature Embedding (Caffe) provides multimedia scientists and practitioners with a clean and modifiable framework for state-of-the-art deep learning algorithms. The C++ framework offering both Python and MATLAB bindings, powers ongoing research projects, large-scale industrial applications, and start-up prototypes in vision, speech, and multimedia. The framework can be used for developing and training general purpose convolutional neural networks and other deep models efficiently on commodity architectures.

CNTK

The Computational Network Toolkit (CNTK) is a general solution software package by Microsoft Research for training and testing many kinds of neural networks. It provides a unified deep learning toolkit that describes neural networks as a series of computational steps via a directed graph. In this directed graph, leaf nodes represent input values or network parameters, while other nodes represent matrix operations upon their inputs. Networks are specified in a text configuration file which allows users to specify the type of network, the location of input data, and how to optimise the parameters.

The combination of CNTK and Azure GPU Lab allowed Microsoft to build and train deep neural nets for Cortana speech recognition which proved to be 10 times faster than their previous deep learning system [60]

TensorFlow

Developed by researchers at Google, *TensorFlow* is an interface for expressing machine learning algorithms. It has been used for research areas such as speech recognition, computer vision, robotics, information retrieval, natural language processing, graphic information extraction, and computational drug discovery. TensorFlow was based on work done by the Google Brain project, DistBelief [19], that was used in a range of products at Google, such as Google Search [31], Google Photos [38], Google Maps and Street View [29], Google Translate [35], and many more. TensorFlow can be used for flexible research experimenting, as well as high performance and robustness for production training of models.

Although the initial open-source release of TensorFlow supports multiple devices (CPUs and GPUs) in a single computer, Google have not released an open-source multi-machine version, though this will be released in the future.

Theano

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently, achieving speeds equal to hand-crafted C implementations. Theano pioneered the trend of using symbolic graphs for programming a network. Many high-level frameworks have been built on top of Theano, which make model definition and training simpler.

OpenAI

Co-founded by Elon Musk, OpenAI [72] is a new research company specialising in artificial intelligence. Their only released product to-date is *OpenAI Gym (Beta)* [72] — a toolkit for developing and comparing reinforcement learning algorithms. The tool is not yet general enough to be suitable for our work, but could prove to be very useful to the reinforcement learning community in the future.

Torch

Torch is a scientific computing framework built on top of the C/CUDA implementation, with wide support for machine learning algorithms. Networks are scripted using LuaJIT, which has been described to run incredibly fast [58].

Library discussion

When choosing which machine learning library to use, our main focuses came down to the modelling flexibility, interface, performance, maintenance, and future support. From our initial research, it became clear that all of the libraries had great modelling flexibility — possibly one of the key requirements of any machine learning library.

We now attempt to asses the performance of these libraries. It is surprisingly difficult to come across new benchmarks looking at all of these libraries. We look at two benchmarks for a selection of these libraries — one from December 2015, following the release of TensorFlow, and a more recent one from April 2016.

A blog by Microsoft from December 2015 [60] showed results of a performance test with the libraries highlighted above. This test included a fully connected 4-layer neural network run on two configurations on a single Linux machine with 1 and 4 Nvidia K40 GPUs, and a mini batch size of 8192. The results of this benchmark is shown below.

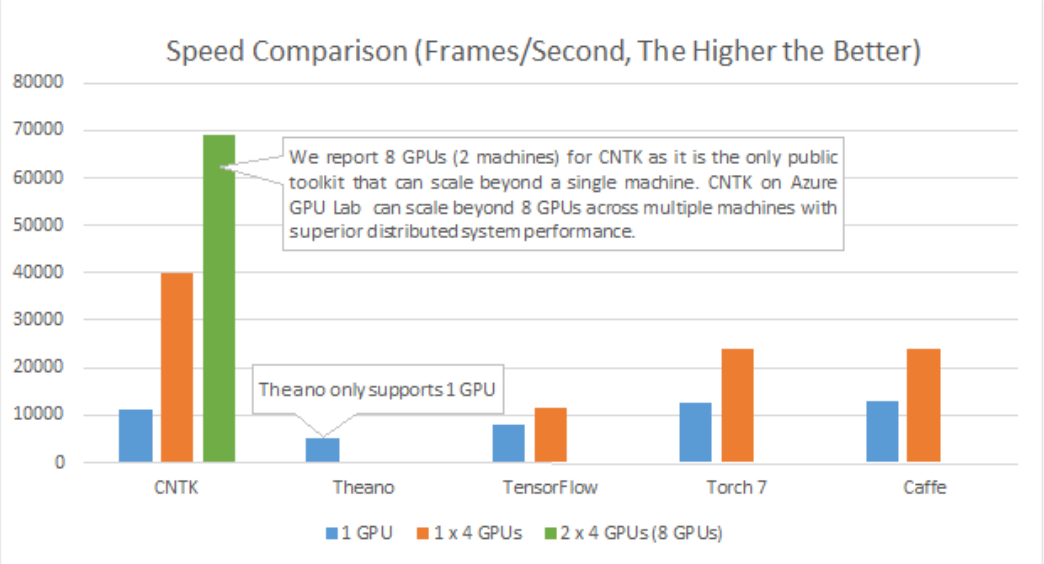
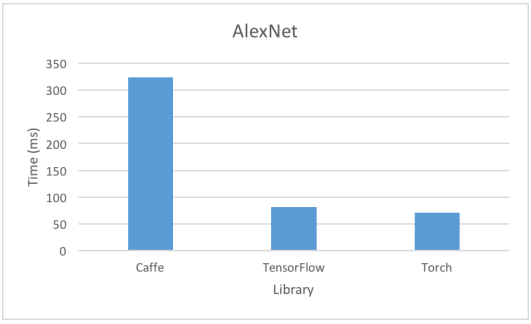


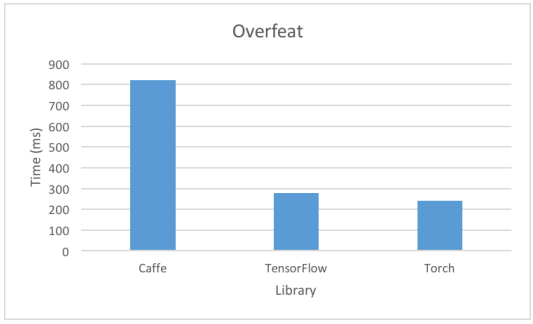
FIGURE 3.6: Benchmark for machine learning libraries from a Microsoft blog [60]. The higher the better.

Although not the worst, TensorFlow does surprisingly badly considering it's backed by such a big company. Moreover, the authors of TensorFlow include co-authors of Theano and Caffe — yet Caffe performs better. Having said that, this benchmark was performed during the early stages of release and Google themselves have acknowledged that the first release of TensorFlow is far from efficient.

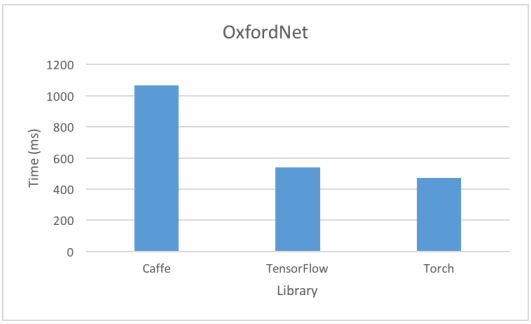
More recently there has been another benchmark that was performed in April 2016 [18] — but only contained 3 of the libraries discussed above. Benchmarking was performed on a system running *Ubuntu 14.04 x86_64* with a *6-core Intel Core i7-5930K CPU @ 3.50GHz* and a *NVIDIA Titan X* graphics card. Popular ImageNet models are selected, and the time for a full forward and backward pass are recorded.



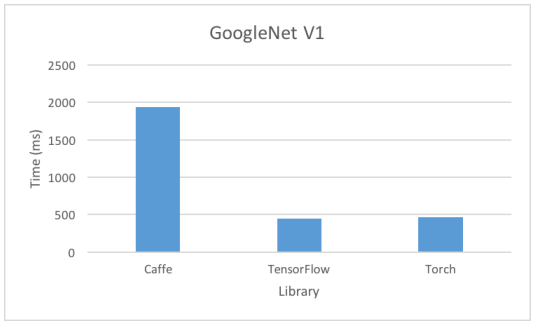
(A) AlexNet [49]



(B) Overfeat [89]



(C) OxfordNet (Model-A) [91]



(D) GoogleNet V1 [94]

FIGURE 3.7: Benchmark figures for 3 of the libraries we discuss: Caffe, TensorFlow, and Torch. Benchmark is performed by recording the time to perform a full forward and backward pass on some popular ImageNet models. The lower the better. Note that these benchmarks are misleading as both TensorFlow and Torch have been benchmarked with CuDNN, while Caffe has not. Data from [18].

The results from figure 3.7 show that Torch outperforms TensorFlow by a small margin on all models except GoogleNet. Caffe performs the worst on all models, though this is an unfair comparison as both TensorFlow and Torch are run with CuDNN, while Caffe is not.

As TensorFlow is a new library, its plausible to assume that their CuDNN implementation is still being tuned. Moreover, Google have been largely focusing on other parts of the system, for example the distributed module for the internal system. For these reasons, we argue that future versions of TensorFlow will be better, or on par with its competitors.

Google are investing large amounts of resources into the machine learning community. Recently they have a built a custom application-specific integrated circuit specifically for machine learning called a Tensor Processing Unit (TPU), and tailored for TensorFlow [30]. These TPUs have been found to deliver an order of magnitude better-optimized performance per watt for machine learning. This recent news makes future development for TensorFlow highly attractive. Due to TensorFlow's backing and potential to be an industry standard, we decided to use TensorFlow for the development of this project.

3.3 Language Choice

The language choice is an important decision as it acts as the glue between the simulation and the machine learning library. Below is a chart showing the popularity of languages for data science from a poll of 713 people from 2013. We now look at each of these languages in some more detail.

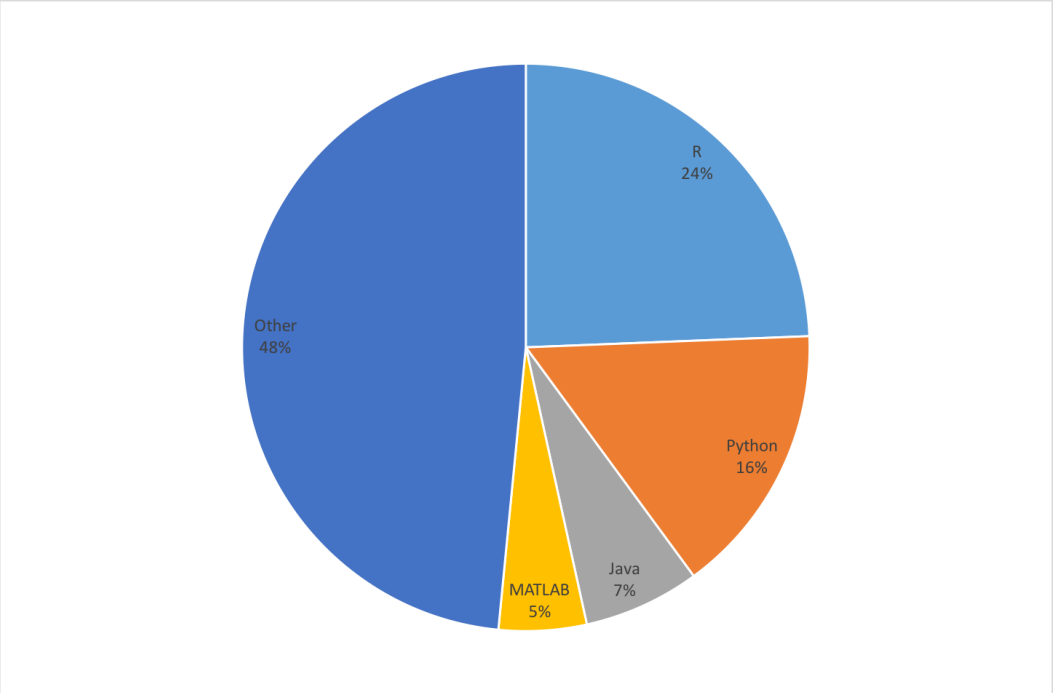


FIGURE 3.8: Programming languages used for data science in 2013 [50].

R

R is both a language and environment for graphics, statistical computing, and by extension machine learning. R provides easy and professional mathematical plot creation and is used extensively in the machine learning community, as shown in Figure 3.8. Despite this, many claim that R has a steep learning curve due to the R-paradigm choosing a highly interactive process over the traditional procedural approach. Moreover, there seems to be a lack of bindings available for the robot simulations discussed in Section 3.1.

MATLAB

MATLAB is used extensively for rapid work with matrices and has several machine learning algorithms available in the form of tool-kits. It also has graphical output which is optimized for interaction, allowing for interactive plotting and presentation. Possibly the biggest problem with MATLAB is its CPU and memory usage, which are valuable resources when working with machine learning algorithms.

Java

Java is a statically typed object-oriented programming language that is used heavily in industry, although less so for machine leaning purposes. From personal experience, Java tends to be slow when using machine learning libraries such as Weka [33]. Using Java for machine learning applications tends to be better when designing production ready applications instead of rapid research and development.

Python

Python is a dynamic typed and object-oriented language that allows for rapid development with a robust design. Many versions also ship with a number of useful packages, like Numpy and Scipy which handle complex matrix manipulation. Given our decision regarding simulation and machine learning library choice, it would seem that Python is a good candidate.

3.4 Summary Of Choices

In conclusion, we have decided that our implementation will be written in Python which will interface with Python bindings for the V-REP simulation and use the Python version on TensorFlow for our machine learning implementations.

Chapter 4

Experimentation Overview

Experimentation is split across two chapters separating our earlier work with V-REP in chapter 5, from later work with our custom simulation in chapter 6. This chapter serves as an overview of our experimental decisions that are used in both chapters 5 and 6. We introduce the arm that we will be simulating and training, followed by a description of the problem in terms of the reinforcement learning framework. We then introduce both the learning algorithm and network architecture that we used throughout our experiments, and conclude with the hardware that our simulation ran on.

4.1 Mico Arm

Weighing in at 4.6Kg and with a reach of 700mm, the Mico Arm [45] is a carbon fibre robotic arm manufactured with 4 or 6 degrees of freedom (DOF) and unlimited rotation on each axis. The arm is designed with mobile manipulation in mind which can be achieved with a 2 finger gripper. For the purpose of our work, we will be using the 6 DOF arm.

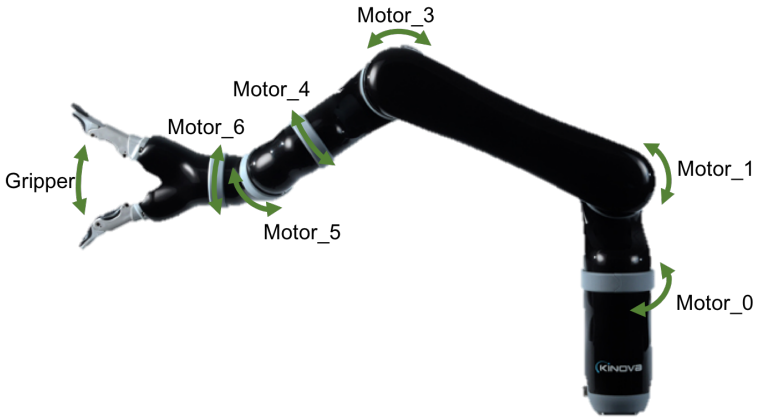


FIGURE 4.1: 6 DOF Mico arm.

