Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Learning how to Grasp Objects with Robotic Gripper using Deep Reinforcement Learning

Author:
Jiaxi LIU

Supervisor:
Dr. Edward JOHNS

Submitted in partial fulfillment of the requirements for the MSc degree in MSc in Computer Science Specialist of Imperial College London

September 2017

Abstract

Training robots to interact with object is an interesting task for researchers in robotic manipulation field. Recently, with the help of deep learning, methods became more intelligent and efficient. Training a robot griper with trial-and-error is the core idea behind the family of reinforcement learning algorithms. Aiming at helping robot agent to act reasonably at the beginning of interacting with the objects, we explore using two different kinds of deep reinforcement learning algorithms: deep Q-networks (DQNs) and deep Q-learning form Demonstrations (DQfD). We design and build up the whole experiment environment for realising these two algorithms. Especially for DQfD, we set up different kinds of simulation environments to sample the grasping space from the surface of the object and output the best grasping positions; we use these outcomes to generate demonstrations for the algorithm, and pre-train the agent by these demonstrations. Our experiment results demonstrate that the agent trained by DQfD algorithm could not only behave more reasonably at the beginning of interacting with the environment, but also perform better with limited training episodes. Moreover, we test our learning algorithm with different input state representations: discrete sparse vectors, matrices and continuous variables; we test different neural network models as well as different loss functions in reinforcement learning algorithm and compare their results. We also show the situation that the robot agent trained by DQfD algorithm working with complicated models. We conclude that our system for training robot agent with DQfD is efficient and accurate.

Acknowledgments

I would like to thank:

- My supervisor, Dr. Edward Johns, for his consistent encouragement and patience. He gave me great help by providing me with necessary materials, advices of great value and inspiration of new directions.
- My family for their unconditional love and support, especially for a second master degree abroad.
- My girlfriend for her understanding and wonderful cooking skills.
- My best friend Bruno Lansac for his climbing guidance, which cured my backache and neckache.

Contents

Chapter 1

Introduction

Grasping a pen, a book or a mug, could be so easy and natural for grown-up human-beings, no need for suspicion or wild imaginings. But for a robot, it is another story. Like a newly born baby, it could be almost impossible for it to grasp a toy at the first try, or the second. It requires a lot of practices and learning from the fails, before the first success.

As for a robot, even if we assume that it has somehow found and recognised the object and equipped with all it needs to grasp, there are still a lot of problems it needs to solve before a successful grasp. First and foremost, it needs some basic knowledge about the object. Reliable robotic grasping is widely believed to be challenging due to uncertainty about many properties such as object shape, pose, materials and mass [42]. Then, after briefly knowing the object, it needs some strategies or policies to follow. To grasp at a right position but with an inappropriate orientation, or without choosing a good gesture could still end up with failure. But a failed experience can be meaningful if it could be taken advantage of, at least it excludes a wrong choice. Furthermore, if the failure is judged-able, it could be used to optimise the strategy or policy the robot is using, which is the idea behind reinforcement learning.

Reinforcement Learning, an exciting method in machine learning, has rapidly been developed and applied in many domains, such as control theory, operation research, game theory, etc. It is aiming at the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment [32]. The robot trying to learn to grasp an object could be regarded as an agent, interacting with an environment for the grasping task [9].

In this grasping task for instance, the position, orientation, even the degree of freedom for very joint of the robot hand could would be the inputs for the reinforcement learning algorithm. One main challenge in reinforcement learning algorithm designing is that the state spaces the agent facing to be very large or continuous, potentially causing the results that the state-action value functions may be hardly represented comprehensively [11]. With high-dimensional inputs, using deep neural networks as a function approximater may be very powerful [5]. This method is deep reinforcement learning. Using deep neural networks trained on large datasets of human

grasp labels [40] or physical grasp outcomes [57] can be used to plan grasps for objects. While aiming to a certain unknown object, using reinforcement learning by gradually accumulating agent experience, could be another approach to generate grasping policy.

Recently, research has proved that the method called Deep Q-Learning from Demonstration (DQfD)[22], using demonstration data to pre-train the agent could help it perform better from the start of learning, even if the demonstration data is not very good. It is a effective way for the circumstances have data of the system operating under a previous controller, either human or machine.

Different from the original deep Q-learning algorithms, DQfD pre-trains the agent with demonstrations, which may have easier accessibilities than a accurate simulation environment. And it can make use of human or other agent's experience. Furthermore, it is proved that, the pre-trained agent could perform more reasonably at the beginning of interacting with the environment [22], which is also a very good quality for robotic manipulation. It could: 1) speed up the robot training process; 2) make it possible to send robot agent to interact with environment directly after pre-training; 3) prevent the robot agent from undesirable actions. Therefore, our project is trying to benefit form all these good qualities, to build up an experiment system for realising DQfD algorithm, training robot gripers to interact with objects.

1.1 Objectives

Our goal is to research the way to train different kinds of robot with different gripers to learn to interact with different objects, especially to grasp these objects. Achieving this would open up the possibility of the robot agent learning to interact with the environment without prior knowledge. In order to accomplish this, we have to:

1. Understand existing solutions. To realise this project we firstly need to have a clear understanding of the knowledge in deep reinforcement learning. We need to be well known of both core concepts as well as popular methods in both deep learning and reinforcement learning. Furthermore, we need to be able to understand the top state-of-art work and comparing them with our proposed work.

2. Explore machine learning libraries and robot simulation environments. To be able to use the methods in deep reinforcement learning, we need to be familiar with the related machine learning libraries. And to carry out our experiments, we need the help of some open source robot simulation environments. We should build up these environments and modify them aiming our project.

3. Implement a training framework. We need to implement a framework to realise the idea of the project, to train a robot hand to interact with objects. It should contain the whole procedure from collecting training samples to on-line

training.

4. Compare and improve algorithms and structures. We should compare different learning algorithms as well as neural network structures, to find out the advantages and disadvantages. Then, we need to improve both the algorithms and structures aiming our task and need.

1.2 Challenges

The work supporting this project is very challenging due to the aim of a large scale of problem with little prior experience. During the course of the project, the biggest challenges encountered were:

1.Experiment planning. We should design the experiments carefully to focus on the essential problems while simplifier some other conditions and then maximise the meaning of the research in this task with limited time and hardware equipment, especially when we are facing a complex state space and long term iteration.

2.Combining theories with practice. We need to have a good knowledge about the related theories, in order to put them into use as well as make improvements. But also, we need to be able to handle some project realization problems while using these theories, especially judge the cause of some unsatisfactory results.

3.Choosing and building up different environments. There are a lot of usable simulation environments and libraries under different operation systems with different pros and cons. It is clear that we could not carry out the whole project with just any single of them. Therefore, we need to choose a combination and make them work together. Without good investigation about the environments before using, could later on cause server problems, crashes or even being in dead corners. Then, carefully building them up one by one should also pay attention to the potential contradictions between different environments and libraries.

4.Handling newly outcome libraries and models. As the cutting-edge of combining robotic science with deep learning algorithms, the libraries imported to support our project could be newly developed and published, so as to some models in the simulation environments. They could be somehow containing errors and bugs. We need to be able to fix them when we are taking advantage of their newly attributes.

1.3 Contributions

We have demonstrated that DQfD have the ability to be used for learning control policies for robot interaction with objects. Along the way we have tried many variations that we hope will be insightful to future work. Our key contributions are:

1. Summerised two combinations of simulation environments. To carry out the project, we need a simulation environment to sample the grasps and another one to on-line learning. The first combination we summarised is using GraspIt! to sample and MuJoCo to train. The second combination is using Dex-Net 2.0 to sample while MuJoCo to train. The pros and cons of these two combinations will be discussed later.

2. Summerised a experiment structure. To do the on-line learning, we build up a client-sever structure using HTTP protecol. The deep reinforcement learning calculation will be carrying out using TensorFlow in a Python program, as the client, while the MuJoCo, as the server, will receive the command from TensorFlow and carry out the simulation.

3. Verified Deep-Q(λ) Learning. Aiming our unique task, We realised and verified a method Deep Q(λ)-Learning, combining the idea of Deep Q-Learning and Q(λ)-Learning. We tested and proved it efficient.

4. Compared different combinations of neural network and reinforcement learning algorithm. Since we are using deep reinforcement learning in this project, we need combine deep learning and reinforcement learning. We tested and compared several combinations of different neural networks and reinforcement learning algorithm.

5. Designed and optimised models. We have three different kinds of simulation environments in this project. Each of them has its own model structure. In order to make them work together, we designed and optimised same model in different simulation environments.

1.4 Outline of Contents

The rest of this report is organised as follows. Chapter 2 introduces the background material covering the family of reinforcement learning algorithms, different artificial neural networks, robotic manipulation and techniques with simulating grasping. Chapter 3 introduces the options available when building up the experiment system, mainly in three directions: the machine learning library, the simulation environments, the networking library. We also introduces the structure for them and

the building up process.

Experimentation is divided into two chapters. In chapter 4, we discusses some early preliminary experiments we did. During these experiments, we gained a deeper understanding of the task we are facing and the tools we have. Also we concluded output forms and evaluation mechanism for our experiment. Following that, chapter 5 introduces the whole DQfD experimental system we implemented, and the core experiments we did. We evaluates our project in Chapter 6 with the advantages and disadvantages for this project. Finally, in chapter 7 we concludes the lessons we learned and some suggestions for future work.

Chapter 2

Background

In this project, the mean method we are using is deep reinforcement learning. We are going to introduce this family of methods in this chapter. We firstly introduce reinforcement learning, and then deep learning, and finally deep reinforcement learning.

2.1 Reinforcement Learning

Reinforcement Learning [65], an exciting method in machine learning, has rapidly developed and applied in many domains, such as control theory, operation research, game theory, etc. It is aiming the problem faced by an *agent* — the learner and decision maker, that must learn behavior through trial-and-error interactions with a dynamic environment [32].

In a standard reinforcement learning model, an agent is connected with environment through perception and action, as shown in Figure 1. Given time step t, the state for both agent and environment could be represented as $s \in S$, and for this state s, an action $a \in A$ can be performed by the agent. After taking an action, the agent will be a new state, and receive a reward $r \in R$. The strategy for the agent to take an action in a certain state is policy π, which should tend to increase the long-run sum of reward. The agent could manage to do this over time by systematic trial-and-error .

Formally, the model consists of:

- a discrete set of/continuous environment states, S;
- a discrete set of/continuous actions, A;
- a set of scalar reinforcement rewards, R.

The agent's job is to find a plicy π, mapping states to actions, that maximises some long-term measure of reinforcement. Usually, it is assumed that, in general, the environment will be non-deterministic: taking the same action a in the same state s on two different time steps t_1 and $t_2$may result in different next states and/or different

rewards. However, it is also assumed that, the environment is stationary: the probabilities of makeing state transitions or receiving rewards do not change over time [2].

Reinforcement learning has several differences from supervised learning. The most important difference is that there is no presentation of sample and label pairs. Instead, the agent will be given the reward and subsequent state immediately after choosing an action. While some aspects of reinforcement learning share commons with search and planning issues in artificial intelligence, which require a predefined model of state transitions.