4.2 The Reinforcement Learning Problem

We now discuss in detail the learning problem; highlighting that our problem is indeed a reinforcement learning problem. Ultimately, we wish for our agent to successfully

manipulate a given object through repeated trial and error with its environment. We model this problem around the reinforcement learning framework discussed in Section 2.1.

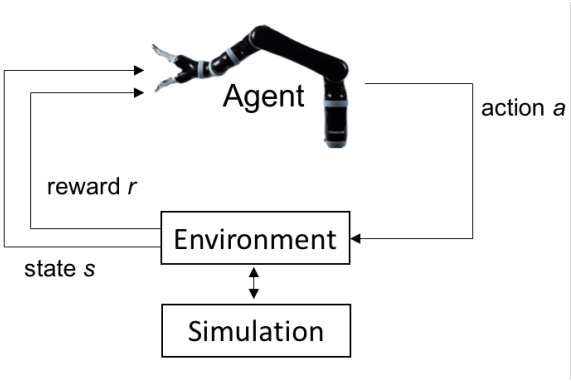


FIGURE 4.2: The reinforcement learning problem.

In this model, the robot arm plays the role of the agent, and everything else together with the simulation makes the environment. The environment is responsible for changing the state of the world via communication with the simulation and issuing rewards based on actions made by the arm — or specifically the neural network that dictates the actions of the arm.

At time t the agent should consider its state s_t and take an appropriate action a based on this. Upon making this decision, the environment and simulation will reflect this action as a change in state to s_{t+1}. This state, along with a reward r, is then issued to the agent, following which the agent repeats the process. This continues until the cube has been successfully picked up, or a certain number of iterations has been reached. From this, we can see that the problem is both *Markov* and *episodic*. We believe it to be *Markov* because knowledge of previous arm positions and actions does not help us when attempting to pick up an object, only the current arm position and action is of use. Moreover, we believe it to be *episodic* due to the existence of a terminal state (picking up the object) or otherwise a cap on a number of iterations to avoid the agent getting in states far from the terminal state.

4.2.1 State and Action Space

Our state space is inherently continuous, making both state representation and action selection difficult. Internally, the state of the arm can be represented by vector of 6 real values corresponding to the joint angles, and an additional integer corresponding to whether the gripper is open or closed. Instead, we represent the state of an arm via integers in order to take us from a continuous space to a discrete space. Actions therefore, allow each joint to rotate 1 degree in either direction, giving rise to 12 actions for joint movement, and 2 actions for opening and closing the gripper.

With 6 joints capable of rotating 360 degrees and a gripper that can be open or closed, the robot can be in one of approximately *4 quadrillion* internal states. The state the agent sees is captured via a greyscale image of the scene, with a resolution of 64×64. As each pixel will be a value from 0 to 256, that makes our state space size $256^{64\times64}$ — more than the estimated number of atoms in the universe. Naturally, this number far surpasses

the usability of classical tabular reinforcement learning methods, so instead we turn to function approximation.

4.3 Deep Q-learning

Our overall goal is to train our agent to pick up an object from a range of starting positions and joint angles, meaning that the agent cannot rely on memorising a sequence of steps that led to a previous successful task, but instead must develop a sophisticated representation of its environment. Deep Q-learning, introduced in section 2.4.1, has shown evidence of learning robust policies based on sensory inputs with minimal prior knowledge.

Online training through gradient descent combined with random starting positions leaves us susceptible to *catastrophic interference* [59]. This causes previously learned information to be forgotten upon learning new information, resulting in the inability to properly learn anything. Assume the agent performs a series of commands that lead to an object being grasped and lifted, resulting in a large reward. Our network would then try to learn that the sequence of state-action pairs results in a high reward and attempt to adjust the weights accordingly through backpropagation. We move the target object and restart the task. The agent now sees a similar state to its previous attempts, and might perform a similar set of actions only to receive a less, possibly negative reward for its actions. Upon performing backpropagation, the agent risks pushing the previously learned weights further away from their optimal values. Fortunately, deep Q-learning incorporates experience replay, which protects us from such a problem by randomly sampling from past experiences. It is therefore only natural that we use these advances in function approximation for our task.

Below we show the main steps of the learning process.

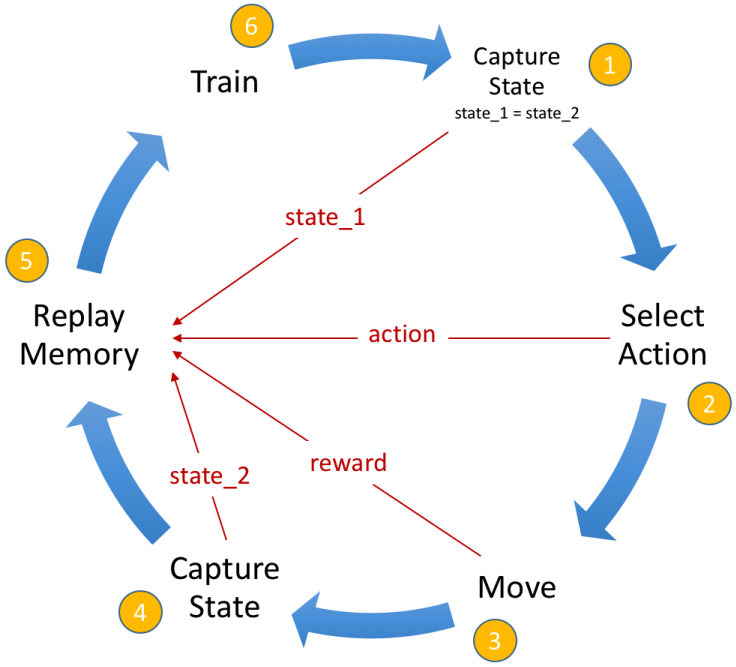


FIGURE 4.3: Learning process using deep Q-learning.

1. If we are at the very first iteration then capture the current state of the agent; otherwise set the previous state to the current state given that we would have not made an action from the previous state.
2. Select an action based on the policy we are following. If we are choosing a move given by our network, then we feed in the state and select the action with the highest *Q-Value*, otherwise we use another selection method.
3. The agent performs the action from the previous step and the environment outputs a reward.
4. Following the move, the new state of the agent is captured.
5. Using the data from steps 1-4, we store the original state, the action taken from that state, the reward from taking that action, and the new state following the action, into the replay memory.
6. We train the network with a randomly uniform batch of memories only if we have gone through a set number of iterations to fill the replay memory to a desired level.

One *iteration* is defined as completing one cycle of the process from figure 4.3. We define an *episode* as a sequence of either a maximum of 1000 iterations or until the agent grasps the cube and lifts it to a adequate height. At the end of an episode the scene is reset and a new episode begins.

4.4 Exploration Method

In order to ensure the right proportion of exploration and exploitation, we follow an ϵ-$greedy$ method whilst linear annealing the value of epsilon over a period of time-steps. This annealing is defined by:

$$\epsilon = \epsilon - \frac{\epsilon_{initial} - \epsilon_{min}}{explore\ period}, \tag{4.1}$$

where the $\epsilon_{initial}$ is the starting value of ϵ, ϵ_{min} is the value we do not wish ϵ to fall below, and $explore\ period$ is the number of iterations it takes to linear anneal ϵ to ϵ_{min}. This annealing is effective because we gradually reduce the amount of exploring the agent as time goes on, and instead focus more on exploiting the knowledge we have gained from exploring.

We usually set $\epsilon_{initial} = 1.0$, $\epsilon_{min} = 0.1$, and $explore\ period = 1000000$ iterations. After finishing the annealing process, ϵ is then fixed at 0.1 in order to ensure that the agent does not get stuck in states.

4.4.1 Network Architecture and Hyperparameters

To avoid many weeks that could be spent tweaking the network and hyperparameters, we decide to take inspiration from a number of successful networks from related work in order to decide on a network architecture and hyperparameter values. Sergey Levine et al. [56] use a 7 layer deep network with 3 convolutional layers, followed by a spatial softmax, and finally 3 fully connected layers. Both papers from Google DeepMind [65] [64] on the other hand, use 2 convolutional layers followed by 2 fully connected layers.

Bearing these architectures in mind, we construct our convolutional network to have 5 layers, and illustrate it below.

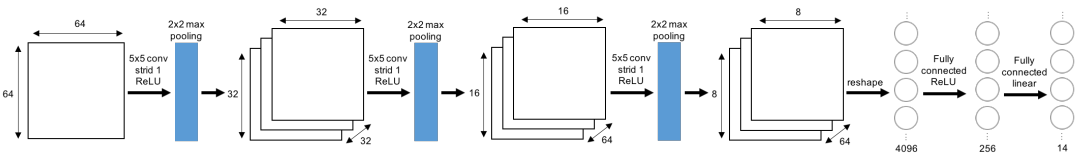


FIGURE 4.4: Our convolutional neural network architecture consisting of 3 convolutional layers and 2 fully connected layers. The network accepts 64×64 resolution greyscale images and outputs motor actions.

Our network contains 3 convolutional layers followed by 2 fully connected layers. Before entering the network, images are resized to 64×64 and converted from RGB to grayscale. The first layer then convolves the grayscale image with 32, 5×5 kernels at a stride of 1. The result of this is then put through a 2×2 max pooling layer. Both layers 2 and 3 convolve their inputs with 64, 5×5 kernels at a stride of 1, before entering a 2×2 max pooling layer as in layer 1.

At this point our images will be of dimension $8 \times 8 \times 64$, where 64 is the number of channels. These are then reshaped from the pooling layer into a batch of vectors and

process the data using a fully-connected layer with 256 neurons. The output layer is then obtained by matrix multiplication, creating 14 outputs corresponding to the Q-values for each action given the current state.

Network weights are initialised with a small amount of noise in order to break symmetry and prevent zero gradients. Additionally the *ReLU* neurons are initialised with a slightly positive initial bias to avoid so called "dead neurons".

During our experiments, we fix the deep Q-learning discount rate at $\gamma = 0.99$ and set our replay memory size to 500000. Gradient descent is performed using the Adam optimisation algorithm [44] with the learning rate set at $\alpha = 0.000006$.

4.5 Hardware

Teaching our agent can take millions of iterations and thousands of episodes, so optimising our experiments such that we completed each iteration fast as possible is an important task. All experiments were run on a *Intel Xeon E5-1630 v3* CPU running at 3.70GHz, 32GB of memory, and a *Nvidia GeForce GTX TITAN* graphics card.

A fundamental requirement for this project was a powerful graphics card to carry out intensive back-propagation each iteration. Before being assigned the machine described above, our initial experimentation was performed on a machine with an *Intel Iris Graphics 6100* chip; upon moving to the *GeForce GTX TITAN* we saw a $\times 750$ speed-up in training time, enforcing the need for the right hardware.

Chapter 5

Early Experimentation

This chapter walks through a number of experiments run using the third-party V-REP simulator. It is in this chapter that we make our first contribution — learning robot control policies through simulation. Following this, we are forced to abandon our original choice of simulator due to long training times. The remaining contributions are then made in chapter 6.

5.1 The Scene

Most of our environment will be captured via a scene we build within our simulation. Within our scene, we include a target cube to manipulate, the Mico arm that would perform the task, a camera to capture images, and finally a table where both the arm and target will rest.

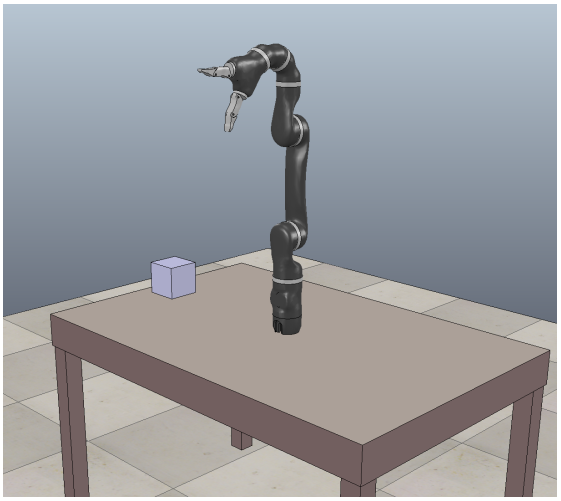


FIGURE 5.1: Scene in V-REP.

The scene was intentionally made minimal in order to reduce the complexity of the task. If the agent could not pick up an object in a minimal scene, then it would have no hope in a cluttered and complex one.

5.2 V-REP Communication

V-REP offers a remote API that allows us to control aspects of the simulation. Communication between our Python application (client) and V-REP (server) are done via socket communication. The server side is implemented via a plugin which we can communicate with via the Python bindings provided.

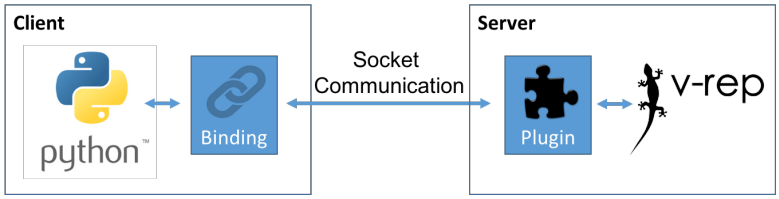


FIGURE 5.2: V-REP remote API communication.

5.3 Underestimating The State Space

In this early stage of development, we did not want to focus directly on achieving our ultimate goal, but instead our first priority was to prove that our convolutional network had the ability to learn policies directly from images. The agent's joint angles are set so that the agent began in an upright position, while the cube is located a few inches away from its side.

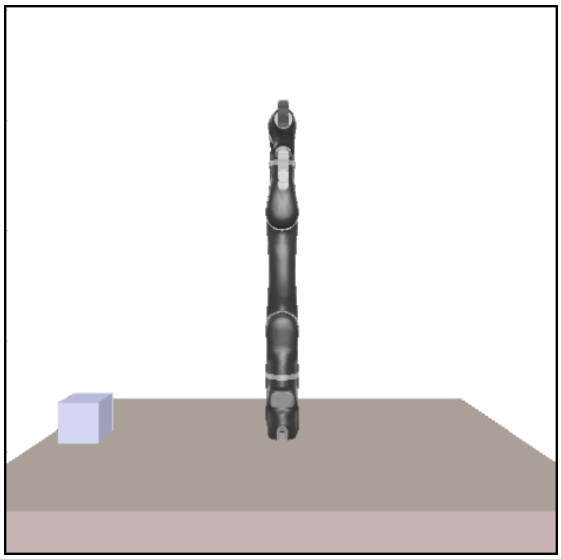


FIGURE 5.3: The starting position of our first experiment. Joint angles are set so that the arm is set in this position at the start of each episode.

At the start of each episode, both the agent and cube are placed in the same starting position, removing the network's ability to generalise at this point. Upon completing the task successfully, the agent is given a large positive reward of 100, whist all other actions are given a small negative reward of -1. The purpose of this was to encourage the agent to choose policies that completed the task in as little time as possible.