2.1.1 Models of Optimal Behavior

Deciding the optimality of the model shall be the fundamental of the algorithms of learning to behave optimally. First and foremost, it is important to specify how the agent take the future into account in the decisions it makes about how to behave at its present state. There are three different models to judge this subject in this area that is widely used.

The finite-horizon model is making the agent to optimise its expected reward for the next h steps at a given time step $t = t_0$:

$$E(\sum_{t=t_0}^{h} r_t); \tag{2.1}$$

r_t represents the scalar reward received at time step t.

The second model is average-reward model. The agent using this model is supposed to take actions that maximise its long-term average reward:

$$\lim_{h\to\infty} E(\frac{1}{h}\sum_{t=t_0}^{h} r_t); \tag{2.2}$$

But one problem with this model is that there is no way to distinguish between two policies that one gains a large amount of reward in the first few time steps while the other does not.

The last model is infinite-horizon discounted model. It takes all of the long-term reward of the agent into account with a geometrically discount factor γ (where $0 \leq \gamma < 1$):

$$E(\sum_{t=t_0}^{\infty} \gamma^t r_t). \tag{2.3}$$

γ can be seen as an interest rate, a probability of living another step, or a mathematical trick to bound the infinite sum [2]. This model is more mathematically tractable than the finite-horizon model, which caused this model widely used. Therefore, in our project, we are going to use this infinite-horizon discounted model.

2.1.2 Measuring Learning Process

The three models discussed in the previous section can assess the policies learned by the agent with a given algorithm. Furthermore, it is required to evaluate the quality of learning process itself. There are a few measures we would like to discuss.

- **Eventual convergence to optimal**. This may be first rule of designing the algorithm, while many of them do come with a provable guarantee of asymptotic convergence to optimal behaviour [70].
- **Speed of convergence to near optimality**; Since optimality is an asymptotic result for the most of time, it is practical to use the speed of convergence to near optimality as a measurement rather than convergence to optimality.
- **Regret** [8]. It is the difference between the optimal reward and the actual reward. This measure is the expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally at the very beginning.

2.1.3 Exploitation and Exploration

It is one of the major difference between reinforcement learning and supervised learning that, a reinforcement-learner must explore the environment properly during its learning, while on the other hand, take advantage of its learned knowledge. A very good case of exploitation and exploration shall be the k-armed bandit problem [8]: the agent is in a room with a collection of k gambling machines. The agent is permitted a fixed number h of pulls. On each turn any arm may be pulled without requiring a deposit but only costing a pull playing a suboptimal machine. If arm i is pulled, machine i pays off 1 or 0, depending on some underlying probability p_i. The payoffs are independent events and the p_is are unknown. The agent should generate a policy to maximise the total of payoffs.

The agent may believe one particular arm has higher payoff probability than the rest; Then, if it find out one seems to be with high probability, should it choose that arm all the time, or should it choose another one that it is not familiar with? There are a wide variety of solutions to this problem. But generally, the answer to these questions depend on how many times the agent is able to pull arms; the more the times to pull, the worse the consequences of prematurely converging on a sub-group of arms, and the more the agent shall explore.

2.1.4 Markov Decision Processes

In the most of the cases of reinforcement learning problem, the agent's policy is not only aiming in getting more immediate reward, but also the next state of the environment.Therefore, the agent have to learn from delayed reinforcement: it may take a long sequence of actions, receiving insignificant immediate reward, but finally arrive at the state with high reward, the reward take place arbitrarily far in the

future. The problem with delayed reward are well modeled as Markov Decision Processes (MDPs) [6].An MDP consists of

- a set of states S;
- a discrete set actions, A;
- a reward function $R : S \times A \to \Re$, and
- a state transition function $T : S \times A \to \Pi(S)$, where a member of $Pi(S)$ is a probability distribution over set S.

Within MDP environments, we need techniques for determining the optimal policy given the infinite-horizon discounted model we mentioned before, as we rely on the fact that it has an optimal deterministic stationary policy [6].

The optimal value of a state is the reward that the agent will gain if it starts in that state and executes the optimal policy:

$$V^*(s) = \max_\pi E(\sum_{t=t_0}^{\infty} \gamma^t r_t). \tag{2.4}$$

And this optimal value function can be defined as

$$V^*(s) = \max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')), \forall s \in S. \tag{2.5}$$

With the optimal value function, we can specify the optimal policy as

$$\pi^*(s) = \arg\max_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')), \forall s \in S. \tag{2.6}$$

And one way to find the optimal policy is to find the optimal value function, which can be determined by iterative algorithm:

ALGORITHM: VALUE ITERATION

initialise V(s) arbitrarily
while policy not good enough
  for $s \in S$
    for $a \in A$
      $Q(s,a) := R(s,a) + \gamma \sum T(s,a,s')V(s')$
    V(s):=$\max_a Q(s,a)$
  end for
end for

Rather than finding via indirectly using optimal value function, another iteration algorithm is policy iteration, and it manipulates the policy directly. It operates as follows:

ALGORITHM: POLICY ITERATION

choose an arbitrarily policy π'
  do $\pi := \pi'$
    compute the value function of policy π:
      solve the linear equations:
        $V_\pi(s) = R(s, \pi(s)) + \gamma \sum T(s, \pi(s), s')V_\pi(s')$
      improve the policy:
        $\pi'(s) := \arg\max_a(R(s, a) + \gamma \sum T(s, a, s')V_\pi(s'))$
  until $\pi = \pi'$

2.1.5 Model-free Methods

In the previous sections we discuss the methods to obtain optimal policies for MDPs with assuming the model itself is well-known, which consists of the state transition probability function$T(s, a, s')$and the reward function $R(s, a)$. Sometimes the model is unknown and the agent must interact with the environment directly to obtain the information. Therefore, there are two ways.

- **Model-free:**Learn a controller without learning a model;
- **Model-based:**Learn a model, and use it to derive a controller.

This project will focus on the former one, since in this project we want the robot to learn from the environment with the states, actions and rewards. Thus, we will discuss some popular model-free methods.

2.1.6 Monte Carlo Methods:

Monte Carlo (MC) Methods is a model-free method and works by averaging the returns from the environment on an episode-by-episode baisis. And its process is:

ALGORITHM: MONTE CARLO METHODS

choose an policy π to evaluate
choose an arbitrarily state-value function V
let R(s) be an empty list, for all states
while not good enough
  generate an episode form π
  for each state s in the episode:
    $r := \sum_n \gamma^n r_n$
    Append r to R(s)
    $V(s) \leftarrow average(R(s))$
  end while

2.1.7 Temporal-Difference Learning

Temporal-difference (TD) learning also learns directly from the environment as a model-free methods. Unlike MC methods wait until the end of every episodes to

learn, TD learning waits for only one time-step and use temporal errors to update the value. The simplest TD learning is TD(0):

$$V(s) := V(s) + \alpha[r + \gamma V(s') - V(s)]. \tag{2.7}$$

Whenever a state s is arrived, its estimated value will be updated to be closer to $r + \gamma V(s')$. If the learning rate α is adjusted properly and the policy is held fixed, TD(0) is guaranteed to converge to the optimal value function. The general TD(λ) is similar to the TD(0) we discussed above,

$$V(u) := V(u) + \alpha[r + \gamma V(s') - V(s)]e(u) \tag{2.8}$$

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k} \tag{2.9}$$

where

$$\delta_{s,s_k} = \begin{cases} 1 & if\ s = s_k \\ 0 & otherwise \end{cases}$$

TD(λ) often converges in a considerably fast speed if used large enough λ, but is computationally expensive at the same time.

2.1.8 Sarsa

A special version of TD control algorithm is Sarsa, derived from the experience tuple $< s, a, r, s', a' >$. Each tuple records that the agent starts from state s, carries out action a, received reward r, move to states s', and decides to do action a'. This tuple could be used to update $Q(s, a)$ using the equation:

$$Q(s, a) := Q(s, a) + \alpha[r + \gamma Q(s', a') - V(s)]. \tag{2.10}$$

And its process is:

ALGORITHM:SARSA

```
1  initialise Q(s,a) arbitrarily
2  for e in episodes:
3     initialise s
4     choose a from s using policy derived from Q
5     for each step in episode:
6        take action a, receive r, s'
7        choose a' from s' using policy derived from Q
8        Q(s,a) := Q(s,a) + α[r + γQ(s',a') − V(s)]
9        s:=s'
10       a:=a'
11       end for
12    end for
```

2.1.9 Q-learning

Q-learning [71] is an off-policy TD control algorithm that directly approximate $Q^*(s,a)$ independent of the policy being followed and be written recursively as

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max'_a Q^*(s',a'). \tag{2.11}$$

since $V^*(s) = \max_a Q^*(s,a)$, we have $\pi^*(s) = \arg\max_a Q^*(s,a)$ as the optimal policy. One step further, we could have the Q-learning rule:

$$Q(s,a) := Q(s,a) + \alpha[R(s,a) + \gamma \max'_a Q^*(s',a') - Q(s,a)]. \tag{2.12}$$

where $< s, a, r, s' >$is an experience tuple similar to the one we discuss with Sarsa. If each action is executed in each state an infinite number of time-steps and α is decayed appropriately, the Q values will converge to Q^* [71].

Q-learning can be extended as in TD(λ) to update states that occurred more steps, to be Q(λ)-learning [56]. Since Q(λ)-learning propagates information rapidly, it works significantly better than one-step Q-learning on a number of tasks and its basis in the integration of one-step Q-learning and TD(λ) returns makes it possible to take advantage of them as well as be a potential bridge between them [56]. Using n-step returns helps propagate the values of the trajectory:

$$R := r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a Q(s_{t+n}, a). \tag{2.13}$$

2.1.10 Generalisation

All of the previous discussion has assumed the possibility of enumerating the states and actions, and therefore a big enough table to store their values, which is called the Q-table. At the same time, these algorithms makes inefficient use of experience, especially when the state space is large and smooth and there are a huge number of states will have similar values as well as optimal actions. And this is one of the inner driven for developing deep reinforcement learning [50], which combined reinforcement learning with a class of artificial neural network [25] known as deep neural networks. In the next section, we discuss this deep neural networks.

2.2 Artificial Neural Networks

Artificial Neural Network (ANN)[3], inspired by biological neural networks has been proved to be able to solve a variety of problems. ANN research has experienced three periods of extensive activity. The first peak in 1940s was due to McCulloch and Pitts' pioneering work [44]. And then in 1960s, Rosenblatt's perceptron convergence theorem [59] and Minsky and Paper showed the limitations of a simple perceptron [52] dampened the enthusiasm of most of the researchers for this area and lasted almost 20 years. Until early 1980s, ANN have again received considerable renewed interest, mostly because of the development behind Hopfield's energy approach [26] in 1982

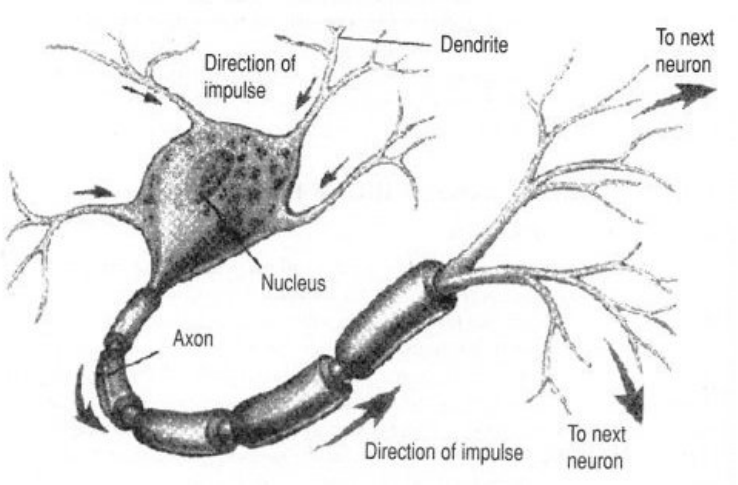

Figure 2.1: Schematic of a biological neuron [10]

and the back-propagation learning algorithm for multilayer perceptrons by Werbos [72], and then popularised by Rumelhart et al [60]. in 1986.