The motivation for this was that successful policies that completed the task with useless or redundant moves would receive a lower average reward than a policy that completed the task with fewer redundant moves, therefore giving a higher expected return for that policy.

After 200 hours of training and almost 2 million iterations, the agent was nowhere near picking up the cube. This was due to the agent never actually picking up the cube while following the $\epsilon\text{-}greedy$ policy, and subsequently never getting a chance to experience greater rewards than -1.

In hindsight, this was not surprising given the starting position and the extremely low probability that a set of random actions would be selected that resulted in the desired behaviour. Granted, giving enough time the agent could potentially perform a series of moves that would pick up the cube, but the amount of time required far exceeds any reasonable number to be considered practical.

One interesting observation whilst watching the was that the agent seemed to spend many iterations choosing the close gripper action when the gripper was already closed, and the open gripper action when the gripper was already open.

Ultimately, we believe the failure of this task is down to a flawed reward scheme. Although the negative rewards encourage faster task completion, they do not give any new information to our agent in regards to how to complete it.

5.4 Learning Robot Control Policies Through Simulation

At this point we had not managed to prove that our convolutional network was capable of learning policies that could complete the task. Realising that starting the agent in an upright position was a mistake, we instead set the arm in a starting position in such a way that the probability for a sequence of actions that would lead to the cube being picked up was substantially higher. We therefore set the joint angles at the start of each episode so that the gripper was situated a few centimetres above the cube.

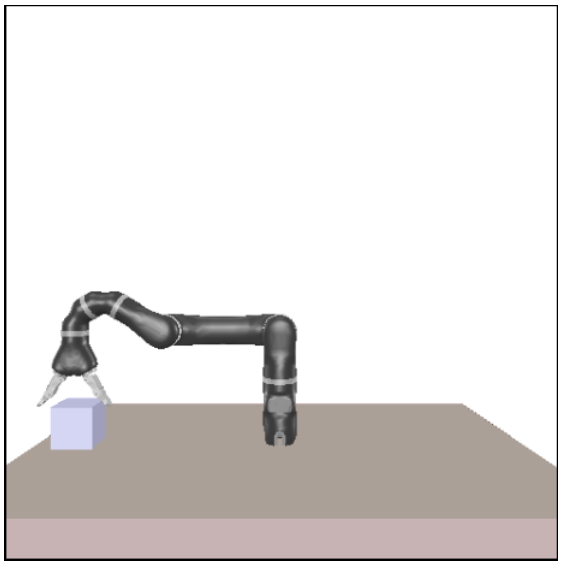


FIGURE 5.4: The starting position of our first successful experiment that picks up the cube. Joint angles are set like this so that the probability to pick up the cube is higher than in the previous experiment.

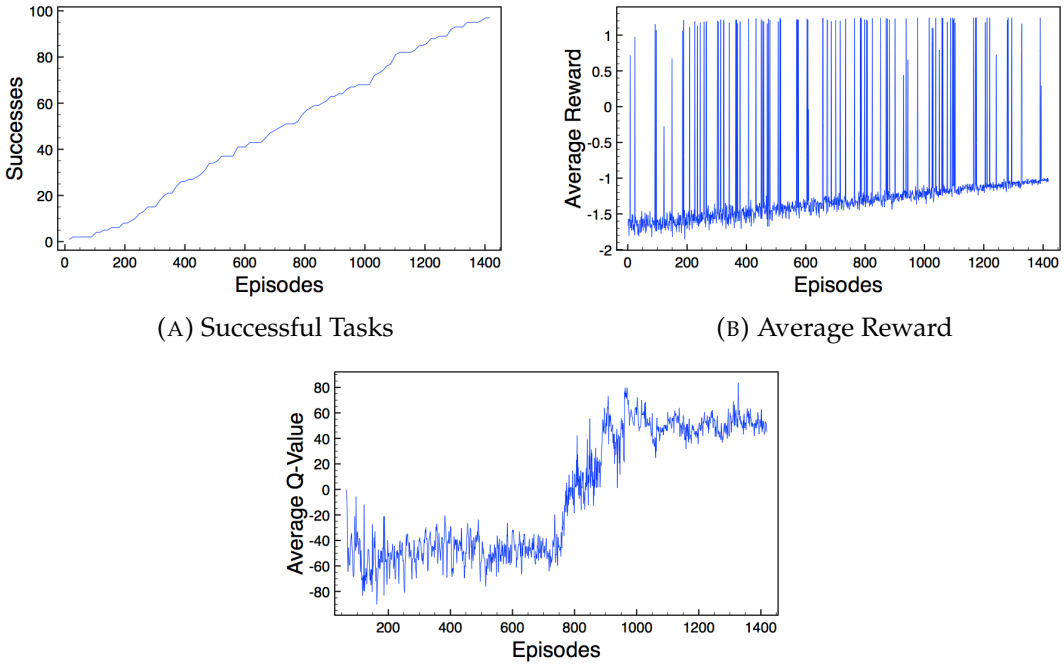
As we now had a greater chance of success through random exploring, we decided to use the same reward scheme from the previous experiment to confirm our hypothesis that these rewards lead to fast task completion given the chance to gain the large reward. In an attempt to solve the problem of trying to close the gripper when already closed, we added a negative reward of -10 in order to deter the agent from taking redundant actions.

Excluding the starting position of the joint angles, all other aspects of this experiment remained the same as the previous experiment.

After approximately 800 episodes, it was clear that the agent had successfully learned a policy to pick up the cube. Not only that, but we witness an episode that reveals something exciting. During this episode, the agent had successfully grasped the cube and began to lift it before it slipped from the gripper just short of the target height. Following this, the agent re-opened its gripper and proceeded to pick up the cube again and successfully completed the task.

The findings of this important milestone were twofold: we had proven that it was possible to learn policy directly from simulations through our convolutional network, and secondly the agent's re-attempt of picking up the cube shows evidence that it is aware of the cube as a source of reward, rather than by following a fixed set of actions each episode.

Observing the graphs below gives us a further insight into the learning process of our agent.



(A) Successful Tasks

(B) Average Reward

(C) Average Q-Value

FIGURE 5.5: The results of our first successful experiment that picks up the cube.

We can see from figure 5.5a that the agent gradually improves at the task of lifting the cube. After 200 episodes it is successful about 4% of the time, while after 900 episodes it is successful about 7% of the time. Figure 5.5b clearly shows the agent learning by improving its average reward as the number of episodes increases. We see that the line tends to stay in negative rewarding due to the -1 reward for each move and the -10 reward for a redundant move — as redundant moves are less probable, it makes sense that the line should be situated between -1 and -2 for the majority of training. Spikes in this graph indicate a successful episode and indeed correlate with figure 5.5a. Although this metric shows some evidence of learning, we do not get an indication of when the agent actually starts to learn. Figure 5.5c illustrates this process nicely with a sudden increase in the average maximal Q-value at around 760 episodes — showing that the agent had found a set of policies that lead to an increase in the discounted reward.

Observing the agent lift the cube along with the graphs presented above gives conclusive evidence that it is possible to learn policy directly from RGB image in a simulation.

5.5 Intermediate Rewards

Now that we had shown that our convolutional network could learn policies to complete the task given enough successful episodes, we could now turn to the problem of state exploration.

With our large state space it is only possible to visit a small fraction of these states in a realistic amount of time. From an upright starting position, using random actions the probability of the agent not only grasping the cube, but then proceeding to pick it up is incredibly low. In section 5.3, we learnt that we need a more progressive reward scheme so that the agent explores interesting state spaces — that is, ones that lead to high rewards.

Our choice to train using simulations allows us to dynamically move the target and access its new position, using this information in such a way so that the agent explores states around the target area. One way of using this positional information is to give the agent intermediate rewards based on its distance from the cube. Given that we can calculate the distance as:

$$distance = \sqrt{(x_{target} - x_{agent})^2 + (y_{target} - y_{agent})^2 + (z_{target} - z_{agent})^2}. \tag{5.1}$$

A trivial linear reward can then be given to the agent:

$$reward = 1 - distance. \tag{5.2}$$

The problem with this is the difference in rewards given between 2 different positions close to the target and 2 equally different positions far away from the target are the same. Ideally the difference between reward from 2 different positions that are both far from the target should be very similar, while rewards between 2 different positions that are close to the target should be significantly greater than the former. The inverse exponential function is a good candidate for this, and so we define the reward as:

$$reward = e^{-\gamma \times distance}, \tag{5.3}$$

where γ is the decay rate.

We apply these changes to a new experiment and change the starting joint angles such that the agent starts further away from the cube, as shown in figure 5.6.

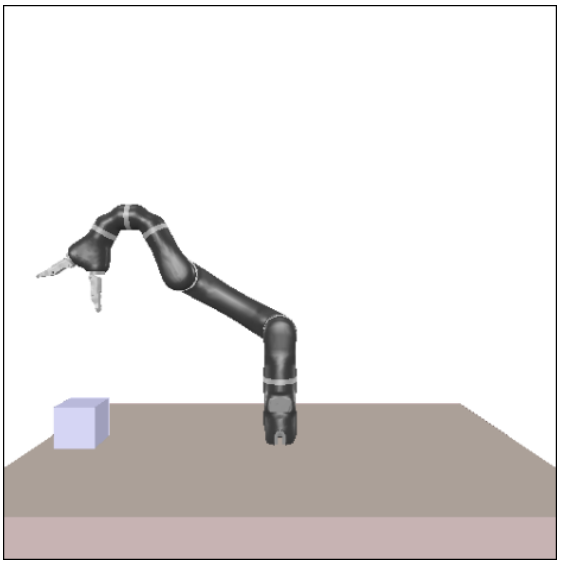
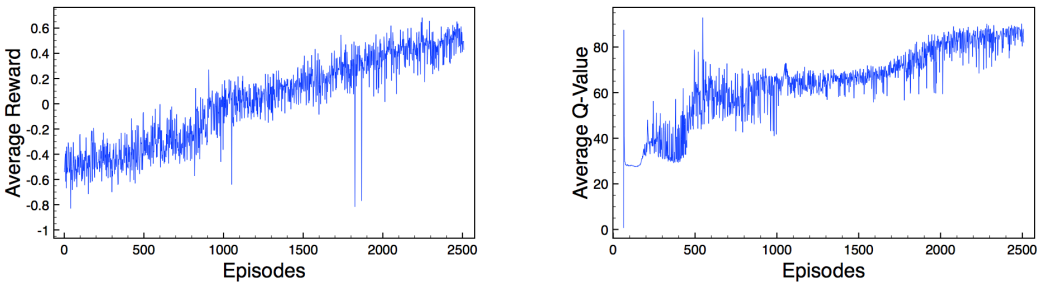


FIGURE 5.6: The starting position of this experiment which uses intermediate rewards to try and guide the agent to pick up the cube.

The reward system would use the exponentially decaying reward explained above, along with a large positive reward for completing the episode successfully.

After running the above configuration for 2 million iterations, the agent had not succeeded in its task. Despite this, our agent did learn to explore areas near the cube, but this was not enough to make it then grasp the cube and proceed to lift it. Whilst observing the agent at this point, you would typically see it direct moving its gripper to the cube followed by juddering that would push and drag the cube.

Despite not completing the task, evidence of learning can be seen from the graphs below.



(A) Average Rewards (B) Average Q-Values

FIGURE 5.7: Results of using intermediate reward to guide the agent to pick up the cube. The agent did not succeed in picking up the cube once during training. Despite this, evidence of learning can be seen through these graphs and by watching the agent move to explore areas near the cube.

Figure 5.7a shows a steady increase in rewards while figure 5.7b shows some learning activity at around 500 episodes, with a more gradual increase again at 1700 episodes. The increase in both rewards and Q-values is a result of the agent learning to explore

areas closer to the cube which is a source of rewards. Unfortunately the cube never makes the sequence of actions that then results in the cube being lifted. Given longer to run, the agent may have continued to explore these interesting areas and then by chance receive the highest reward for lifting the cube — but the agent had already exceeded what we felt was a practical amount of training time.

Ultimately, we hypothesise that the failure of this task was similar to that highlighted section 5.3 — that the probability of a sequence of actions that would result in the agent picking up the cube was extremely low. The incremental rewards did indeed improve the chances that the agent would accomplish this, but of course, as epsilon decreases we are more likely to progress to the cube but also then less likely to make a random move that will result in the cube being lifted.

It was clear that incorporating intermediate rewards enticed our agent to explore areas near the cube, but now we needed to further entice the agent to make subsequent actions to grasp and lift the cube through introducing additional rewards.

Chapter 6

Later Experimentation

This chapter contains the second half of our experiments using a new custom-built simulation. We first highlight the problems that led us abandoning V-REP and then introduce our custom-built replacement. We proceed to apply the lessons learnt from the previous chapter and highlight our remaining contributions.

6.1 Problems

The work documented in the previous chapter had taken a considerable amount of time to accumulate due to our reliance on V-REP for simulating the robot arm. As the project progressed, it became apparent that substantially longer training durations would be needed for each experiment. Unfortunately, many of the operations within V-REP take longer than we would have hoped. For example, capturing images within V-REP took approximately 200 milliseconds, leading to increases in each iteration time. In addition to this, joints are moved using PID controllers and cannot be moved instantaneously, causing an unnecessary delay between sending the command and completing the move. The average delay for a one degree rotation was 100 milliseconds, while opening or closing the gripper averaged 500 milliseconds. This required explicit sleeps in our code to ensure the joint had reached its desired rotation before attempting to capture an image of the changed scene — an incredibly large bottleneck when running millions of these commands. On average, an experiment that ran for 2 million iterations would take approximately 10 days to run.

A separate issue was the inability to run V-REP, as well as any other OpenGL application, without a display on the lab machines. Upon working with the computing support group (CSG), it seemed that the issue was down to a conflict in graphic drivers. Unable to solve this problem, CSG allowed us to reserve a machine and remain physically logged in. The problem with this was frequent interruptions by students logging the machine out — causing delays in our experiments. This problem was then later solved with the help of CSG and is documented in appendix A.

6.2 Custom Built Simulation: Robox

Despite spending a large amount of time choosing a third-party simulation at the start of the project, we decided to instead develop our own simple simulation in the hope of

reducing training time. The result of this was *Robox* — a 3D simulation developed in *Unity* [98] and *C#*.

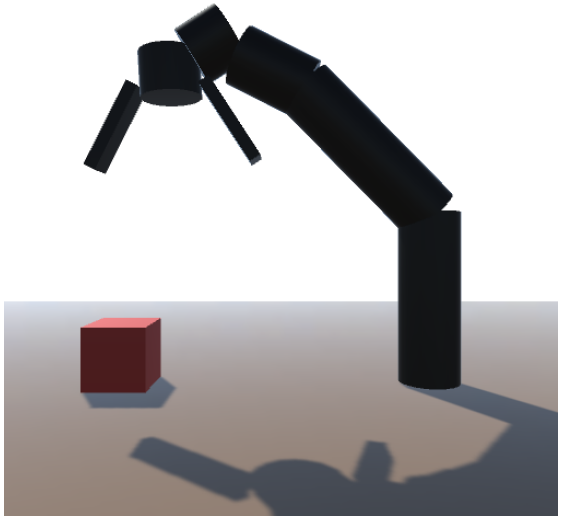


FIGURE 6.1: Custom built simulation: Robox.

The simulation models a simplified version of the Mico arm and gives us a number of benefits over our original choice of V-REP. Robox addresses our concerns with speed due to moving the arm to the desired joint angles without simulating the motor actions. This allows for faster iterations and gives us a $\times 10$ speed-up over V-REP. As we have full access to the source we are able to fully customise the tool to our needs with the ability to add new features as the need arises.

With our new simulator in place, we were able to take the lessons learnt from our earlier experiments, along with the ability to run many more iterations per hour, and revisit some of our initial attempts with some slight variations. We start this effort with an extension of our work on intermediate rewards from section 5.5.

6.3 Static Cube

In our previous chapter, we have shown that it is possible to learn directly from images when given the opportunity to succeed at a task. We now wish to enhance this by showing that it is possible to increase the chance of success when given sufficient training time and by engineering well designed intermediate rewards. We show this by keeping the cube starting position constant as well as the starting joint angles — though these should generalise to other starting angles surrounding the cube.

In addition to a large positive reward for task completion, the agent receives a reward based on the exponential decaying distance from the gripper to the cube, a reward of 1 while the cube is grasped, with an additional reward on top of this depending on the distance the cube is from the ground. These rewards were chosen to guide the agent firstly towards the cube, secondly to grasp the cube, and finally to lift the cube. The combination of these rewards along with the large reward for task completion will result in optimal rewards for this task.

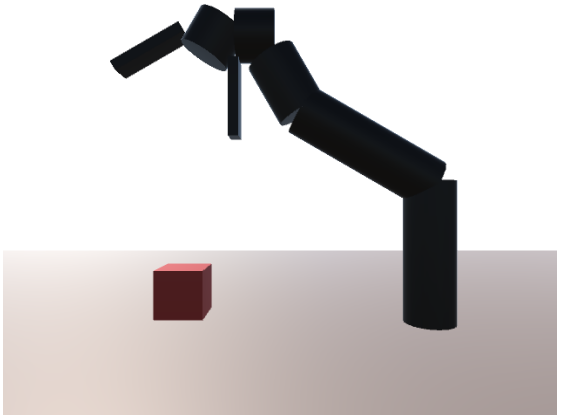
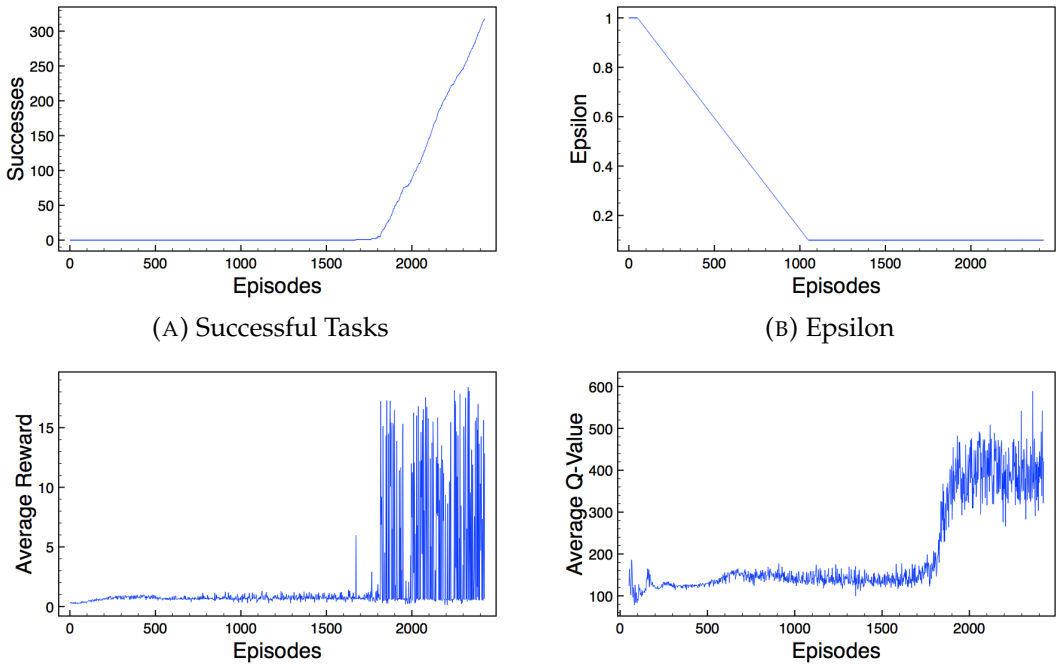


FIGURE 6.2: The starting position of an experiment where the cube position and joint angles are kept constant at the start of each episode. Intermediate rewards are used to guide the agent to interesting state spaces.

Figure 6.2 shows an example of what the network receives as input. After 1.8 million iterations, the agent had learnt a set of policies to pick up the cube and could generalise to other starting states that exist along the path to the cube. Moreover, the agent exhibited behaviour suggesting that it was aware of the cube as a source of reward by occasions where it dropped the cube and returned to retrieve it in an attempt to regrasp it. When the agent is tested in the same environment it was trained in, it had a success rate of 56%. This is discussed further detail in section 6.8 along with comparing with other agents. Below we show our training results.



(A) Successful Tasks

(B) Epsilon

(C) Average Reward

(D) Average Q-Value

FIGURE 6.3: The results of using intermediate rewards to guide the agent to interesting state spaces. The agent succeeds to learn a set of policies to pick up the cube.

From the graphs in figure 6.8, we see that the first 1000 episodes are spent exploring the state space as we anneal the epsilon from 1.0 to 0.1, which then remains constant for the remainder of the experiment. We see conclusive evidence of learning after approximately 1800 episodes from graphs 6.3a, 6.3c, and 6.3d. At this point, the number of successful episodes tends to increase linearly (figure 6.3a) along with an overall higher frequency of increased average rewards (figure 6.3c). Finally the average Q-values illustrate this learning very clearly with a sharp increase at around 1800 episodes before then stabilising with a Q-value of roughly 400 (figure 6.3d). We believe this is evidence of the agent finding a stable policy that completes the task.

Below we show the weights of the 32, 5×5 kernels from the first layer of the network after 1.8 million iterations. We point out interesting sections of the images below by using the following convention: (x, y) — where x is the column and y is the row.

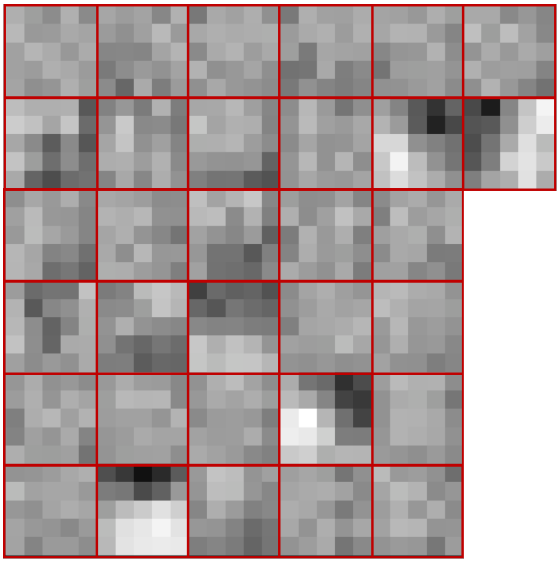


FIGURE 6.4: The 32 kernels from the first layer of the trained network — showing us what type of features our network looks for in our images.

Upon observing the weights, we can see what the network has learned to respond to. We are able to make out a horizontal edge detector in $(2, 6)$, a vertical edge detector in $(6, 2)$, and finally two upper right hand corner detectors in $(4, 5)$ and $(5, 2)$.

With the success of this experiment, we now look to see what the agent understands about its environment. Specifically, we are interested in knowing if the agent is aware of the cube as a source of rewards. We have already discussed evidence of this by its behaviour while re-attempting grasps, but now we look at the activations in the feature maps to give additional evidence.

(A) 32 maps from layer 1

(B) 64 maps from layer 2

(C) 64 maps from layer 3

FIGURE 6.5: The activations in the feature maps of the trained network during the agent's attempt to pick up the cube from a constant starting position and joint angles.

Figure 6.5a shows us the resulting 32 images after passing through the first convolutional layer, while figures 6.5b and 6.5c are the resulting 64 images after passing through the second and third convolutional layers respectively. Most notably are the activations in both $(8, 2)$ and $(2, 8)$ of figure 6.5b, which seem to show high activation on both the cube and segments of the arm. This leads us to believe that the network is not only capturing the state of the joint angles, but also the position of the cube.

Below we show 68 frames of a successful episode. There are 5 images labelled from A to E that correspond to the frame numbers in the graph.

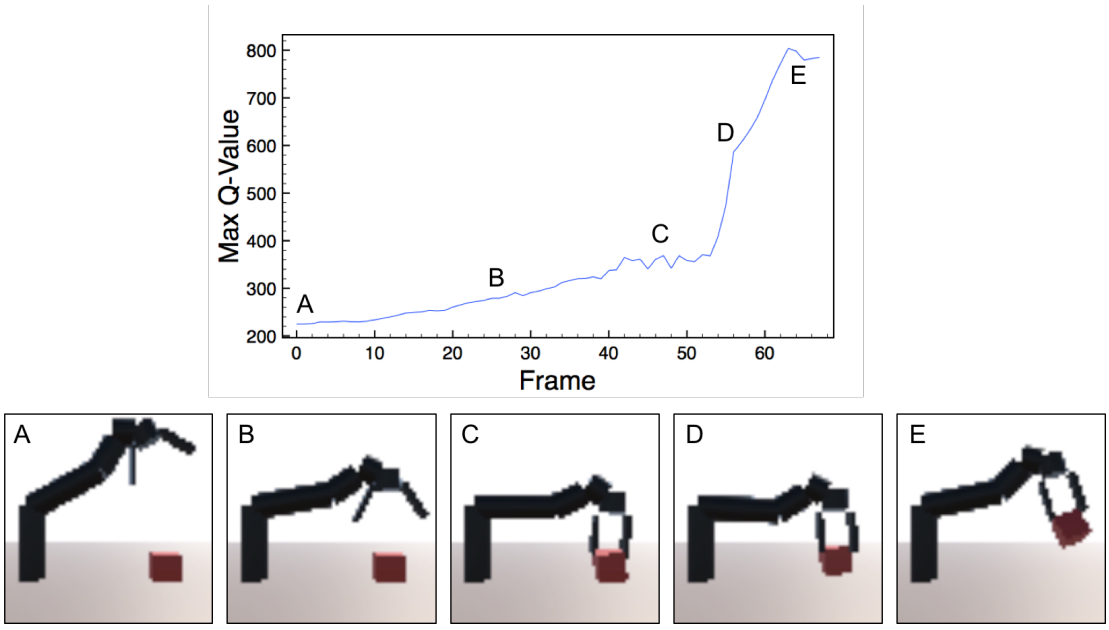


FIGURE 6.6: A frame-by-frame Q-value visualisation of a successful completion of a task. At each point in the episode, the highest Q-value is chosen and plotted. The labels from A to E correspond to the frame numbers in the graph.

The frames from A to C show a steady increase in the Q-values which is a result of the agent getting closer to the cube. At frame C, the Q-values fluctuate a little due to the

network trying to determine if it has grasped the cube or not. By the time we get to frame D, there has been large jump in the Q-values where the agent has successfully determined that it has the cube. The Q-values then continue to rise as the distance of the cube from the ground increases, and finally peaks as the agent expects to receive a large reward. The figure above gives evidence that the value function is able to evolve over time for a manipulation task.

6.4 Adding Joint Angles

As we have access to joint angles when training the agent, we wanted to know if inputting both images and joint angles would reduce the amount of time it took to learn policies to complete the task. Up to now, our agent has to interpret 2 pieces of information from an image: the state of the arm (joint angles), and the position of the cube. Explicitly adding in joint angles to the network reduces the amount of information the agent has to interpret from the image.

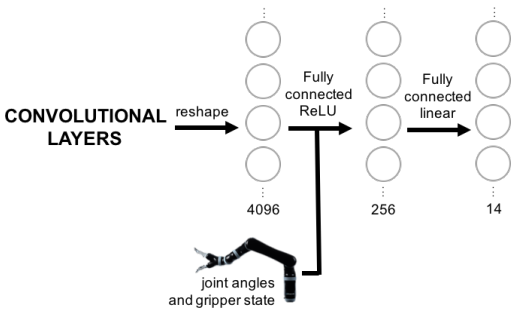
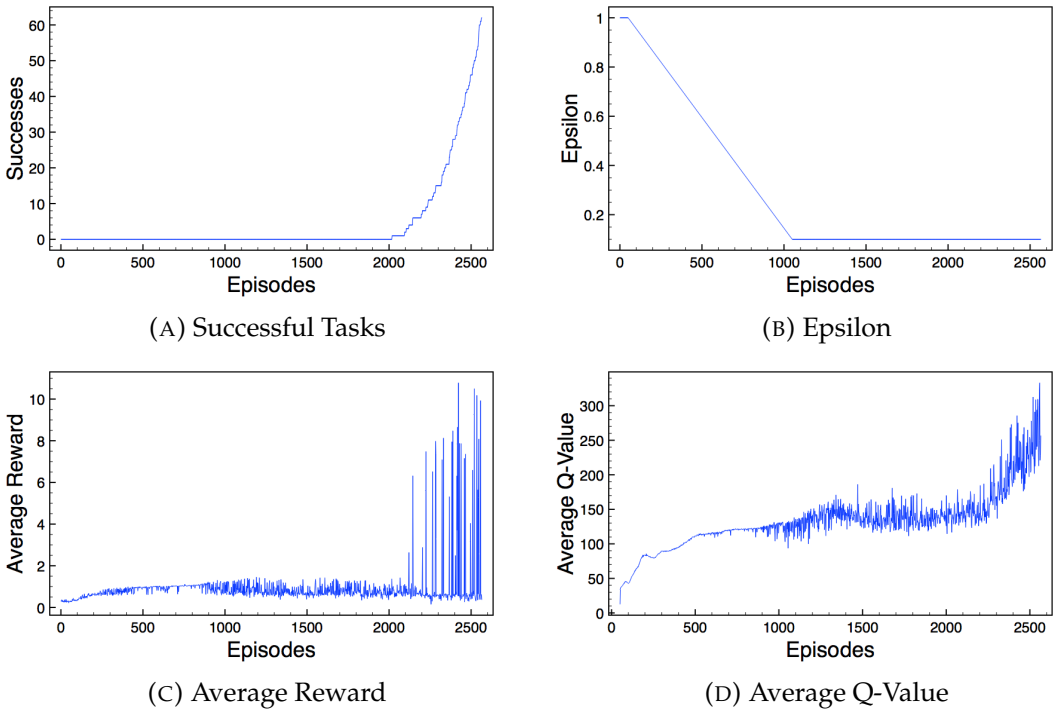


FIGURE 6.7: Concatenating the 6 joint angles and gripper state to the first fully connected layer of the network.

We modify our existing network by concatenating the 6 joint angles and gripper state to the first fully connected layer, as shown in figure 6.7, and then train the network as in the previous experiment. When the agent is tested in the same environment it was trained in, it had a success rate of 5%. This is discussed further detail in section 6.8 along with comparing with other agents. Below we show our training results.