A neuron is a certain kind of biological cell that processes information, which contains a cell body and two types of out-reaching tree-like branches: the axon and the dendrites (see Figure 2.1). The cell body has a nucleus containing the information about hereditary traits and a plasma to hold the molecular equipment for producing material that the neuron needs. A neuron receives impulses from other neurons through its dendrites and transmits impulses by its axon.

2.2.1 Artificial Neurons

McCulloch and Pitts [44] designed a binary threshold unit to simulate the biological neuron as an artificial neuron (see Figure 2.2).

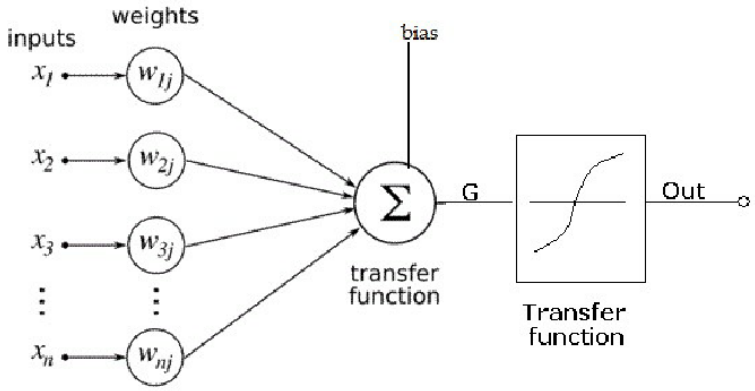

Figure 2.2: Artificial neuron model [67]

Artificial neuron computes a weighted sum of n inputs signals $x_i, i = 1, 2, ..., n$, with

biases, and generates an output of 1 if the sum is above a certain threshold. Otherwise it outputs 0 as the results:

$$y = h(\sum_{i=1}^{n} w_i x_i + b_i) \tag{2.14}$$

where $h(\cdot)$ is the unit step function, also known as the activation functions, and w_i is the synapse weight associated with the ith input.

2.2.2 Structure of ANNs

ANNs can be viewed as weighted directed graphs. In these graphs, artificial neurons are nodes and directed edges are the connections between neuron outputs and neuron inputs. Based on the connection pattern, the architecture of the structure, ANNs can be grouped into two main categories (see Figure 2.3):

- **feed-forward networks**,in which graphs have no loops;
- **feedback networks**, in which loops occur due to feedback connections.

In the most common feed-forward networks, also called multilayer perceptron, neurons are organised into layers with unidirectional connections between them. And generally speaking, feed-forward networks are static and produce only one set of output values. On the other hand, feedback networks are dynamic systems. Because of the feedback paths,the modification of the inputs to neurons leads the network to enter a new state.

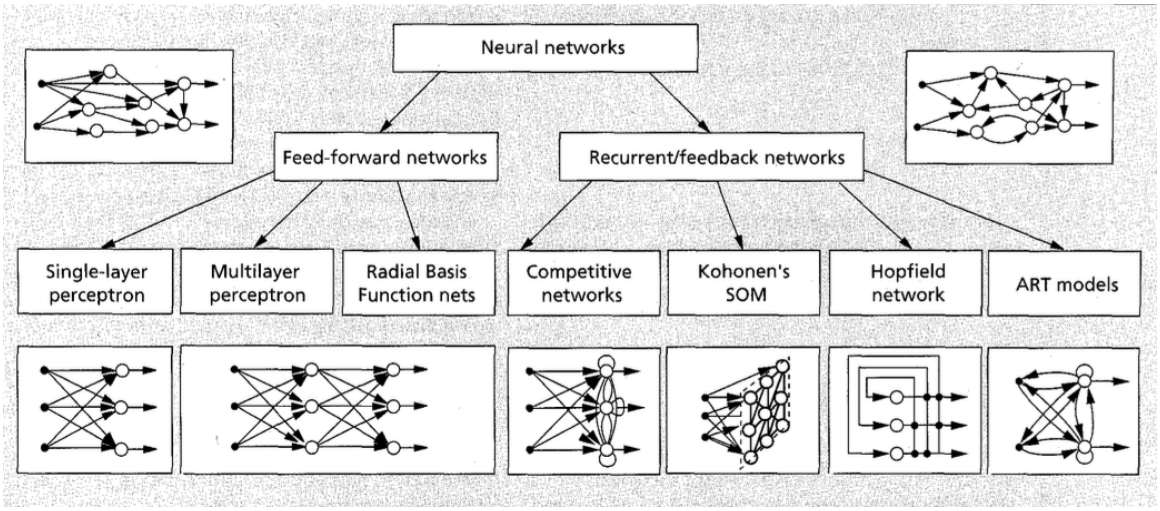

Figure 2.3: Different neural networks architectures [27]

2.2.3 Learning Process

The fundamental trait of intelligence is the ability to learn. Although a precise definition of learning may be difficult, one learning process for the ANN context could be

updating its network architecture and connection weights in order to make the network more efficient. And this process usually being down by learning the available training patterns, and the network's performance is improved over time by iteratively updating its weights. Without the need for human experts to set out learning rules, one of the most attractive property for ANNs is that they can matically learn from examples, especially the underlying rules from given collections of representative examples. The learning algorithm refers to this procedure in which the network use the learning rules for adjusting the weights. In this project, we use gradient descent to update these weights.

Gradient descent is a popular method in the field of machine learning. It is a first-order iterative optimization algorithm for finding the minimum of a function. It takes steps to the negative of the gradient of the function to find a local minimum:

$$w := w - \alpha \frac{\partial F}{\partial w} \tag{2.15}$$

where α is the learning rate determining the size of the steps the algorithm taken to minimise the function F.

2.2.4 Deep learning

Deep learning, also known as deep structured learning or hierarchical learning, is family of machine learning methods. It has some features:

- **Multi-layers structure**. Deep learning use a cascade of many layers of nonlinear processing units for feature extraction and transformation. Each layer uses the output from the previous layer as input;
- **Learning of multiple levels of features**. To have a hierarchical representation, higher level features are derived form lower level features.
- **Learning of multiple levels of representations**. Different levels form different hierarchy of concepts and the representations correspond to different levels of abstractions.

Deep learning architectures such as deep neural networks [62], deep belief networks [24] and recurrent neural network [21] have been proved effective in many fields such as face detection [53], visual recognition [64] [37], audio recognition [39] [51], pedestrian detection [63], and natural language processing [14]. In this project, we use a group of deep neural networks as the deep learning architecture.

A deep neural network (DNN) is an ANN with multiple hidden layers between the input and output layers. DNNs could model complex non-linear relationships, by its higher layers enabling composition of features from lower layers., potentially modeling complex data with fewer units than a similarly performing shallow network [7].

A typical set of a DNN model is shown in Figure 2.4. Within this DNN, layer k computes an output vector $\mathbf{h}^k$ using the output h^{k-1} of the previous layer, starting with the input $\mathbf{x} = \mathbf{h}^0$,

$$\mathbf{h}^k = tanh(\mathbf{b}^k + W^k \mathbf{h}^{k-1}) \tag{2.16}$$

with parameters $\mathbf{b}^k$ (a vector of offsets) and W^k (a matrix of weights). And the tanh

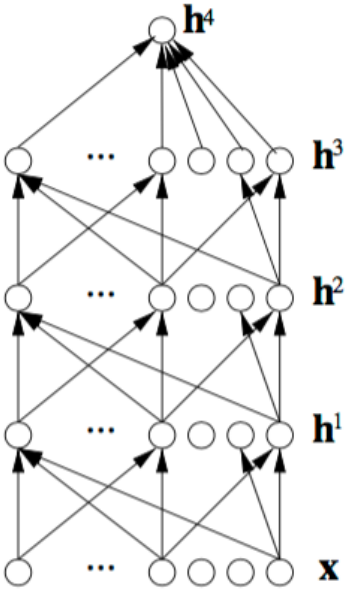

Figure 2.4: A typical structure of DNN model

can be replaced by $sigm(u) = 1/(1 + e^{-u})$ or other saturating non-linearities.

The top layer output $\mathbf{H}^l$ is used for making prediction and shall be combined with a target y to form the loss function $L(\mathbf{H}^l, y)$, and then may have a non-linear function different from other layers. A commonly used one is softmax function $\mathbf{x} = \mathbf{h}^0$, with:

$$\mathbf{h}_i^k = \frac{e^{\mathbf{b}_i^l + W_i^l \mathbf{h}^{l-1}}}{\sum_i e^{\mathbf{b}_i^l + W_i^l \mathbf{h}^{l-1}}} \tag{2.17}$$

where W_i^l is the ith row of weights, $\mathbf{h}^l$ is positive and $\sum_i \mathbf{h}^l = 1$.

2.2.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) [38] were inspired by the visual system structure and then organised in two types of layers: convolutional layers and sub-sampling layers. Each layer has a topographic structure making each neuron associated with a fixed two-dimensional position that corresponds to a location in the input image. The idea behind these setting is to overcome two problems of using normal DNNs for image inputs. First, typical images could be quite large and then fully connected first layer could results in a large number of weights. Second, the

ineffectively using of the inner structure of the input, which can be presented in any order without affecting the outcome.

A typical CNN dubbed LeNet-5 [38] is shown in Figure 2.5. The input of this CNN is an image of characters that are firstly approximately size normalised and centered. Every unit in a layer receives inputs from a group of nodes located in a small neighborhood in its previous layer. In the first hidden layer of LeNet-5, units are organised in six feature maps. A unit in a feature map has 25 inputs from a 5×5area in the input image, and therefore has 25 trainable coefficients plus a bias. Different feature maps in the same layer use different set of weights and biases, thereby extracting different types of local features. The second hidden layer of LeNet-5 is a subsampling layer, comprises six feature maps. The receptive field of each unit is a 2×2 area in the previous layer, and each unit computes the average or maximum of its four inputs, multiplies it by a trained coefficient, adds a trainable bias and passes the result through a sigmoid function. Successive layers of convolutions and subsampling are typically alternated resulting in a "bi-pyramid": the number of feature maps is increased as the spatial resolution is decreased.