(A) Successful Tasks

(B) Epsilon

(C) Average Reward

(D) Average Q-Value

FIGURE 6.8: The results of adding joint angles into the network. They show that training is no faster than omitting the joint angles.

We see similar results to the previous experiment, except the time taken to learn successful policies has increased. Looking back at figure 6.3a and comparing with figure 6.8a, we see that the network that used the image as input started succeeding regularly about 200 episodes before the network that takes both images and joint angles.

These results seem somewhat counter intuitive — we are reducing the amount of information that needs to be interpreted from the image to just locating the cube, yet despite this, training time is not reduced. This seems to be an open problem that we can only speculate about at this point until further research is performed. One possible explanation could be that the inputs are dominated by the image and that the joint angles were not prominent in comparison. Our network receives 4096 inputs from the 64×64 resolution input image and just 7 input from the robot configuration, meaning that the image has over 585 times more representation in the input values in comparison to the robot configuration.

Also, as the agent carries out random moves with probability ϵ, the previous experiment could have by chance received better random moves that made the agent train faster. Either way, there seems to be no substantial evidence that adding in the joint angles reduces training time, and so the remainder of our work was carried out using the original network configuration in figure 4.4 — though we would hope that further work investigates this open problem further.

6.5 Moving The Cube

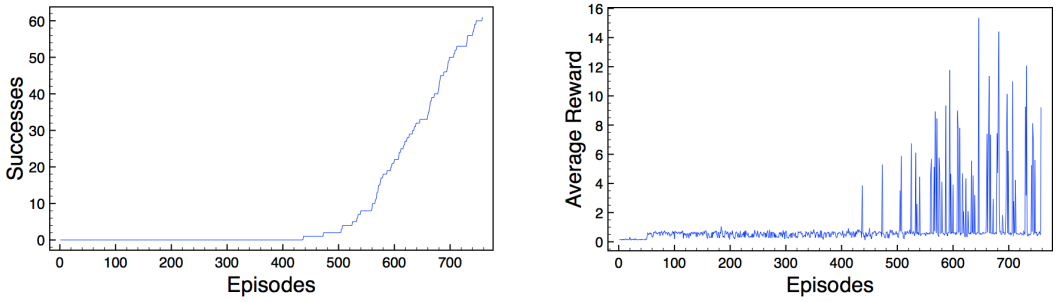
We now wish to extend upon our success from section 6.3 and attempt to make our network generalise to different cube positions, as well as other starting joint angles. The area in which the cube starts each episode is shown via a top down view in Figure 6.9.

FIGURE 6.9: The area (approximately $200cm^2$) in which the cube can be placed at the start of an episode.

We give the agent the network that was trained in our earlier experiment from the previous section as the weights will already be close to the desired weights for this experiment — reducing the number of episodes needed to learn. Each time the agent is successful in completing the task, the cube is moved to a random location within the highlighted area. Moreover, at the start of every episode, the joint angles are set as in section 6.3, but with a random variation of 20 degrees for each joint.

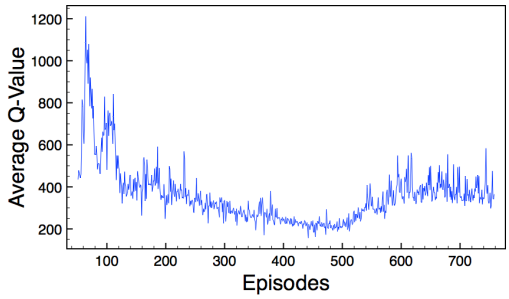
For this experiment, we set the initial value of epsilon to 0.5, and then anneal this down to 0.1 over the course of 1 million iterations. We choose this starting value of epsilon because we already have a network that explores interesting parts of the state space, and so our epsilon does not need to be set as high. On the other hand we do not wish to restrict the agent from exploring the new locations the cube could go, so we feel 0.5 is about right.

The experiment succeeds with the agent being able to generalise to other starting cube positions and joint angles. When the agent is tested in the same environment it was trained in, it had a success rate of 52%. This is discussed further detail in section 6.8 along with comparing with other agents. Below we show our training results.



(A) Successful Tasks

(B) Average Reward



(C) Average Q-Value

FIGURE 6.10: The results of training out network to generalise to different starting positions and joint angles.

The first thing to notice about these results is that it only takes approximately 500 episodes for the agent to start to learn policies that complete the task. This shorter period was to be expected given that we start with the network weights from our previous experiment and use a smaller starting value of epsilon.

The second thing to notice is the sharp increase in the average Q-value in the initial episodes, which could be a result of the network adjusting to its new environment. We see a similar increase in the Q-values of approximately 200 as we did in section 6.3 when the agent learned a policy that completed the task. Following this increase, the agents Q-values re-stabilise, suggesting we have found a stable policy for task completion.

Although we do not expect either the kernels or feature maps to differ greatly from the previous section, we show them below for completeness.

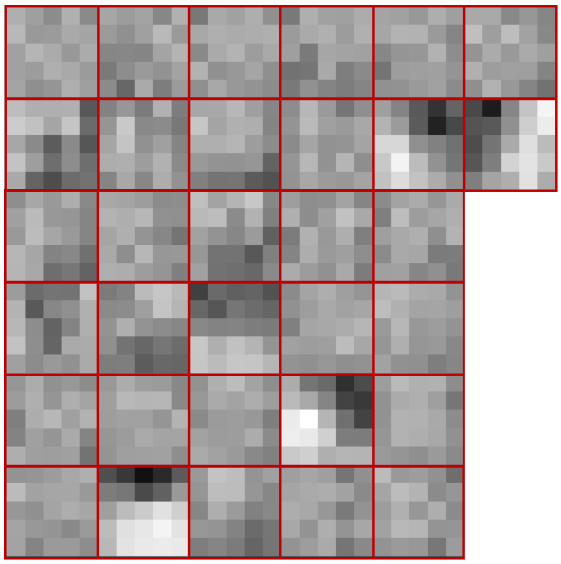
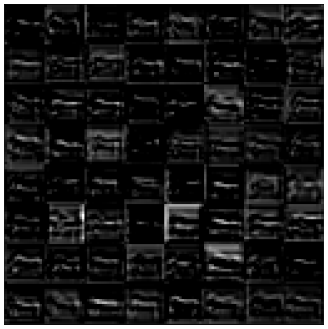


FIGURE 6.11: The 32 kernels from the first layer of the trained network when trained to generalise — showing us what type of features our network looks for in our images.

What we do notice in both figures, is that the weights have become stronger in figure 6.11, leading to increased activity in figure 6.12. We see some development of new detectors, such as a bottom right hand corner detector in $(1, 2)$, and evidence of detectors changing into different ones, such as the once top left hand corner detector in $(6, 2)$ turning into a vertical edge detector.

(A) 32 maps of layer 1

(B) 64 maps of layer 2



(C) 64 maps of layer 3

FIGURE 6.12: The activations in the feature maps of the trained network during the agent's attempt to pick up the cube from different starting positions and joint angles. Activations have gotten stronger since being trained to generalise.

6.6 Generalisation

We now have a trained network that generalises well to different starting states — including joint value variations and target positions. Without further training, we wish to see how well this network generalises when put into situations and environments that it has not previously seen. We test the agents capability to generalise in the following ways:

- Making slight variations to the target cube.
- Adding clutter into the scene
- Expanding the starting position of the target

Each test was run for 50 episodes with success being counted as the agent successfully lifting the cube a few inches of the ground.

6.6.1 Shape Variations

Although our network from the previous section had only been trained on one cube size, we wished to see if the agent could succeed when slightly changing the target object. Naturally, we could not drastically alter the target without further training, so our focus was to stretch and shrink the cube we trained on as well as trying testing on a sphere that is approximately the same size as cube. Below are a sample of the variations tried.

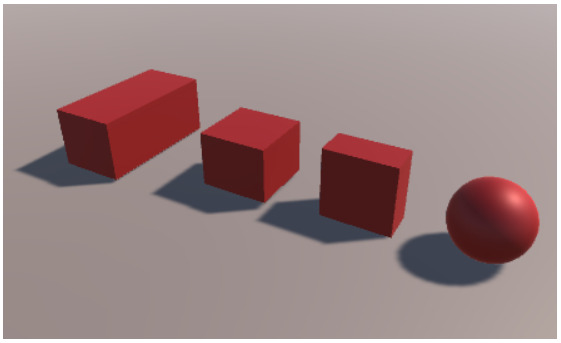
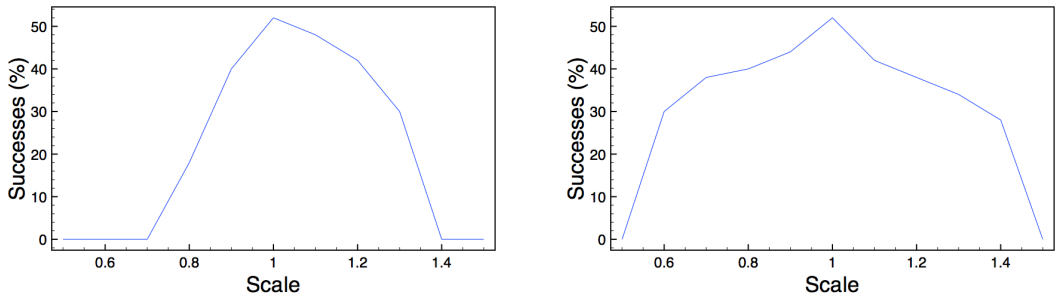


FIGURE 6.13: A sample of variations that we apply to the target object to see how well it generalises without any additional training.

The agent was not able to generalise to the sphere in the far right of figure 6.13. Having said that, the agent spent a large amount of time with its gripper above the sphere as if it was attempting to pick it up. We expected that the sphere might cause problems as some of the kernels illustrated in figure 6.11 look for edges and corners that are associated with the cube shape it was trained on. If during training we would have randomised the shape, we hypothesise that the agent would have generalised well to also succeed with the sphere.

Below we show how well the agent generalise when varying the height and depth of the cube. The x axis shows how much the cube is scaled in comparison to the trained size.



(A) Height variations (B) Depth variations

FIGURE 6.14: The change to the percentage of successful episodes when being tested on a target object with varying height and depth.

Both the graphs in figure 6.14 show that the agent performs best when tested with a scale of 1. This was to be expected as this was the size of the cube the agent was trained on. The graph in figure 6.14a shows us that when varying the height of the cube, the agent generalises best to the cube height being increased rather than being decreased. When the cube height is *increased* by 30%, the agent succeeds 30% of the time. Conversely, when the cube height is *decreased* by 30%, the agent does not succeed once. One explanation for the dislike to a reduction to height could be that the edges are not as pronounced, especially when our images are a resolution of 64×64. If this was the case, then this would mean that there would be no activations on the cube — leading to the inability to locate the target.

The graph in figure 6.14b shows us that when varying the depth of the cube, there seems to be no significant difference between the percentages of success between increasing and decreasing the depth of the cube. Moreover, the drop in successes is more gradual in comparison to figure 6.14a. This is most likely due to less emphases on the depth of the cube to calculate its position in comparison to the front face of the cube, which is directly effected by altering the height rather than the depth.

6.6.2 Clutter

While keeping the target object the same as what was used to train the agent, we add clutter to the scene in the form of a simple bear shown in the figure below.

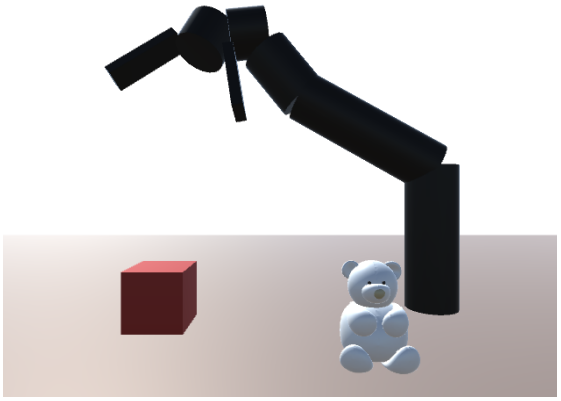


FIGURE 6.15: An example of introducing clutter into the scene in the form of a bear. We test the agent to see how well it generalises when adding in clutter without any further training.

Surprisingly the agent adapted well when a bear was positioned far enough away from the cube, such as in the figure 6.15. When the bear got close to the cube however, the agent seemed to struggle with locating its target, and executed what looked like a random selection of moves. In order to see how well the agent generalises, we increasingly add clutter into the scene until the agent no longer succeeds. Due to the agent unable to generalise at all when the bear is placed near the cube, we instead place the objects near to the base of the arm.

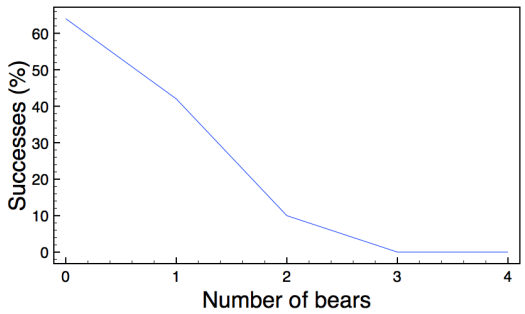


FIGURE 6.16: The results of how adding in additional clutter deteriorates the success rate of the agent.

The graph in figure 6.16 shows us our base success of 64% when no clutter is added to the scene and a drastic decrease when there are 2 or more additional objects added. This could be a result of adding too much information into the scene that the agent has never seen before. We hypothesis that if further training was carried out with the addition of clutter, the agent would generalise to a higher number of bears in the scene.

6.6.3 Expanding Starting Position

The final generalisation test we perform is a gradual increase the area in which the cube can be placed. For this test we sample positions from a uniform distribution within a defined area. For this test it is important to remember that the agent in test

was trained within a $200cm^2$ area. The target object the same as what was used to train the agent.

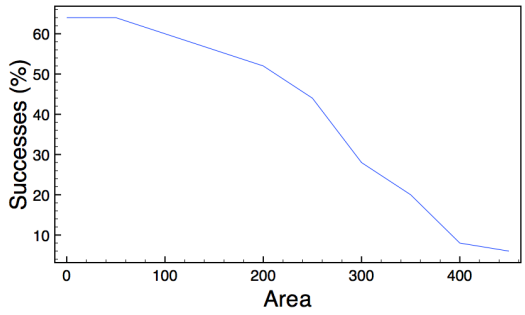


FIGURE 6.17: The results of how the agent generalises to increases in the area in which the cube is positioned at the start of each episode. The area is defined in terms of cm^2.

The results in figure 6.17 show that the agent performs best when restricted to the area in which it was trained, and continues to perform well for approximately $100cm^2$ above its training area. This can be somewhat misleading, as the agent is not necessarily generalising to positions outside of the $200cm^2$ area, but rather the cube is placed within this area due to the uniform distribution of positions. The percentage of successes naturally decreases as the area increases, but does not hit 0% even at an area of $450cm^2$ due to the uniform distribution.

6.7 Attempt at Reducing Training Time

We have experienced much success from our agents trained using our simulator. Although training times seem to be within the standard time in this field, we wanted to see if it was possible to reduce this time.