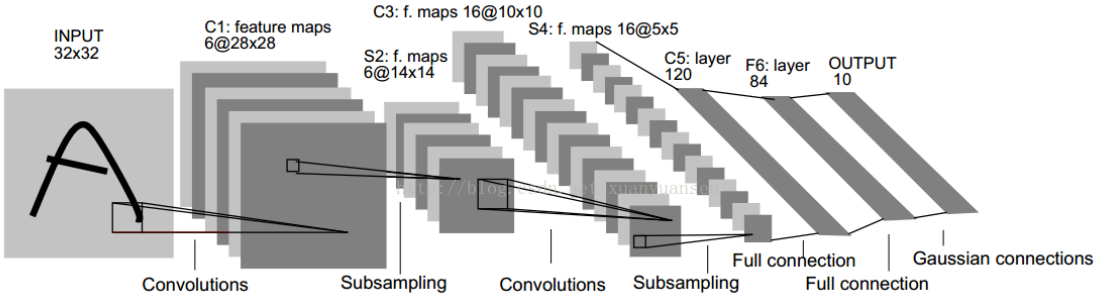

Figure 2.5: Architecture of LeNet-5, a CNN [38]

Fixed-size CNN have been applied to many applications, such as handwriting recognition, machine-printed character recognition, on-line handwriting recognition and face recognition.

2.2.6 Deep Reinforcement Learning

After discussing the reinforcement learning and deep learning, we now could move on to the family of methods combined them together. One main challenge in reinforcement learning algorithm designing is that the state spaces the agent facing may be very large or continuous, potentially caucusing the results that the state-action value functions may be hardly represented comprehensively [11]. Therefore, the tasks reinforcement learning agents achieved are mostly in domains that fully-observed, low-dimensional state spaces. However, with high-dimensional sensory inputs, using deep neural networks as a function approximater proved to be effective that the agent could learn successful policies directly from high-dimensional sensory inputs [50]. Researchers using one of the deep reinforcement learning method called

deep Q-learning[49] could even performed at human-expert level playing Atari 2600 games.

There are two techniques deep Q-learning used to address the instabilities within original Q-learning:

- **experience replay**, which randomises over the data to remove correlations between observation sequences and smooth over changes in the data distribution;
- **periodically update**. The iterative update that adjusts the action-values only periodically updates, thereby reducing correlations with the target.

The algorithm of deep Q-learning is listed behind:

ALGORITHM:DEEP Q-LEARNING

initialise replay memory D to capacity N
initialise action-value function Q with random weights θ
initialise target action-value function $\hat{Q}$ with weights $\hat{\theta} = \theta$
for episode = 1,M do
  initialise sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t=1,T do
    with probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
    execute action a_t and observe reward r_t and image x_{t+1}
    set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    store transition $< \phi_t, a_t, r_t, \phi_{t+1} >$ in D
    sample random minibatch of transitions $< \phi_j, a_j, r_j, \phi_{j+1} >$ from D
    if episode terminates at j+1: $y_j = r_j$
    otherwise $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \hat{\theta})$
    perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \theta))^2$
    Every C steps reset $\hat{Q} = Q$
  end for
end for

2.2.7 Deep Reinforcement Learning from Demonstration

In many real-world problems, we may not have an accurate simulator for the system but have data of the system being operated by its previous controller. At this point, Deep Q-learning from Demonstration (DQfD) [22] make its agent to learn from these demonstration data before running in the real environment, using the pre-train to imitate the demonstrator with a value function that satisfies the Bellman equation and proved to be effective at the very beginning of training. The key ways in DQfD is listed below:

- **Demonstration data**: DQfD is given a set of demonstration data;

- **Pre-training**: DQfD initially trains it neural network solely on the demonstration data before starting reaching the real environment;
- **Supervised losses**: a large margin supervised loss is used to make the action values of the demonstrator above the other action;
- **L2 Regularization losses**: L2 regularization losses is added to prevent overfitting on demonstrations;
- **N-step TD losses**: The agent updates its network with target from a mix of 1-step and n-step Q-learning returns;
- **Demonstration priority bonus**: The demonstration data are given a bonus of ϵ_d to boost the frequency to be sampled.

The Pseudo-code of DQfD is listed:

ALGORITHM:DEEP Q-LEARNING FROM DEMONSTRATION

initialise demonstration data set D^{replay}
initialise weights for initial behavior network θ
initialise weights for target network θ'
loop **for** step = 1,k do
  sample a mini-batch of n transitions from D^{replay}
  calculate loss J(Q) using target network
  perform a gradient descent step to update θ
  if t mod τ = 0: $\theta' := \theta$
  end for
loop **for** step = 1,2,... do
  sample action from behavior policy a $\pi^{\epsilon Q_\theta}$
  play action a observe(s',r)
  store (s,a,r,s') into D^{replay}
  overwriting oldest self-generated transition if over capacity
  sample a mini-batch of n transitions from D^{replay}
  calculate loss J(Q) using target network
  perform a gradient descent step to update θ
  if t mod τ = 0: $\theta' := \theta$
  s:=s'
  end for

In this project we are going to implement DQfD as the key learning algorithm for training the agent to grasp object. Therefore, to realise this algorithm, we not only need to implement the original DQN algorithm, but also to generate some demonstrations from simulation environments by ourselves to pre-train our neural network, before sending the agent to use DQN to interact with the environment. We are going to use simulation tools to sample from the grasping space to find some good grasping points, then use these points to create demonstrations.

2.3 Related Work

There are a few more fields related to this project, containing algorithms and simulation environments. In this section we are discussing these related techniques.

2.3.1 Object Manipulation

Reinforcement learning has been widely used in robotics containing missions about bipedal [66] [43] [20] and quadrupedal [35] locomotion, all kinds of sports [34], robotic visualisation and especially object manipulation [57]. It is regarded as the oldest task in this field and succeed in variety of area from robotic system to play with children [16] to water serving assistant helping the elderly [54]. The task of object manipulation is generally divided into two main area: 1) arm planing, refers to determining the best path that results in a collision free motion of the arm; 2) grasp synthesis, refers to the problem of finding a grasp configuration that satisfies a set of criteria relevant for the grasping task [9].

Recently, researcher has turned to focus on overall control pipeline to instead 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 [41]. It succeeded in learning 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. Further work tried to remove the need for instrumental training set-up but still being able to learn a variety of manipulation skills by using hand-eye coordination [18].

2.3.2 Robotic Environment Simulation

Using simulators instead of doing experiments with real-world robots and object could give convenience to researchers. Within simulation environments, it is able to model continuous state and train the agents with tiny time-steps that is unprocurable for a real-world experimental environment. However, differences between the simulation and real-world accumulate over time. Within a short period of time, a small error in the model can lead to the simulated robot diverging rapidly from the real system. This can be mitigated by very accurate simulation or using short horizon [28].

2.3.3 Collision Detection and Contact Determination System

Judgement of the contact between the robotic gripper and object could be the first challenge in grasping learning. GraspIt! is an environment for grasp analysis and planning, which can serve as a test bed for grasp evaluation, grasp synthesis as well as manipulation planning algorithms [46]. It can import a wide range of different kinds of robots, and generate the environment with the objects or obstacles robots

can interacts with. With its help, we could not only generate grasp data much more quickly than in the lab using an actual robot, but also then use the data on another physical/simulation system. Within GraspIt!, collision detection and contact determination system could be used to detect and mark the collision between robots and objects as well as prevent passing through. The grasp analysis system with the ACIS geometric modeling engine can reports contact types and locations with high precision for any configuration of the hand, which will be use to accurately compute the quality measures [48].

2.3.4 Grasp Planning and Evaluation

To generate, execute, record and compare a certain number of grasps, we need an effective grasp planning as well as evaluation. Two categories of methods aim to this topic: analytic methods, which consider performance according to physical models, and data-driven methods, which typically use human labels or the ability to lift the object in physical trials [42]. GraspIt! consists two parts to help: the first to create a set of starting grasp locations, and the second to test the feasibility and evaluate the quality of these grasps. It will firstly place the hand at the starting position, and form the fingers in the pre-grasp-shape. Next, the hand will be guided moved along the grasp approach direction from a grasp starting position toward the object. If any of the fingers is not blocked, it will close around the object until contact or joint limits prevent further motion. If at least one finger is in contact with the object, it will evaluate the grasp, otherwise it will try another grasp [48].

A good feature of GraspIt! is that its evaluation results in a scalar value, which will be very convenient used in our reinforcement learning as the reward. It uses a quality metric that determines the magnitude of the largest worst-case disturbance wrench that can be resisted by a grasp of unit strength. If none of the robot fingers touches the object, the quality will be -1. And it will give 0 to a grasp if it does not have force-closure (F-C), meaning there exists some set of disturbance wrenches that cannot be resisted by the grasp. Otherwise, the quality of the grasp will be a number between 0 to 1. Also, it is noteworthy that, this quality can be influenced by the material of both the robot and the object.

But the shortage of this grasp planning in GraspIt! is obvious. It is using simulated annealing as the searching algorithm, and evaluating a grasp even only one finger touches the object, which could highly be a local minimum instead of a global one, which has also been proved in our experiments. Furthermore, the outcomes will be only a table showing the grasps separately with highest quality and the positions as well as orientations of these grasps. Since there is no visualized outcome, we need to generate the grasping position manually in other environment. Aiming these problems, we use Dex-Net 2.0 as some kind of supplement and comparison of GraspIt!.

Dex-Net 2.0 uses the Grasp Quality Convolutional Neural Network (GQ-CNN) ar-

chitecture, which has approximately 18 million parameters. It takes depth images of the objects as input and estimate grasp robustness by passing the image and gripper depth through the network. It contains four convolutional layers followed by 3 fully connected layers and a separate input layer for the z, the distance of the gripper from the camera. The network estimates the probability of grasp success (robustness), which can be used to rank grasp candidates (see Figure 2.6) .

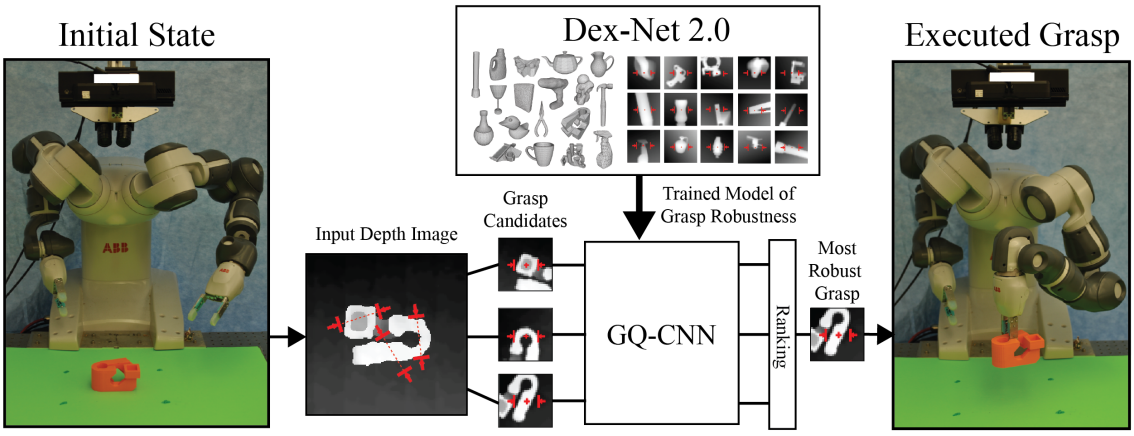

Figure 2.6: structure of Dex-Net 2.0 [42]

Under this architecture, the Baxter will use the depth-RGB picture as the put, then predict grasp robustness from a point cloud, send the test positions into the GQ-CNN and output the qualities of the grasps. It will finally choose the most robust grasp to execute. It is claimed that the Dex-Net 2.0 grasp planner is as reliable and 3 faster a method based on point cloud registration, and had 99% precision on a test set of 40 novel objects [42].