So far we have been training our agent via unsupervised learning through randomly exploring the environment. In this section we experiment with supervised learning to find out if it is possible to use this to reduce training time.

We consider using the information available from the simulation in order to allow the agent to learn by observing a teacher or oracle $\mathcal{O}$ perform the task. This requires us to implement a way of calculating a set of actions that would be needed to achieve this task.

6.7.1 Inverse Kinematics

Two common problems in robotics are *forward kinematics* and *inverse kinematics*. Forward kinematics solves the problem of calculating the position of the gripper given the joint angles. A far more useful but harder problem is inverse kinematics, which calculates the joint angles given the desired gripper position.

Using a simple inverse kinematics method, we design an algorithm that takes a position above the cube in order to to calculate a target pose that gets the agent in the

position with its gripper above the cube. From this, we are able to execute a fixed set of actions that would result in the cube being grasped and subsequently lifted.

6.7.2 $\mathcal{O}$-$Greedy$ Algorithm

We modify the existing deep Q-learning algorithm by incorporating a hybrid of ϵ-$greedy$ exploration and teaching via an oracle to give our novel $\mathcal{O}$-$greedy$ algorithm shown below.

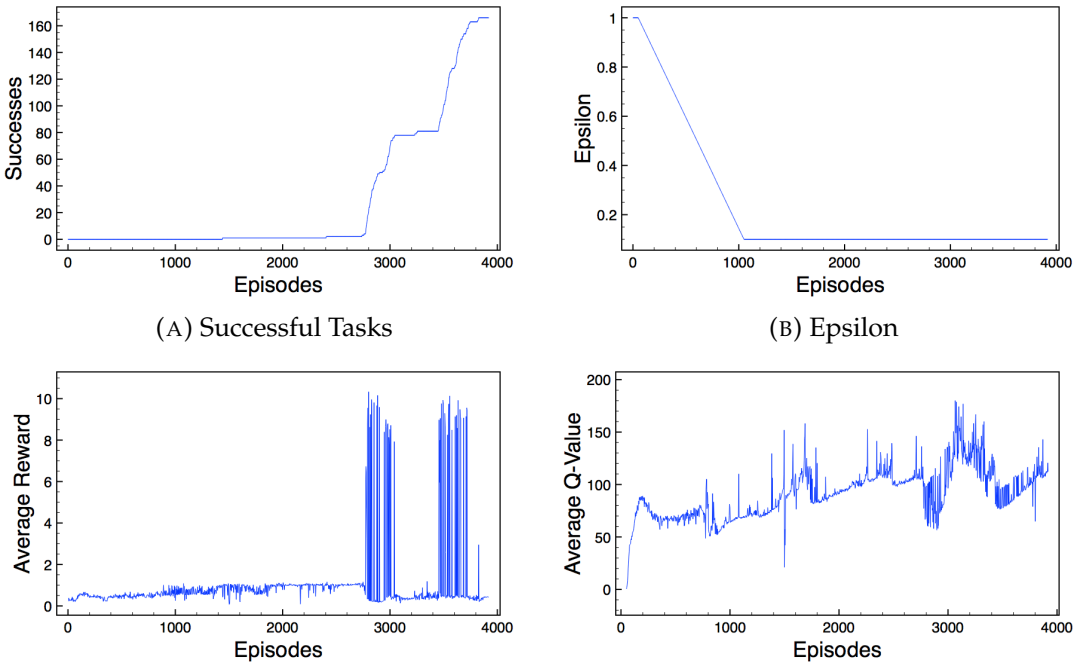
LISTING 6.1: Our novel $\mathcal{O}$-$greedy$ algorithm