But within Dex-Net 2.0, all the experiments are carried out with parallel-jaw grasps, with a griper named Baxter. The algorithm is not considering the specific action for every finger but just close its jaw as the grasp movement. Therefore, these system is not yet suitable for other gripers, especially gripers with three or more fingers. However, this is not a big problem for our project, as we planned to experiments with two-finger gripers or pinch grasp with five-finger MPL.

Also, the grasping prediction is coming from the previously generated cloud points. Thus, some very good points may be able to successfully grasp the object with high grasping quality, but still not the best position, may somehow cause the failure in more complicated real-world task.

2.4 Summarisation

In this chapter we introduced the background of our project. We firstly introduced the family of reinforcement learning algorithms. Then we introduced different kinds of artificial neural networks and finally with the knowledge of deep learning and

reinforcement learning, we introduced deep reinforcement learning and especially the core algorithm, deep reinforcement learning form demonstration. Finally, we introduced some other techniques using in this project containing the robotic manipulation, robotic environment simulation, collision detection, and grasp planning and evaluation.

Chapter 3

Research Choice

Three important decisions have been made at the early stage of this project: the simulation tools we needed to sample the grasp database as well as to carry out our training experiment; a language not only to implement our reinforcement learning algorithms but also with support from a machine learning library to construct and train the deep neural networks; the structure to combine these two part together.

3.1 Simulation Environment

At the very beginning of our project, we noticed that there are a few third-party open source simulation tools and some of them are well-developed and have been widely used. With investigation and study, we found some simulation tools satisfy our requirement for this project and thus decided to focus on the project itself and not to develop one simulation tool by ourselves.

3.1.1 Sampling Simulations

In this project we are training a robot hand to grasp objects. As we are referring the DQfD learning algorithm, we need some demonstrations to pre-train our network. Therefore, the first part of our project is to get these demonstrations. We plan to sample a huge mount of grasps and choose the best ones to generate our demonstration data, and the simulation environments we chose are GraspIt! [47] and Dex-Net 2.0.

GraspIt! is an interactive grasping simulator. It can import a wide variety of hand and object models and then evaluate the grasps formed by these hands with objects. The core features we need from this simulator include:

- **a robot and object library**: includes several hand models and objects this project needs, such as the Barrett Hand and humanoid hand and several objects;
- **a fast collision detection and contact determination system**: which could generate grasps quickly;

- **grasp analysis routines**: that evaluate the quality of grasps;
- **visualization methods**: can show the weak point of a grasp and create projections of the grasp wrench space;

GraspIt! is powerful with the whole process automatically carried out, planning, sampling and evaluating, which is what we need for this project. But still, GraspIt! has some shortages. The most obvious one is that, it is using simulated annealing as the searching algorithm for the grasps and it could fall into local minimums and give them as its output instead of global minimums. Especially, we found that, when the hand or the object model is complicated, the state space then being large and high-dimensional, its planning for searching algorithm could be aimless with higher chance to fall in local minimums.

Dexterity Network 2.0(Dex-Net 2.0) [42] is a synthetic dataset. Trying to make the planning more effective, researchers combined convolutional neural network model with grasping evaluating algorithm to generate a Grasp Quality Convolutional Neural Network (GQ-CNN). After trained by Dex-Net 2.0, GQ-CNN can be used to plan grasps in short time (0.8s) with a high success rate (93%) on objects with adversarial geometry and even higher success rate on household objects. The most important feature for Dex-Net 2.0 are:

- **a dataset**: the Dex-Net 2.0 is a dataset associating 6.7 million point clouds and analytic grasp quality metrics with parallel-jaw grasps planned using robust quasi-static GWS analysis on a dataset of 1,500 3D object models;
- **GQ-CNN model**: trained to classify robust grasps in depth images;
- **grasp planning method**: samples antipodal grasp candidates and ranks them with GQ-CNN.

Dex-Net 2.0 proved to be precise for its output and its grasp planning method and variety of dataset are what we take advantage of in this project. However, all Dex-Net 2.0's success is built on the setting that the griper to carry out the grasp is a parallel-jaw. So we use the outcome of Dex-Net 2.0 as an alternate for GraspIt! in the second half of our experiment.

3.1.2 Training Simulation

We chose MuJoCo [68] as the simulation environment. MuJoCo is a model-based physics engine standing for Multi-Joint dynamics with Contact, providing a fast and accurate simulation. It scales up some computationally-intensive techniques, such as optimal control, physically-consistent state estimation, system identification and automated mechanism design and apply them to complex dynamical systems in contact-rich behaviors. While it still has the traditional applications such as testing and validation of control schemes before deployment on physical robots, interactive scientific visualization and virtual environments .

MuJoCo has many unique features attracting. The most attracting one to this project is that, there are multiple ways to access MuJoCo's functionality. It could not only use a static C library linked to programs, but also a standalone executable interactive interface, visualized in OpenGL. While all the models are created in intuitive XML format with built-in model compiler and runtime simulation module is written in C tuned to good performance.

The library we using is called MuJoCo Pro. It is a dynamic library with C API, includes the XML parser, model compiler, simulator and interactive OpenGL visualizer. It further exposes a large number of functions for computing physics-related quantities, not necessarily in a simulation loop. MuJoCo Pro could handle a lot of model-based computations such as control synthesis, state estimation, data analysis, etc. With all MuJoCo's conveniences, we could use MuJoCo Pro to verify the model we trained with grasp data and visualize the whole procedure.

3.1.3 Other simulation tools

There are many other robot simulation tools available, but for some or other reasons, we did not choose to use. We will discuss them in this part. The first one is Microsoft Robotics Developer Studio [31]. It is a free 3D simulated environment for robot controlling research. As a Microsoft tool, its programming is coding with C#, which is not as popular as Python or R in Machine learning, and its support was suspended in September 2014. V-REP[58] has a large range of feature for its environment, which supports a lot of different programming languages such as C/C++, Python, Java, Lua, Matlab, Octave or Urbi. It is also allowed to change the physics engines during simulation. But V-REP is still developing and being reported with bugs, especially when it compiles its files. Next we investigated RoboLogix [36] a simulator allows users to write their own movement sequences, modify the environment and use sensors on five-axis industrial robot. However, it is also the only robot that could be used with very simple functions. Webots [45] is one of the most popular robot simulations used by thousands of companies, universities and research centers. It could be programmed and simulated with C/C++, Java, Python, Matlab or URBI. But its robots and objects are not for free and need to be purchased separately.

3.1.4 Different Grippers

In this project, we discuss four different kinds of grippers: the two-fingered Baxter [19], the three-fingered Barrett Hand [69], a five-fingered humanoid hand and Modular Prosthetic Limb (MPL) [30]. And due to the definition of these models, we are using Baxter in both MuJoCo and Dex-Net 2.0, the Barrett Hand in both GraspIt! and MuJoCo, humanoid hand in GraspIt! and MPL in MuJoCo.

Baxter (see Figure 3.1(a))is collaborative industrial robot with ability to work in less structured environments and adapt to changes in its surrounding [19]. It has a video camera in each end effector, where they can best be positioned and used

to locate objects. Each end effector also includes an infrared range-finder used to locate the edges of objects and determine height for picking from stacks of objects. In this project, we will only use Baxters hand as a two-finger griper.

The Barrett Hand (see Figure 3.1(b)), is ubiquitous in robotics research due to its durability and relatively low cost. It is produced by Barrett Technology with each of its three fingers having two joints. One finger is stationary while the other two can spread synchronously up to 180 degrees about the palm. It has four internal degrees of freedom: one for the spread angle of the fingers, and three for the angles of the proximal links [69]. In this project, we use the Barrett Hand as a three-finger griper.

One of the hardest problems in robotic grasping is the creation of control algorithms for new hand designs that are beginning to rival the human hand in complexity. However, if we wish to reproduce human-like grasping it would seems natural to draw inspiration not only from the hardware of the human hand, but also from the software [13]. Thus, we are using a five-fingered humanoid hand defined by GraspIt!. While in our training experiment, we are using the MPL model (see Figure 3.1(c)).

Figure 3.1: (a)Two-finger Baxter; (b) three-finger Barrett; (c) five-finger griper MPL

3.2 Machine Learning libraries

After choosing our simulation environments, we need to make a decision about the machine learning library we use, taking the programming language into consideration. As we are going to execute a deep reinforcement learning algorithm, it is important to be able to effectively build up deep neural networks. We compare some of the popular options: TensorFlow, Theano, Caffe and MXNet.

3.2.1 TensorFlow

TensorFlow is an open source software library for expressing machine learning algorithms using data flow graphs. Its mean features are:

- **transferability**: A computation expressed using TensorFlow [1] can be executed with little or no change on a wide variety of heterogeneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards.
- **flexibility**: The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learning systems into production across more than a dozen areas of computer science and other fields, including speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery [1];

TensorFlow can run on multiple CPUs and GPUs and was originally developed for working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research [15], but it is now open sourced and be applicable in a wide variety of other domains.

3.2.2 Theano

Theano [4] is a Python library that allows it users to define, optimise and evaluate mathematical expressions involving multi-dimensional arrays. The most important feature for Theano are:

- **transparent use of a GPU**: it could use GPU and thereby computes much faster than on a CPU;
- **efficient symbolic differentiation**: it does derivatives for functions with one or many inputs;
- **dynamic C code generation**: it could evaluate expressions fast.

Also, Theano handles large-scale computationally intensive scientific investigations well, but also approachable to be used with personal computer.

3.2.3 Caffe

Caffe [29] is a deep learning framework developed by Berkeley AI Research.The key features for Caffe are:

- **expressive architecture**: its models and optimisation are defined by configuration without hard-coding. It could be switched between CPU and GPU by setting a single flag and then deploy to mobile devices;

- **extensible code**: it has been forked by over 1000 developers in its first year and then tracks good code and models;
- **fast speed**: it is announced to be the fastest convnet implementations available.

3.2.4 MXNet

MXNet [12] offer powerful tools to help developers exploit the full capabilities of GPUs and cloud computing. MXNet places a special emphasis on speeding up the development and deployment of large-scale deep neural networks. In particular:

- **device placement**: its easy to specify where each data structures should live;
- **multi-GPU training**: MXNet makes it easy to scale computation with number of available GPUs.
- **automatic differentiation**: MXNet automates the derivative calculations that once bogged down neural network research.
- **Optimized Predefined Layers**: in MXNet, the predefined layers are optimized for speed, outperforming competing libraries.