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialise sequence s_1 = (x_1) and preprocessed sequenced φ_1 = φ(s_1)
    With probability O, follow follow fixed set of actions this episode
    for t = 1, T do
        if following fixed path:
            select action a_t given by oracle
        else:
            With probability ε, select a random action a_t
            otherwise select a_t = max_a Q*(φ(s_t), a; θ)
        Execute action a_t in emulator and observe reward r_t and state x_{t+1}
        Set s_{t+1} = s_t, a_t, x_{t+1} and preprocess φ_{t+1} = φ(s_{t+1})
        Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
        Sample random minibatch of transitions (φ_j, a_j, r_j, φ_{j+1}) from D
        if terminal φ_{j+1}:
            y_t = r_j
        else if non-terminal φ_{j+1}:
            y_t = r_j + γmax_{a'} Q(φ_{j+1}, a'; θ)
        Perform a gradient descent step on (y_j Q(φ_j, a_j; θ))^2
    end for
end for
```

In order to evaluate the efficiency of the proposed algorithm above, we run two similar experiments — one using ϵ-$greedy$ and the other using $\mathcal{O}$-$greedy$. As this is not the primary goal for this work, we choose to run a smaller network that does not accept images, but instead only works with joint angles. This means that we do not need our allocated machine to train the network and can instead use a common laptop, therefore not obstructing our other experiments.

The convolutional network used in our experiments so far is replaced with a 2 layer fully connected network with 7 inputs corresponding to the 6 joint angles and gripper state. Naturally, for this experiment there is a reduction in overall iteration time due to a simpler network and no need to capture and process images. As there is no image, we keep the target object and joint angles constant at the start of each episode.

We first share our results from running the standard ϵ-$greedy$ method.



(A) Successful Tasks

(B) Epsilon

(C) Average Reward

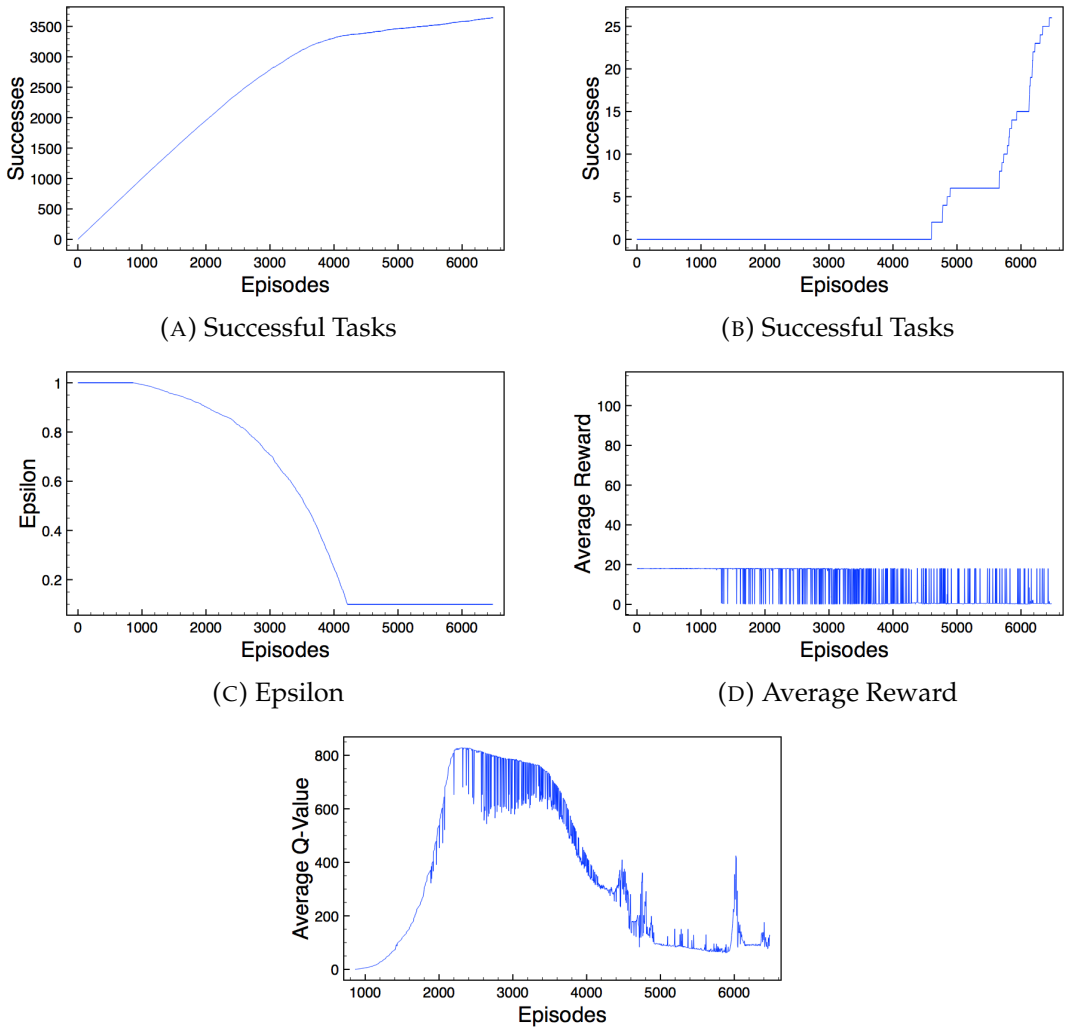
(D) Average Q-Value

FIGURE 6.18: The results of running ϵ-*greedy* with a simple fully connected network with no visual input.

The first thing to point out about these results is the greater number of episodes (shown in figure 6.18a) it takes to start succeeding frequently in comparison to the results produced by our convolutional network in section 6.3. Interestingly, at approximately episode 3000, we see a long period of unsuccessful attempts after a series of successful ones. Moreover, in figure 6.18c and 6.18d, we see a dip in both the average reward and average Q-values, suggesting evidence of unlearning. The agent then seems to spend some time re-learning what it had already learnt as shown by a steady increase in its Q-values.

Despite running a greater number of episodes in comparison to section 6.3, the running time was far less due to the smaller network training time and no need to capture images from simulation. We expect that the greater number of episodes required for training was due to the lack of visual feedback in occasions where the agent accidentally knocked the cube. If the agent had begun to learn what joint angles gave the greatest reward, but then occasionally knocked the cube, this would then cause a fluctuation in rewards that could confuse the agent.

We now replicate the experiment as above, but instead use our $\mathcal{O}$-*greedy* method, and show the results below.



(A) Successful Tasks

(B) Successful Tasks

(C) Epsilon

(D) Average Reward

(E) Average Q-Value

FIGURE 6.19: The results of running $\mathcal{O}$-*greedy* with a simple fully connected network with no visual input.

To evaluate these results, we cannot track the progress alone as with the other experiments because we would expect the agent to succeed frequently by following the oracle. We therefore track both the total number of successes (figure 6.19a) and the number of successes excluding those episodes where we were following the oracle (figure 6.19b).

At first glance, these results look like it takes longer to succeed than the previous experiment, but note the epsilon in figure 6.19c compared to figure 6.19b — there is a small delay between the time the epsilon reaches its constant value and then first stream of successes than in the previous experiment. Although the episode where it seems to have learnt a policy is higher than that in the previous experiment, the shorter delay tells us that actually this method learnt faster than the previous. The reason for the higher number of episodes are due to more episodes being completed compared to the previous experiment because ϵ amount of times we follow the oracle which will complete the episode in far less iterations than the 1000 iteration cap set per episode.

In hindsight, it would have been more clear to instead measure progress in iterations rather than episodes for both these experiments.

The average Q-values in figure 6.19e behave as we would have imagined. During high values of epsilon, we are more likely to follow the oracle and in doing so, frequently receive a high reward causing an overestimation of the Q-values in each of the states. It is not until epsilon decreases and allows for more random state exploration does the values then begin to drop and fall to more realistic values.

We believe that the comparison of these two experiments clearly shows faster training when using $\mathcal{O}$-$greedy$. Unfortunately, due to time constraints, we were unable to investigate this work further, but discuss possible future work for this in chapter 8.

6.8 Comparison Of Agents

So far we have trained several agents in different scenarios. We now take some of these agents and test their performance in their trained scenario and also in scenarios in which they were not trained. Below we summarises the agents and the scenarios in which they were trained:

- **Agent A.** Constant cube position and joint angles.
- **Agent B.** Same as agent A, but we add in the robot configuration as input to the neural network.
- **Agent C.** Variations in cube position and joint angles.

We apply each of the agents above to the two scenarios discussed so far: keeping the cube static, and moving the cube with variations in the robot configuration. Each test was run for 50 episodes with success being counted as the agent successfully lifting the cube to a designated height. The results of these tests can be seen in the table below.

Situation	Agent A	Agent B	Agent C
A: Static	56%	50%	64%
B: Variations	2%	4%	52%

TABLE 6.1: Success rates of agents applied to its trained scenario and different scenarios.

Highlighted cells indicates the success rate of an agent that is tested in the same situation in which it was trained. All the agents perform well in their trained scenarios — succeeding at least 50% of the time. We would expect these successes to increase given longer training times. We can see that both agents A and B perform poorly when applied to situation B. This was to be expected as the agent was not expected to generalise to other starting states that were not on the path to picking up the cube. Agent C on the other hand, performed well when put into a different situation, telling us that agents trained with variations are able to adapt well to situations with little or no variation.

6.9 From Simulation to Real-World

Many of today's robot control tasks are trained on real-world robotic platforms which often require human interaction, expensive equipment, and long training times. We have been incrementally working to try and remove this dependency on real-world training and instead train in simulation. We now present our findings for directly transferring trained policies from simulation to real-world.

Before preparing our agent for real-world transferring, we first needed to ensure that the simulated world resembled the real-world as much as possible. This meant that we had to make significant visual changes to our robot simulation if we had any hope of a successful transfer.

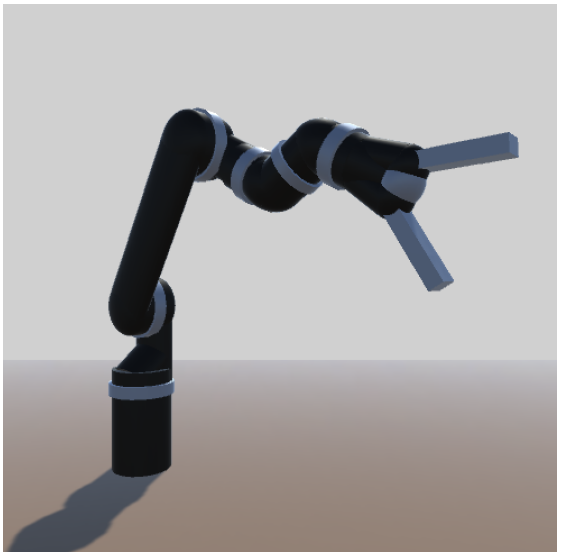
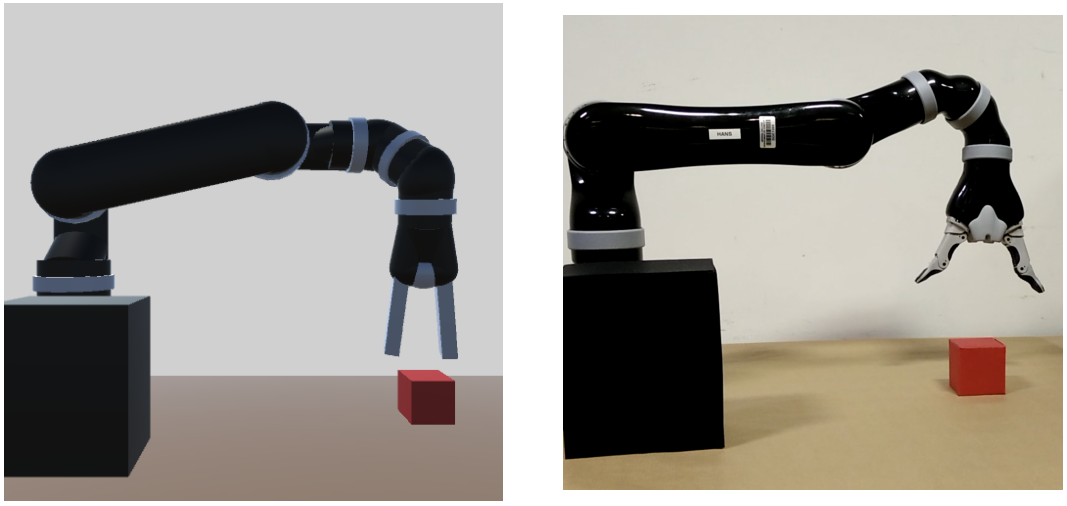


FIGURE 6.20: Robox visual improvements in preparation for transferring policies from simulation to real-world.

Above, we present our updated robot simulator which features several visual upgrades to closely match the real-world replica. To prepare for simulation training, we first set out a scene in the real world and then made final adjustments to the simulation.



(A) Simulated scene

(B) Real-world scene

FIGURE 6.21: Preparing both the virtual and real-world scenes for training. A black box in front of the base in both scenes in order to avoid modelling the clamp holding the arm to the table.

Above, we show the real world scene on the right and its simulated replica on the left. We place a black box in front of the base in both scenes in order to avoid modelling the clamp holding the arm to the table.

Due to the looming deadline, we were unable to train the agent long enough to complete the task fully and lift the cube. However, the agent had trained long enough so that it was exploring interesting areas surrounding the cube — giving us just enough to test if transferring was possible. If the real-world replica followed similar action choices to the simulated version and explore areas around the cube, then we could consider the transfer a success.

We took the network trained in simulation and ran it directly on the real-world version of the arm with epsilon fixed at 0.1. The results are incredibly encouraging. As we had hoped, the agent exhibited similar behaviour to its simulated counterpart — suggesting that transferring from simulation to real-world had been achieved.

Moreover, observing the activations in 6.22 further enforces the effectiveness by showing similar activations for both simulation and real-world. The activations on the left are from simulation while the activations on the right are from the real-world.

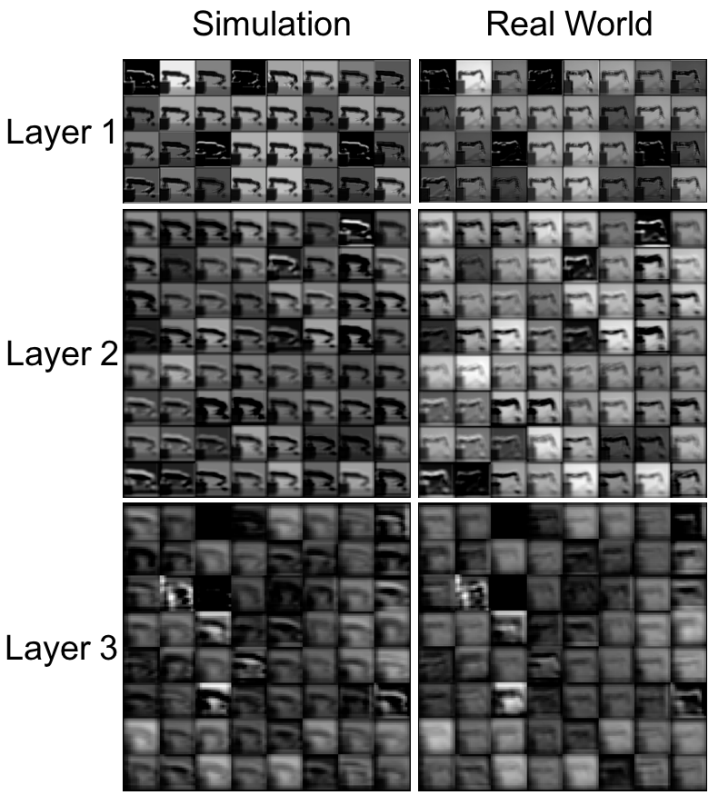


FIGURE 6.22: The activations in the feature maps of the trained network when receiving inputs from both the virtual world and real-world. Feature maps on the left are from simulation, while feature maps on the right are from the real-world.

All layers show very similar activations to the respective counterpart layers. Even though our agent was not trained with shadows, the agent still manages to transfer its policy to the real-world. We can see that our network filters the majority of shadows out from the real world in the second layer — showing the robustness of our approach.

Following the successful transfer, we hope to further train the agent in simulation to complete the task, and once again run the trained network in the real-world to show the real-world applications of this approach. We believe this is a critical step towards using robot simulations to decrease the dependency on physical robots and ultimately improve productivity of training object manipulating agents.

Chapter 7

Evaluation

In this chapter we evaluate our work via critical appraisal before comparing and contrasting with existing techniques.

7.1 Critical Appraisal

7.1.1 Strengths

- **Generalisability.** We have shown that our agents have the ability to generalise during training, can adapt to variations in the target shape, and can cope with the addition of clutter.
- **Reducing training time.** In an attempt to reduce training time we have looked at new ways for the agent to explore its environment. In section 6.7.2, we put forward the $\mathcal{O}$-$greedy$ algorithm that reduced the amount of time it took to learn successful policies. Moreover, using intermediate rewards we encouraged the agent to progress to more rewarding states in a shorted period of time.
- **Mico arm simulator.** Our final Mico arm simulator is an accurate replica of the real-world arm. We believe this tool has the potential to be used in many other cases due to its instantaneous moves and visual accuracy.
- **Numerous simulations tried.** During the course of this work, we have used both V-REP and 2 versions of Robox to train our agents. Each of these simulations brought success when teaching policies, and in doing so we have actually shown that our methods extend to different virtual environments.
- **Tests in both simulation and real-world.** We have been the first to attempt a mapping from a 3D simulation to real-world using deep Q-learning. This is an important milestone in the path to virtual training and opens up the possibility to further developments.

7.1.2 Weaknesses

- **Network architecture.** Apart from one experiment where we added in the robot configuration, the network architecture did not change during our work. Although this is reasonable given the time constraints of the project, we would have liked to experiment further with variations in the architecture to see if we could have reduced training times.

- **Lost time in early experimentation.** A large amount of time was spent waiting for experiments to run in the early stages of the project. While there were other factors, such as having being allocated only one machine in the labs, long training times were mainly due to our choice in simulator. During our background research, we investigated many third-party simulators that could simulate our robot accurately. However, it was not until we began experimenting that we realised that we did not need the majority of the features, and instead they became a burden on our training time. Although happy with progress made in the project as a whole, if we had committed to a custom built simulator from the start, then we could have potentially progressed further with our research.
- **X-axis labelling.** In the majority of our graphs throughout this work, we have been recording all of the statistics in terms of epsiodes. In hindsight, we believe that it would have been more informative to record these in terms of time-steps. For example, in section 6.7.2, we compare two methods in terms of training time. As one of these methods was more likely to succeed than the other, this resulted in the number of episodes being higher even though the number of iterations was actually lower, giving potentially misleading graphs.
- **Test results.** When tested in their trained environments, agents succeeded between 50% and 56% of the time. This is primarily due to halting the experiments at the earliest signs of learning in order to facilitate initiation of new experiments. Ideally we would have liked to allow each experiment to run for longer which may have improved the success rate.
- **Camera position.** Although we generalised to different target positions and robot configurations, we have not attempted to generalise the placement of the camera. This is important because when replicating the scene in the real-world, we cannot guarantee that the camera will be placed exactly in the same position as in the simulation, which may lead to poor transfer of policies from simulation to real-world.
- **Insufficient data in early experimentation.** We felt that the results in chapter 5 were lacking in comparison to chapter 6. This was mainly due to the majority of the work in chapter 5 being performed before writing began on the report. If given more time, we would have liked to re-run many of the experiments from chapter 5 to ensure that we record all of the necessary data.

7.2 Comparison to Existing Techniques

Our work draws heavy influence from Google DeepMind's work on training agents to play classic Atari-2600 games [64] through the use of deep Q-learning, introduced in a previous paper [65]. Like DeepMind, we also use deep Q-learning, but the underlying task we are attempting to achieve is vastly different. Due to the nature of their task, they must capture several images and concatenate them in order to supply the state. This is required to capture additional information that is not possible from one image, for example, the direction the ball is moving in pong. We however, are able to capture all the relevant information within a single image.

DeepMind train agents to play 2D games, which result in 2D images being sent through the network. Our work, on the other hand, deals with a 3D environment that brings

with it a number of issues. First and foremost, we require more information to be interpreted from our image, such as the location of both the object and the arm, the joint angles, and whether the object has been grasped by the arm. Moreover, the network must learn not only the x and y position of the cube, but also the z position — making the necessary actions to then align the joint angles so that it will grasp the cube. We also have to account for the additional time spent rendering the 3D environment in comparison to DeepMind's 2D environment.

Due to both our large state space and complex task, the probability of randomly succeeding is incredibly low in comparison to DeepMind's work. This makes learning incredibly more difficult, and requires well engineered intermediate rewards in comparison to rewards generated directly from the score of the Atari games.

What DeepMind did especially well in this paper was test a set of 49 games using the same algorithm, network architecture and hyperparameters, which reinforces the flexibility of their method. If we had more time, we would have liked to have followed a similar approach and apply our work to a larger array of tasks and shapes, as discussed in section 8.2.

Zhang et al. [107] unsuccessfully try to apply a network trained in simulation to the real-world in a target reaching task. The task they try to accomplish is significantly simpler in many ways. Firstly, they train their agent in a 2D simulation — creating a much smaller state space than our 3 dimensions. Secondly, they use a 3 joint arm in comparison to our 6 joints, further reducing the size or their state space. Finally, our task of target manipulation in the form of grasping and lifting an object is substantially more complex than a 2 dimensional target reaching task.

Zhang et al. use DeepMinds network [64] with little modification — even inputting 4 images into the network each time. As described above, the DeepMind team use such a technique because they are unable to capture the full state in a single image. Zhang et al. also use this technique, which is something that we did not do because we feel the entire state is able to be captured with only a single image, and does not need 4 images for target reaching and grasping.

Sergey Levine et al. [56] [25] attempt to tackle similar problems as our work with a few notable differences. Their work encompasses a large range of tasks, including inserting a block into a shape sorting cube, and screwing on a bottle cap to name a few. Having said that, tasks start from states in which the object in question has already been grasped, simplifying and eliminating one of the most difficult parts of the manipulation task. Our work, on the other hand, is focused more on a complete sequence of tasks, constituting object locating, grasping, and finally performing a task, such as lifting.

Sergey Levine et al. perform their tasks using a physical PR2 robot which require manual reward issuance and resetting. Our tasks are performed using a simulated Mico arm which allows for unsupervised learning and potential for a far greater number of iterations.

Pinto and Gupta [78] focus more on predicting grasp locations rather than performing the end-to-end manipulation we achieve. Although they use convolutional neural network, their architecture and output are significantly different to ours. Their network contains an additional 2 convolution layers, following a similar architecture to AlexNet [49], whereas we use an architecture similar to that in DeepMind's paper [64].

Chapter 8

Conclusion

We have shown that deep Q-networks can be used to learn policies for a task that involves locating a cube, grasping, and then finally lifting. These policies are able to generalise to a range of starting joint configurations as well as starting cube positions. Moreover, we show that policies trained via simulation are able to be directly applied to real-world equivalents without any further training.

Our journey into this exciting field has been both challenging and rewarding, with many valuable lessons learnt along the way. In this chapter, we discuss some of these lessons along with possible directions for future work.

8.1 Lessons Learnt

- **Deep Q-learning.** We have witnessed the power of deep Q-learning and its potential to be used in fields beyond video games. This now begs the question as to what else deep Q-learning can be applied to.
- **Dealing with large state spaces.** Even in a task with $256^{64\times64}$ states, it is possible to find policies to complete the task given the right reward structure that promotes exploring interesting states.
- **Powerful hardware for training.** We were well aware that machine learning solutions can take a large amount of time to train, but due to slower iteration speeds than first thought, training turned out to be significantly longer than expected. We have learnt the importance of having a powerful graphics card in order to reduce training time and therefore run as many experiments as possible.
- **Third party tools.** Third party tools can be incredibly helpful and save a lot of time if they are matched perfectly to your needs. On the flip side, they can also be an incredible hindrance in the long run if you find yourself working around features that you don't necessarily need. This was indeed the case with our choice in V-REP — the tool itself is good, but with speed being our main concern, it would have been better to create our own tool from the start — saving both time and effort.
- **Record as much data as possible.** In the initial experimentation, not enough statistics were being recorded, but this did not become apparent until progress was made on the report. In hindsight, as much data as possible should have been recorded from the start.

- **Tensorflow.** Although TensorFlow does seem to have a bright future ahead of it, the small community, lacking tutorials, and missing features caused frequent problems. In hindsight, it may have been better to use a well established machine learning library such as Torch — at least until TensorFlow had matured.

8.2 Future Work

Transferring from simulation world to physical world

Our work has shown that simulations can indeed be used to learn policies for robot control. Moreover, we have shown that we are able to transfer learned policies from simulation to the real-world — but this is only the beginning. The focus now falls on extending these methods and proving their practicality. These methods could drastically increase the learning capability of robotic agents.

Lifting different objects

Throughout our work we have focused on lifting a cube in order to simplify the overall task. We have have shown that the agent is able to adapt when given slight variations to this shape. One could extend this to study the effectiveness of our work when trained with a variety of different objects. We hypothesise that with enough training time it should be possible for our solution to generalise to other objects of similar size — for example spheres, cylinders and pyramids. These objects could then be replaced by more complex shapes such as mugs and stationary.

Performing other manipulation tasks

Locating an object, performing a grasp, and finally lifting an object could be the beginning of a larger overall task. With a means of performing these first steps, the agent could be further trained to perform additional tasks, such as stacking objects, pouring liquids, or placing an object in a designated area.

Dealing with complex environments

If robots are to be used for manipulation in complex and dynamic environments, then they must be able to deal with unseen objects without obstructing its ability to perform a task. We have shown that it is possible for an agent to generalise to variations in the target shape as well as introducing clutter to the scene without further training. Future work could explore the effectiveness of training agents in a number of different environments so that the degree in which we add variations post-training can be increased.

Investigate further the benefit or hindrance of joint angles

In section 6.4, we briefly looked at concatenating in joint angles to our neural network and found that training was no faster than a network without the angles. We suggested that the inputs could be dominated by the image and that the joint angles were not prominent in comparison. Future work could investigate this further by reducing the image size so that the inputs are dominated less by the image and allow the joint angles to carry more importance.

Investigating the $\mathcal{O}$-$Greedy$ algorithm further

In section 6.7.2, we presented our novel algorithm to reduce training time when applied to a network with only joint angles as input. Due to time constraints, no further work was able to be carried out with this algorithm. Further work could investigate how $\mathcal{O}$-$Greedy$ performs when using images as input and determine if our training times can be improved using this technique.

Alternative reinforcement methods

Our work has focused heavily on deep Q-learning, but naturally there are other possible reinforcement methods to choose from that we have not covered. Future work could investigate the use of alternative reinforcement learning methods in robot manipulation.

Alternative uses of deep Q-learning

As shown in both this and previous work, deep Q-learning has had great success in playing video games and now robot manipulation. This begs the question as to which other fields this method can be applied to. Robot locomotion often relies heavily on visual input in order to make decisions, and solving this has been attempted through reinforcement learning. One could extend existing research in this area, for both bipeds [95] [24] [27] and quadrapeds [48] through the use of deep Q-learning.

Convolutional neural networks are not restricted to images, they have also been shown to work with both video [43] and sound [74]. It would therefore be possible to study the effectiveness of deep Q-learning when these types of data are given to an agent in a reinforcement learning task.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] John Aloimonos, Isaac Weiss, and Amit Bandyopadhyay. "Active vision". In: *International journal of computer vision* 1.4 (1988), pp. 333–356.

[3] *Anykode*. http://www.anykode.com/.

[4] *AristoSim*. http://www.mtabindia.com/srobotics.htm.

[5] *Atlas*. http://www.bostondynamics.com/robot_Atlas.html.

[6] *AX On Desk*. http://www.nachirobotics.com.au/content_common/pr-simul_axondesk.seo.

[7] IA Basheer and M Hajmeer. "Artificial neural networks: fundamentals, computing, design, and application". In: *Journal of microbiological methods* 43.1 (2000), pp. 3–31.

[8] Frédéric Bastien et al. "Theano: new features and speed improvements". In: *arXiv preprint arXiv:1211.5590* (2012).

[9] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *arXiv preprint arXiv:1207.4708* (2012).

[10] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. URL: http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false.

[11] Yoshua Bengio. "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1 (2009), pp. 1–127.

[12] Jeannette Bohg et al. "Data-driven grasp synthesis - a survey". In: *Robotics, IEEE Transactions on* 30.2 (2014), pp. 289–309.

[13] David L Bowers and Ron Lumia. "Manipulation of unmodeled objects using intelligent grasping schemes". In: *Fuzzy Systems, IEEE Transactions on* 11.3 (2003), pp. 320–330.

[14] Richard R Carrillo et al. "A real-time spiking cerebellum model for learning robot control". In: *Biosystems* 94.1 (2008), pp. 18–27.

[15] Ken Chatfield et al. "Return of the devil in the details: Delving deep into convolutional nets". In: *arXiv preprint arXiv:1405.3531* (2014).

[16] Ken Chatfield et al. "The devil is in the details: an evaluation of recent feature encoding methods." In: *BMVC*. Vol. 2. 4. 2011, p. 8.

[17] Ronan Collobert et al. "Natural language processing (almost) from scratch". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2493–2537.

[18] *convnet-benchmarks*. https://github.com/soumith/convnet-benchmarks.

[19] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1223–1231.

[20] Marc Peter Deisenroth. "Learning to control a low-cost manipulator using data-efficient reinforcement learning". In: ().

[21] Li Deng. "Three classes of deep learning architectures and their applications: a tutorial survey". In: *APSIPA transactions on signal and information processing* (2012).

[22] Renaud Detry et al. "Learning object-specific grasp affordance densities". In: *Development and Learning, 2009. ICDL 2009. IEEE 8th International Conference on*. IEEE. 2009, pp. 1–7.

[23] *Dyson 360 Eye*. https://www.dyson360eye.com/.

[24] Gen Endo et al. "Learning CPG-based biped locomotion with a policy gradient method: Application to a humanoid robot". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 213–228.

[25] Chelsea Finn et al. "Learning Visual Feature Spaces for Robotic Manipulation with Deep Spatial Autoencoders". In: *arXiv preprint arXiv:1509.06113* (2015).

[26] Carlos Rosales Gallegos, Josep M Porta, and Lluís Ros. "Global Optimization of Robotic Grasps." In: *Robotics: Science and Systems*. 2011.

[27] Tao Geng, Bernd Porr, and Florentin Wörgötter. "Fast biped walking with a reflexive controller and real-time policy searching". In: *Advances in Neural Information Processing Systems*. 2005, pp. 427–434.

[28] Jared Glover, Daniela Rus, and Nicholas Roy. "Probabilistic models of object geometry for grasp planning". In: *Proceedings of Robotics: Science and Systems IV, Zurich, Switzerland* (2008).

[29] Ian J Goodfellow et al. "Multi-digit number recognition from street view imagery using deep convolutional neural networks". In: *arXiv preprint arXiv:1312.6082* (2013).

[30] *Google supercharges machine learning tasks with TPU custom chip*. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[31] *Google Turning Its Lucrative Web Search Over to AI Machines*. http://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-ai-machines.

[32] Xavi Gratal et al. "Scene representation and object grasping using active vision". In: *IROSâĂŹ10 Workshop on Defining and Solving Realistic Perception Problems in Personal Robotics*. 2010.

[33] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[34] Matthew Hausknecht et al. "A neuroevolution approach to general atari game playing". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 6.4 (2014), pp. 355–366.

[35] *How Google Translate squeezes deep learning onto a phones*. http://googleresearch.blogspot.co.uk/2015/07/how-google-translate-squeezes-deep.html.

[36] Kai Huebner et al. "Grasping known objects with humanoid robots: A box-based approach". In: *Advanced Robotics, 2009. ICAR 2009. International Conference on*. IEEE. 2009, pp. 1–6.

[37] *ImageNet*. http://www.image-net.org/.

[38] *Improving Photo Search: A Step Across the Semantic Gap*. http://googleresearch.blogspot.co.uk/2013/06/improving-photo-search-step-across.html.

[39] Anil K Jain, Jianchang Mao, and KM Mohiuddin. "Artificial neural networks: A tutorial". In: *Computer* 3 (1996), pp. 31–44.

[40] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[41] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* (1996), pp. 237–285.

[42] Ishay Kamon, Tamar Flash, and Shimon Edelman. "Learning to grasp using visual information". In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*. Vol. 3. IEEE. 1996, pp. 2470–2476.

[43] Andrej Karpathy et al. "Large-scale video classification with convolutional neural networks". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2014, pp. 1725–1732.

[44] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[45] *KINOVA MICO Arm*. http://www.robotnik.eu/robotics-arms/kinova-mico-arm/.

[46] Jens Kober, Erhan Oztop, and Jan Peters. "Reinforcement learning to adjust robot movements to new situations". In: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22. 3. 2011, p. 2650.

[47] Jens Kober and Jan Peters. "Policy search for motor primitives in robotics". In: *Machine Learning* 84.1 (2011), pp. 171–203.

[48] Nate Kohl and Peter Stone. "Policy gradient reinforcement learning for fast quadrupedal locomotion". In: *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*. Vol. 3. IEEE. 2004, pp. 2619–2624.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[50] *Languages for analytics / data mining / data science*. http://www.kdnuggets.com/polls/2013/languages-analytics-data-mining-data-science.html.

[51] Quoc V Le. "Building high-level features using large scale unsupervised learning". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8595–8598.

[52] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[53] Honglak Lee et al. "Unsupervised feature learning for audio classification using convolutional deep belief networks". In: *Advances in neural information processing systems*. 2009, pp. 1096–1104.

[54] Sergey Levine and Vladlen Koltun. "Learning complex neural network policies with trajectory optimization". In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014, pp. 829–837.

[55] Sergey Levine, Nolan Wagener, and Pieter Abbeel. "Learning Contact-Rich Manipulation Skills with Guided Policy Search". In: *arXiv preprint arXiv:1501.05611* (2015).

[56] Sergey Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *arXiv preprint arXiv:1504.00702* (2015).

[57] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Tech. rep. DTIC Document, 1993.

[58] *Luajit is so damn fast*. http://blog.mno2.org/posts/2015-02-21-luajit-is-so-damn-fast.html.

[59] Michael McCloskey and Neal J Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem". In: *The psychology of learning and motivation* 24.109-165 (1989), p. 92.

[60] *Microsoft Computational Network Toolkit offers most efficient distributed deep learning computational performance*. http://blogs.technet.com/b/inside_microsoft_research/archive/2015/12/07/microsoft-computational-network-toolkit-offers-most-efficient-distributed-deep-learning-computational-performance.aspx.

[61] *Microsoft Robotics Developer Studio*. https://msdn.microsoft.com/en-us/library/dd939239.aspx.

[62] Andrew T Miller and Peter K Allen. "Graspit! a versatile simulator for robotic grasping". In: *Robotics & Automation Magazine, IEEE* 11.4 (2004), pp. 110–122.

[63] Ajay Mishra, Yiannis Aloimonos, and Cheong Loong Fah. "Active segmentation with fixation". In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 468–475.

[64] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[65] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[66] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. "Acoustic modeling using deep belief networks". In: *Audio, Speech, and Language Processing, IEEE Transactions on* 20.1 (2012), pp. 14–22.

[67] Antonio Morales, Pedro J Sanz, and Angel P Del Pobil. "Vision-based computation of three-finger grasps on unknown planar objects". In: *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*. Vol. 2. IEEE. 2002, pp. 1711–1716.

[68] *MotoSim*. http://www.motoman.co.uk/en/products/software/product-view.

[69] *MuJoCo*. http://www.mujoco.org/.

[70] Arun Nair et al. "Massively parallel methods for deep reinforcement learning". In: *arXiv preprint arXiv:1507.04296* (2015).

[71] Andrew Y Ng et al. "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[72] *OpenAI Gym*. https://gym.openai.com/.

[73] Margarita Osadchy, Yann Le Cun, and Matthew L Miller. "Synergistic face detection and pose estimation with energy-based models". In: *The Journal of Machine Learning Research* 8 (2007), pp. 1197–1215.

[74] Taejin Park and Taejin Lee. "Musical instrument sound classification with deep convolutional neural network using feature fusion approach". In: *arXiv preprint arXiv:1512.07370* (2015).

[75] Peter Pastor et al. "Learning and generalization of motor skills by learning from demonstration". In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, pp. 763–768.

[76] Jan Reinhard Peters. "Machine learning of motor skills for robotics". PhD thesis. University of Southern California, 2007.

[77] Justus H Piater. "Learning visual features to predict hand orientations". In: *Computer Science Department Faculty Publication Series* (2002), p. 148.

[78] Lerrel Pinto and Abhinav Gupta. "Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours". In: *arXiv preprint arXiv:1509.06825* (2015).

[79] Florian T Pokorny, Kaiyu Hang, and Danica Kragic. "Grasp Moduli Spaces." In: *Robotics: Science and Systems*. 2013.

[80] Jordan B Pollack and Alan D Blair. "Why did TD-Gammon Work?" In: *Advances in Neural Information Processing Systems* (1997), pp. 10–16.

[81] *Roboguide*. http://robot.fanucamerica.com/products/vision-software/roboguide-simulation-software.aspx.

[82] *Robologix*. https://www.robologix.com/.

[83] *RobotExpert*. http://www.plm.automation.siemens.com/en_gb/products/tecnomatix/free-trial/robotexpert.shtml.

[84] *RobotStudio*. http://new.abb.com/products/robotics/robotstudio.

[85] *RoboWorks*. http://www.newtonium.com/.

[86] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. "Robotic grasping of novel objects using vision". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 157–173.

[87] Ashutosh Saxena et al. "Robotic grasping of novel objects". In: *Advances in neural information processing systems*. 2006, pp. 1209–1216.

[88] Stefan Schaal, Christopher G Atkeson, and Sethu Vijayakumar. "Scalable techniques from nonparametric statistics for real time robot learning". In: *Applied Intelligence* 17.1 (2002), pp. 49–60.

[89] Pierre Sermanet et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks". In: *arXiv preprint arXiv:1312.6229* (2013).

[90] Pierre Sermanet et al. “Pedestrian detection with unsupervised multi-stage feature learning”. In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, pp. 3626–3633.

[91] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[92] Kihyuk Sohn et al. “Efficient learning of sparse, distributed, convolutional feature representations for object recognition”. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2643–2650.

[93] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[94] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842.

[95] Russ Tedrake, Teresa Weirui Zhang, and H Sebastian Seung. “Stochastic policy gradient reinforcement learning on a simple 3D biped”. In: *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. Vol. 3. IEEE. 2004, pp. 2849–2854.

[96] Gerald Tesauro. “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[97] *Torch*. http://github.com/torch/torch7.

[98] *Unity3D*. https://unity3d.com/.

[99] *V-REP*. http://www.coppeliarobotics.com/features.html.

[100] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. PhD thesis. University of Cambridge England, 1989.

[101] *Webots*. http://www.cyberbotics.com/.

[102] D Randall Wilson and Tony R Martinez. “The general inefficiency of batch training for gradient descent learning”. In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[103] *WorkCellSimulator*. http://www.it-robotics.it/products/3d-simulation/workcellsimulator/.

[104] *Workspace*. http://www.workspace5.com/.

[105] Dong Yu et al. *An introduction to computational networks and the computational network toolkit*. Tech. rep.

[106] Matthew D Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks”. In: *Computer vision–ECCV 2014*. Springer, 2014, pp. 818–833.

[107] Fangyi Zhang et al. “Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control”. In: *arXiv preprint arXiv:1511.03791* (2015).

Appendix A

Headless OpenGL

This appendix applies only to the computing labs at Imperial College London.

When running programs for long periods of time, it is not always possible to remain physically logged in to lab machines. This is generally overcome by instead connecting to a machine via *ssh* and keeping a persistent session through a terminal multiplexer such as *tmux*. Programs running in this manner will not have access to a physical display and so a display server such as *Xvfb* can be used to run the application 'headless'. Unfortunately, OpenGL applications fail to run heedlessly using conventional means which we believe is caused by conflicting graphic drivers that we are unable to solve due to restricted super user access.

After numerous attempts at running our OpenGL application via *ssh*, we document our final solution below for future students at Imperial who also come across this problem.

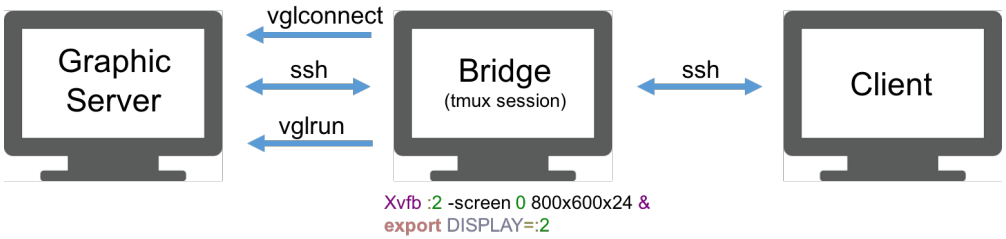


FIGURE A.1: Headless OpenGL

N.B. although the Graphics Server and Bridge are represented as two separate machines, they can in fact be the same machine.

1. Establish a ssh connection from your client machine to the machine where you will run our OpenGL application.
2. Start a display server: *Xvfb :2 -screen 0 800x600x24 &*.
3. Set the virtual display as the display: *export DISPLAY=:2*.
4. Connect to the same machine using VirtualGL: *vglconnect -s <machine_name>*. This will establish another *ssh* connection to the same machine that rendered content will be sent through.
5. Start your application: *vglrun <OpenGL_application>*