3.2.5 Comparison for Libraries

Obviously, we are using only one machine learning library for the whole project. To choose the library we take the following features into consideration: the language it uses, the hardware support, the speed for calculation, the flexibility and the superiority models (see Table 3.1). TensorFlow has many advantages we need in this

Table 3.1: Comparison for Libraries

Library	TensorFlow	Theano	Caffe	MXNet
Language	Python	Python	Python/Matlab	Python/R
Hardware	CPU/GPU/mobile	CPU/GPU	CPU/GPU	CPU/GPU/mobile
Speed	Fast	Medium	Fast	Fast
Flexibility	Good	Good	Normal	Good
Model	CNN/RNN	CNN/RNN	CNN	CNN

project. Therefore finally we chose TensorFlow as the machine learning library we use for this project, and chose Python as the language we program.

3.3 Communicating Structure

In this project, we are using deep reinforcement learning algorithm. After choosing MuJoCo as the training simulation tool, Python as the language and TensorFlow

as the machine learning library, we need a structure to bind them together. The structure should be able to send message between two projects, one is calculating the deep reinforcement learning, and the other uses its output to do the experiment and sends back the results. This communication process itself should be fast, stabilized, low cost and locked. Therefore, it is very natural to think of Hypertext Transfer Protocol.

3.3.1 Request-Reply Structure using HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, and hypermedia information systems [17]. HTTP is used to exchange or transfer hypertext, and here in our project we use it to transform information between our deep reinforcement learning algorithm and MuJoCo. As there are two communicators in this structure and they are asking and answering interchangeably, we choose Request-Reply structure (see Figure 3.2) In this structure, the deep rein-

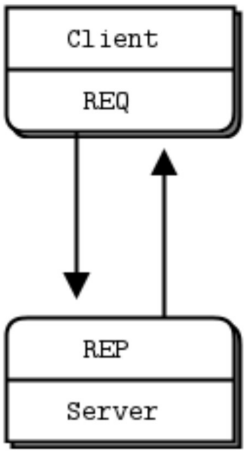

Figure 3.2: Request-Reply structure

forcement learning algorithm written in Python shall be the client and keep sending action request. While on the other hand, the MuJoCo is acting as the server, after receiving the request, it will carry out the movement and send back the observation as reply.

3.3.2 Networking library

The networking library we choose in this program is ZeroMQ [23]. ZeroMQ is not a standard message broker but a lightweight messaging library which provides messaging capabilities. Distributed applications can use ZeroMQ for high-throughput and low-latency communication, profiting from its ability to implement direct connection among producer and consumer, with no intermediate entities involved.

The idea behind ZeroMQ is powerful, it allows for high-performance and low-latency communcation, but comes with additional complexity at the application level.

3.4 Building up Environments

After choosing all the libraries and simulation tools we need, we build them up one by one and use the networking library to bind them together.

3.4.1 TensorFlow Installation

The TensorFlow is the core library we use in this project for realise deep reinforcement learning algorithm. Thus, we firstly need to build the environment for it. It is a little bit different from installing TensorFlow on different system operations. We tried Mac OS X and Ubuntu Linux and due to the performance, stability, as well as referring to the operation requirement from the other simulation tools we are using, we chose to build on Ubuntu 16.04. And the rest of this work will only discuss the situation under Ubuntu 16.04. **TensorFlow CPU Version Installation**:There are two main versions of TensorFlow: 1) the CPU version, with CPU support only; and 2) the GPU version, giving GPU support. To determine using which version of TensorFlow is depending on the project calculation demanding as well as the hardware support. However, even with GPU hardware, it is officially suggested to install the CPU version first. Thus we follow this suggestion.

There are five different mechanisms to install the CPU version: 1) virtualenv; 2) native pip; 3) Docker; 4) Anaconda; 5) installing from sources. We tried the first four options, and they are all very easy. We take the native pip way as an example.

As the pip or pip3 package manager is usually installed on Ubuntu, we firstly need to confirm the version of pip or pip3 (by issuing a pip -V or pip3 -V command) that is installed, and it is recommended to have a version 8.1 or higher pip or pip3. If not, we should issue the following command and then install TensorFlow:

```
$ sudo apt-get install python-pip python-dev # for Python 2.7
$ pip install tensorflow # Python 2.7; CPU support
```

If it went well we shall then be able to import TensorFlow library into a Python program making a TensorFlow program. We could use the official Mnist example to test our installation. After running the program mnist_softmax.py we could have the following output proving our TensorFlow with CPU support being properly functioning.

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-images-idx1-ubyte.gz
0.9206
```

TensorFlow GPU Version Installation:After successfully installed the CPU version, we could move on to the GPU version as it will be a lot faster due to the help of GPU. Before installing it, we need support from NVIDIA CUDA and cuDNN. They could be found on their official website. We choose to install CUDA 8.0 with cuDNN v5.1, and set the environment variables for them to enable GPU support. At the beginning of installing TensorFlow, we are again facing a choice: to use which method to install. We tried them all and the way straightly installing from sources is the most complicated but stable one for our environment.

To install from sources, we need to first of all install Bazel and some supporting libraries for it, such as swig, python-dev and python-wheel. Then we could download TensorFlow and configure its installation in terminal.

Although we are installing from sources, we still need to create a pip package and install it, which could be done with help of Bazel. After that, TensorFlow with GPU support could be successfully installed. And we could use the Mnist program to test again and got the following output.

```
I tensorflow/core/common_runtime/gpu/gpu_device.cc:907] Found
device 0 with properties:
name: GeForce GT 750M
major 3 minor: 0 memoryClockRate (GHz) 0.9255 pciBusID 0000:01
Total memory 1.95GiB
Free memory 1.47GiB
I tensorflow/core/common_runtime/gpu/gpu_device.cc:928] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_device.cc:938] 0: Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:997] Create
TensorFlow device(/gpu:0)->(device:0, name:GeForce GT 750M,
pci bus id: 0000:01:00.0)
0.9203
```

And this time, some information about GPU's functionality will be output. Also, it could be clearly felt that the program is running much faster, while the computing results of the testing program are the same.

3.4.2 GraspIt! Installation

GraspIt! is the simulation environment we need to sample the grasps for the first half of our project. To install it, we need first to install some libraries (such as Qt, Coin, SoQt, Lapack, etc.) They could be simply installed using apt-get in terminal with simple commands. After that, we could then install GraspIt! itself. GraspIt! can be integrated into ROS workspace on Ubuntu via its package, graspit-ros, which pulls in GraspIt! as a git-submodule. With its help, we could then setup a ROS interface for GraspIt!. After all these done, we could visualize the whole workspace and the grasp procedure through the GraspIt! interface.

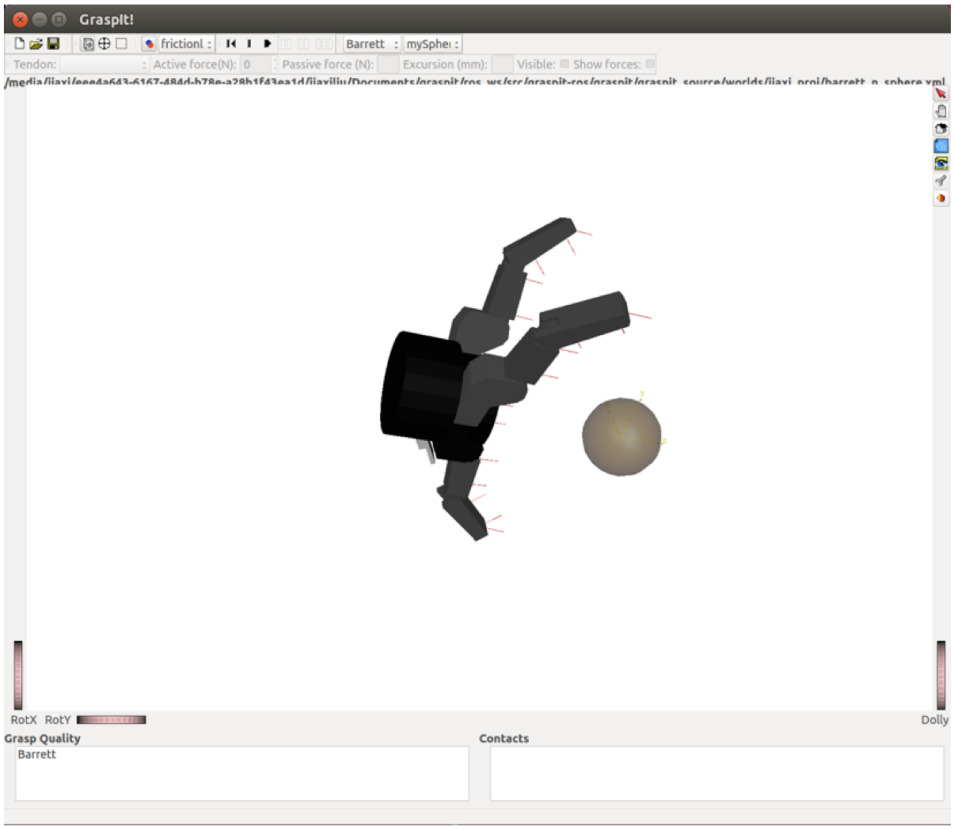

Figure 3.3: Interface of GraspIt! with the Barrett hand and a sphere

Within GraspIt! interface, we could import the robot hand we choose into the workspace. As a test flight, we firstly choose the Barrett Hand with a sphere, which could be later on changed to other hands with a more complicate object (see Figure 4.1).

3.4.3 Dex-Net 2.0 Installation

There are two different version of Dex-Net 2.0: the Python-only installation and ROS installation. The former is intended for users who are only interested in training GQ-CNNs for grasp planning. While installation as a ROS package is intended for users who wish to use GQ-CNNs to plan grasps on a physical robot. As we are not going to training a physical robot at this stage, we choose the Python-only installation. Also, there is an option about whether use TensorFlow GPU support. As we have installed TensorFlow GPU version, we could choose this option. Then Python-only and GPU supported installation could be installed by the following command:

```
$ git clone https://github.com/BerkeleyAutomation/dex-net.git
$ sudo sh install.sh gpu python
```

The first lone clones the project from Github. And the second line runs the dex-net installation helper script. After installation we could the following command to test it, which is also recommend before using the module:

$ python setup.py test

If all the tests passed, we could then successfully enter its command line interface by command:

$ python apps/dexnet_cli.py

then we should be able to start to use Dex-Net 2.0 with its Command line interface (see Figure 4.3).

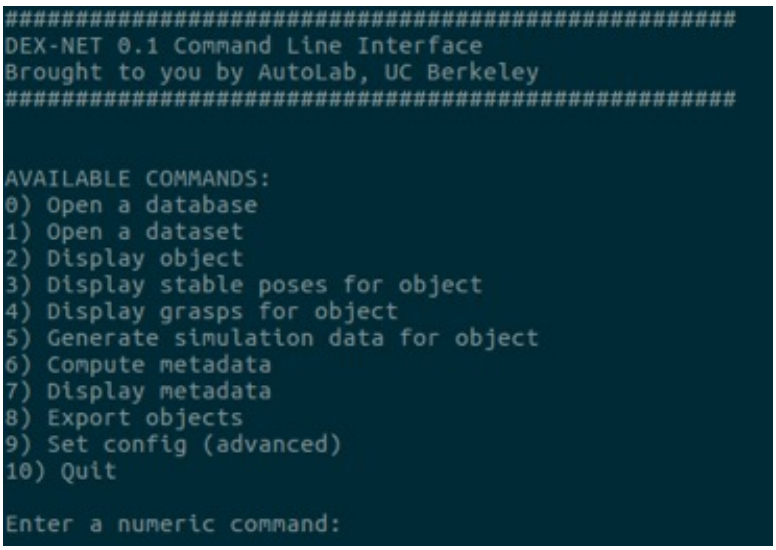

Figure 3.4: Command line interface of Dex-Net 2.0

3.4.4 MuJoco Installation

MuJoCo Pro is a commercial product. In order to use it, We need to follow its requirement to apply for a license. After this application, we will be sent an activation key in text file and it should be placed in a right path. And then we could down the MuJoCo Pro library, which contains an XML parser, compiler, simulator, abstract visualizer and OpenGL renderer. After setting environment variables for it, we could use terminal to run the example it provides to have a look at its interface (see Figure 4.2(a)).

The example is working with a humanoid model, which is not what this project focus on. We could either write our own model in XML files or download one on-line. Our first robot model, the Barrett Hand, is widely used for teaching and researching. Thus, it is not hard to find a usable one with mesh files and structure developed. The one we have found may not be aimed to use in MuJoCo but other ROS workspaces, thus it requires some adjustments to follow the limits of MuJoCo to its models. After the adjustments, the Barrett hand could then be used within MuJoCo Pro as shown in Figure 4.2(b). We could then firstly do some simple interaction with this hand model by mouse dragging. Its joints could be then rotated. However, other more

complicate controlling should be down by modifying simulation source files, cooperating with more adjusting about the hand model file itself, which is the way we execute our on-line learning, and will be discuss in the following chapter.

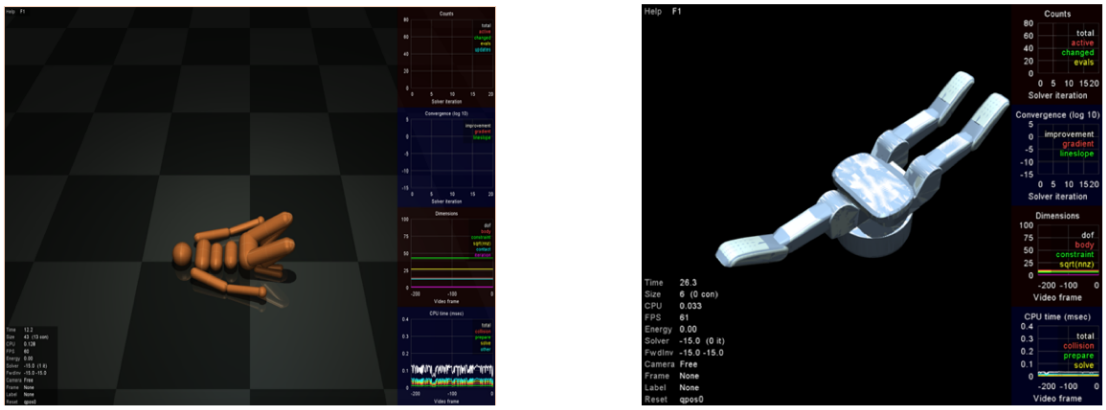

Figure 3.5: (a) Interface of MuJoCo Pro; (b) Barrett Hand in MuJoCo

3.4.5 ZeroMQ Installation

We have chosen ZeroMQ as our Networking library then in this section we introduce the installation about it. As we are sending message between our deep reinforcement agent, a Python program and simulation tool MuJoCo, a C++ program, we need some two kinds of libraries for them. **ZeroMQ package for MuJoCo (C/C++)**:First of all we install some necessary packages, such as libtool, pkg-config, build-essential, etc. And then we install libsodium package to help us later on installing ZeroMQ. And it could be download form its website and easily installed after configured. And then we download, configure and install ZeroMQ package. After that, we could then import zmq.hpp into our C++ project. **ZeroMQ package for Python**:After installed the standard ZeroMQ Package, we could then install its python package in our Ubuntu. And there are several ways to do it such using easy_install, pip, source code, etc. The one we choose in this project is to use pip:

```
$ sudo pip install pyzmq
$ python setup.py build_ext –inplace
$ python setup.py test
```

If it passed the test, we could then use ZeroMQ package in our Python program by import zmq.

3.5 Summarisation

In this chapter, we introduced the trade-off between the machine libraries and simulation tools. We compared different machine library first and reason for choosing

TensorFlow, which to be implementing in Python. Then, we introduced some simulation environments, specifically we are using GrapIt! and Dex-Net 2.0 as the sampling simulation tools, and we chose MuJoCo as the training simulation environment. To communicate between TensorFlow and MuJoCo, we need a networking library as the final piece of puzzle, which will be performed by ZeroMQ. ZeroMQ will send message between deep reinforcement learning agent and simulation environment.

Chapter 4

Preliminary Experimentation

In the first half of the experimentation, we did some basic experiments about implementing reinforcement learning, deep reinforcement learning. We sampled some grasps using GraspIt! with its two main planning method: grasping planning and eigen-grasping planning. And we did some test-flight in MuJoCo with the same model we use in GraspIt!.

4.1 Reinforcement Learning Experiments

The core part of our project is using reinforcement learning algorithm to learn a policy for robotic hand agent. Before we use the robot hand to grasp directly in a simulation environment, we simplified the problem we facing and did some pre-research and implementation about reinforcement learning algorithm.

4.1.1 Model Simplification

In our project, the agent, a robot hand, shall be moving in a 3D space, the environment. At each state, it could choose among several actions to take, such as to move, to rotate or to close its fingers. After taking an action, it should then observe the environment, and get the information about the new state it in and get the reward about its last action. One episode shall end if the robot finished its whole grasping procedure, contains the movement to the position it desired, the rotation it needs, the closing finger movements and maybe at last raising the object to a certain height. And specifically, if the object is successfully grasped to a certain height, the agent should be given a positive reward in the end. On the contrary, if it finished all its action without successfully grasping the object it should be given a negative reward. It is worth mentioning that, most of the actions during the grasping procedure should be given a small negative reward, in order to prevent the agent moving aimlessly.

With the situation described above we try to simplify it into a model. And we found it perfect to represent it as a ”gridworld” puzzle. A simple version of gridworld could

be represented as a 3×4 2D state space shown as Figure 4.1 and let's call it gridworld 1.0.

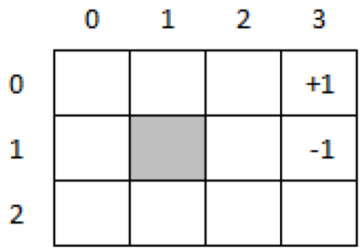

Figure 4.1: 3×4 2D state space gridworld 1.0

There are 12 states, and each state will give a certain reward to the agent: two terminal state point (0,3) and point (1,3) will give 1 or -1 respectively, and the rest will give -0.04. Within every state, the agent could choose an action to make from four actions, to go up, down, left or right. If it hit the wall or the obstacle (the gray grid), it will remain in the previous state, otherwise move into the new state and get the reward. At the beginning of a episode, the agent will start from the left down corner, point (2,0). We will use the reinforcement learning to help the agent learn to gain the most total reward: to quickly move to the point (0,3). Also, to realise the ϵ-greedy idea, the agent will have a small probability to move randomly.

We shall start with the simple version of Q-learning, using Q table. We firstly initial a table called Q table to store the Q values for very action the agent could choose in every state. This gridworld has 3×4 states, and each state has 4 actions: going up, down, left and right. Thus we need to build a $3 \times 4 \times 4$ table for all of the Q values. Each time the agent in a specific state, it will look for the table, and choose the action with the highest value to take its next step. After each step, the agent will receive a reward, and it will be use to update the Q value in the table. The agent will keep looking for the table and take action until it get into the terminal state, which is the end of an episode and it will start again from the start state. With this procedure, we could gradually update the table and get all the state-values to help the agent move from start point to the terminal point.

The state-values we got after 500 episodes are shown in Table 4.1:

Table 4.1: State-values for gridworld after 500 episodes

0.81	0.89	0.95	+1
0.69	obstacle	0.53	-1
start point	0.61	0.56	0.18

We could see from the table that, the agent has generated a very clear route that to move from the down left corner to the up right corner by going up first and then going right. Then we could make one step further using Q-network and this is where TensorFlow can show its capabilities. To beginning with, we build a two-layer

network, one for input and one for output. We are defining a very simple model here that using 12 nodes to describe the current state as the input, and 4 nodes for the four actions to take as the output. To complete the network we define 12x4 weights w and 4 bias b. When training the network, the agent will firstly observe its state using the network to have four action-value, and the highest action-value for a state will be its state-value. The agent will choose an action with highest value and get into the new state. And then use the network again to get the state-value for the new state and use Bellman equation to calculate the value Q' to update the previous state's action value. The loss function is define with square error between Q and Q'. And we use gradient descent optimizer to minimize the loss. We print the total steps the agent needs to take from the beginning to the terminal state in Figure 4.2:

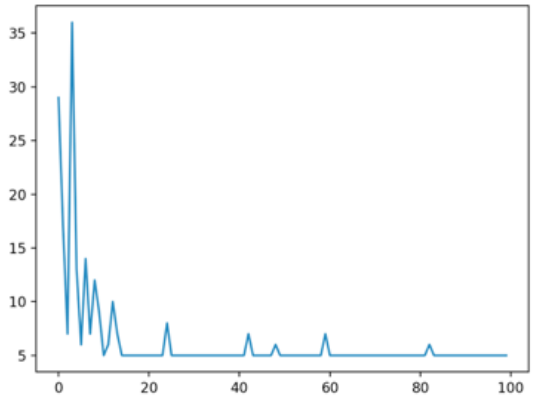

Figure 4.2: the steps agent needs using 2 layer network

It is clear that it could quickly learn the fastest way that take 5 steps from beginning to the end. The reason for the turbulence is that we take the exploration and exploitation idea into account here by using ϵ-greedy strategy: there is a certain chance for the agent to take the action that is not with the highest action-value.

Then we add one more hidden layer with 20 nodes to the network. The result in Figure 4.3 shows that the agent will need far more steps in every episode to find the right way. But it will still converge after 50 episodes training.

4.1.2 Model with Modified Reward Function

In the last section, we introduce a way to simplify our model and use the reinforcement learning to solve it. We made a few assumption and one of them could be to improved in this section, that the reward function should be more precisely defined.

In the grasping mission, we may have some points in the state space satisfy the following feature: 1) they are not a terminal points, thus not finishing the grasping episode; 2) it is on the right trial or very close to a good grasping finish point; 3) going from start point to these points or from these points to terminal points could avoid obstacle or have shortcuts. We call these points **good middle-points**. In the

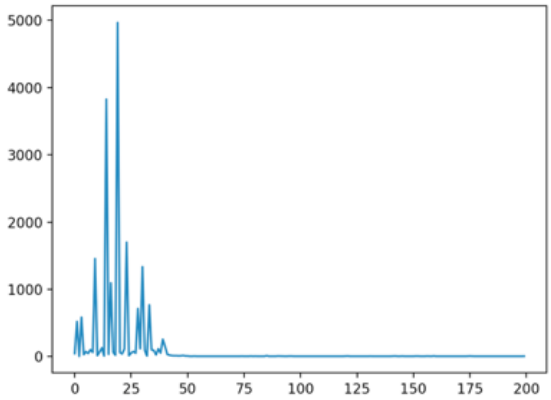

Figure 4.3: the steps agent needs ssing 3 layer network

previous settings, the agent will not be give any extra reward when reaching these points. However, with extra reward, we could encourage the agent to move to these points and then help it to finish a good grasping. we should We then change our state space in the following steps:

1. Change the whole states space from 3×4 to 10×10;
2. Add one extra-reward points at (5,5); if the agent move to this point it will receive 0.5 as reward;
3. Move the starting point from the down left corner to up left (0,0);

The negative terminal point and the positive terminal point are still at the down right corner at point (9,8) and (9,9) respectively. With these modification, we actually defined a more complicated reward function: gain 1 at (9,9), -1 at (8,9), 0.5 at (5,5), otherwise -0.04. Let's call it gridworld 2.0

And again, we use the two-layer neural network to solve this problem. But now, we need 100 nodes as input since we have a 10×10 state space, with 100×4 weights and 4 biases. And after training, we print out all the action-values in a table with green and red colour. Green represents positive value and red represents negative values. The brighter the colour, the higher of absolute value of the action-value. After 1000 episodes training, we got the action-values showing in Figure 4.4 and the steps need for episodes in Figure 4.5.

In Figure 4.4 We could see from the action-values that, the agent has generalised a policy that move to an extra-reward point first and then to the terminal point. But in the Figure 4.5 we could also found that, even trained with 1000 episodes, the agent could still sometimes take 100 steps or even more in a single episode, showing that the policy is not quite converged, since the optimal policy should only take 20 steps in this gridworld 2.0. The reason behind is that the reward function has been changed to more complicated. The agent could get extra reward when reaching the good-middel point we difined and thus it may turn back instead of heading to the terminal point after once got the reward.

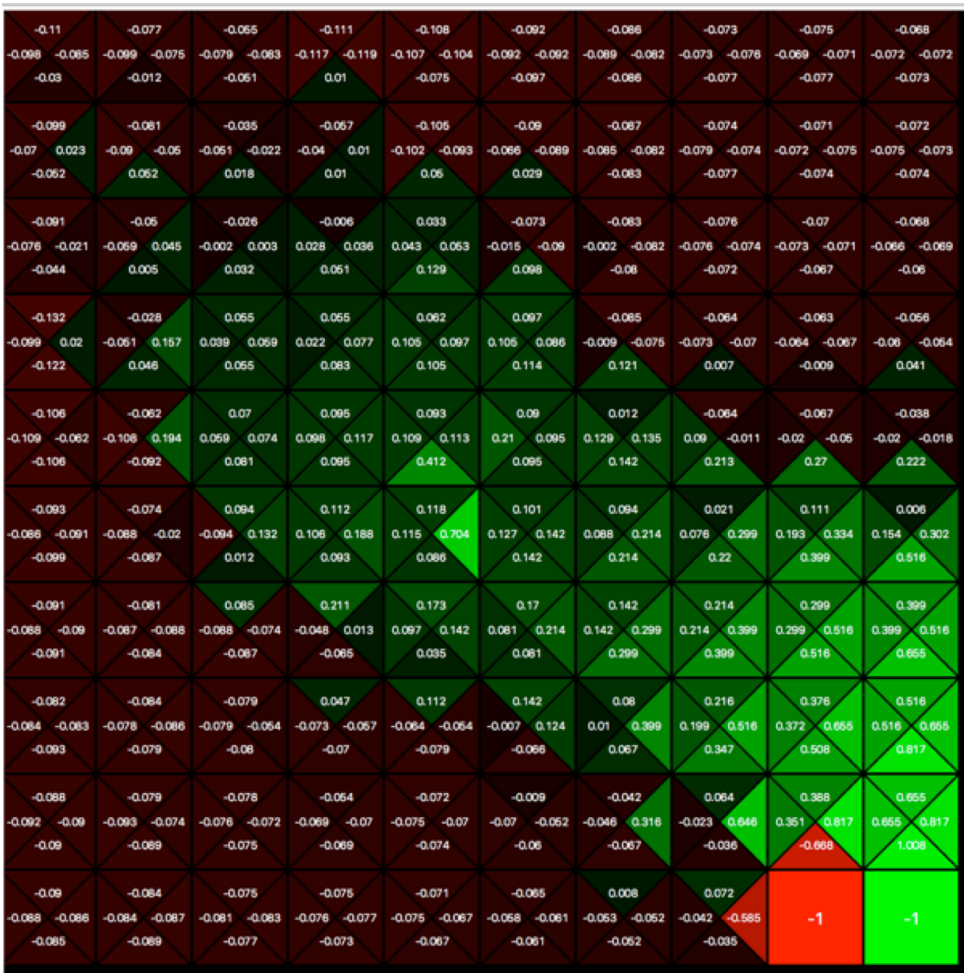

Figure 4.4: the action-values for gridworld 2.0

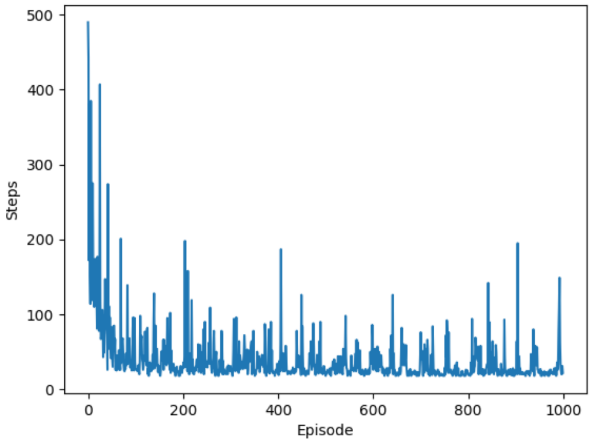

Figure 4.5: the steps needs for every episode in gridworld 2.0

We then think about the idea of $Q(\lambda)$-learning, then try to change to loss function in reinforcement learning. In the original loss function, after each step, the agent

will use tuple $< s, a, r, s' >$ to update its action-value. Then, each time, it is actually taking one step further into consideration. We could change the loss function to take two steps into consideration.

$$J_2(Q) := (R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \gamma^2 \max_{a_{t+2}} Q(s_{t+2}, a_{t+2}) - Q(s_t, a_t))^2 \tag{4.1}$$

In this function, $R(s_t, a_t)$ represents the reward the agent got at state s_t taking action a_t. And we could take one more step, using the loss function:

$$\begin{aligned} J_3(Q) := \quad & (R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \gamma R(s_{t+2}, a_{t+2}) \\ & + \gamma^3 \max_{a_{t+3}} Q(s_{t+3}, a_{t+3}) - Q(s_t, a_t))^2 \end{aligned} \tag{4.2}$$

It turned out to be effective in this problem. The steps the agent needs after changing the loss function are shown in Figure 4.6(a) and (b) respectively:

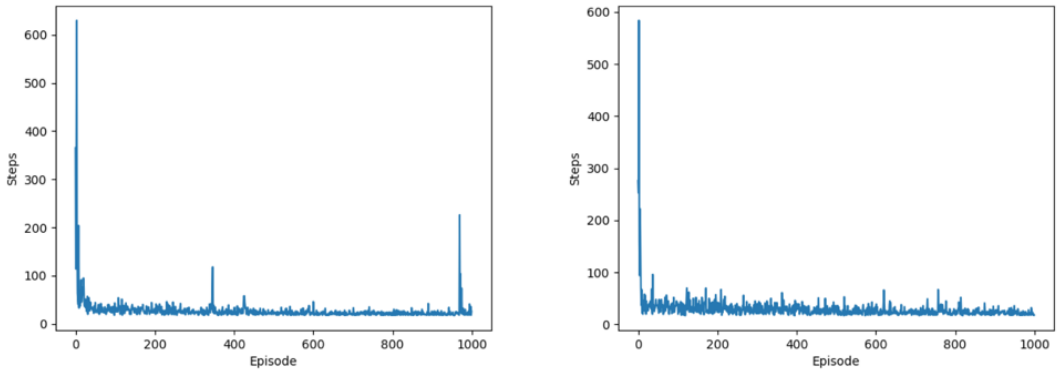

Figure 4.6: (a) action-values with two steps; (b) action-values with three steps

It is very clear to see from the figures that, after changing the loss function taking two steps into consideration, the result is a lot better. The average steps the agent needs for each episode reduced sharply and steps in every episode becomes more steady, which means the agent generated a police to heading to terminal state, instead of loop around. But still, there are some high peaks. When we take three steps into consideration using the equation 4.2 as loss function, it becomes even better and we eliminated these peaks.

4.1.3 DQfD Algorithm Demo

The main idea in DQfD is to use demonstration data to pre-train the agent. To realise this idea we need some demonstrations. We will still use the same state space, the gridworld 2.0, and use the following steps to create some demonstrations by ourselves.

1. Cancel the extra-reward at point (5,5);
2. Start the agent from point (5,5) at the beginning of each episode;

3. Train the agent with reinforcement learning for N episodes;
4. Record the agent's behavior with the tuple $< s, r, a, s' >$ after training for M episodes.

Since the tuples are recorded after the agent has been trained for a certain number of episodes, we could regard these experience then as some demonstrations. Then we use the DQfD algorithm discussed above. We use these demonstration to pre-train our agent T times before the agent interacting with the environment. We set the N=1000, M = 100, T = 100 and repeated our experiment, then got the results in Figure 4.7:

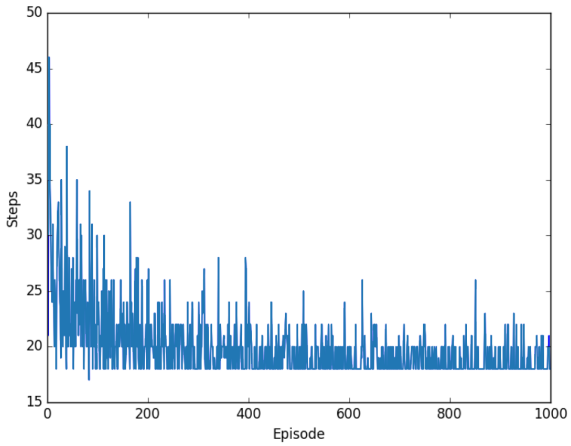

Figure 4.7: the steps agent needs with DQfD algorithm

We could see that, without pre-train, in the first few episodes, the agent will need around 600 steps to find the right way out. But with pre-trained by demonstration, it only needs around 45 steps at the very beginning.

4.2 Parameters Adjustment

In the previous part we have discussed the case that adding an extra-reward and presented our solution by taking more step into consideration. But we only add one extra-reward point, at the middle of state space, which might be natural for the agent to pass by. Also, we haven't discussed any parameters using in the algorithm. First of all, we change the state space to gridworld 3.0 as follow:

1. Add a second extra-reward point at (1,8) and (8,1), if the agent move to these two points it will receive 0.5 as reward;
2. Move the negative terminal point to (7,7).

After making these two changes, we made a symmetric state space with two points that the agent could have extra-reward. And these two points are not at the center of the state space, therefore we have a more obvious trace of the agent see Figure 4.8.

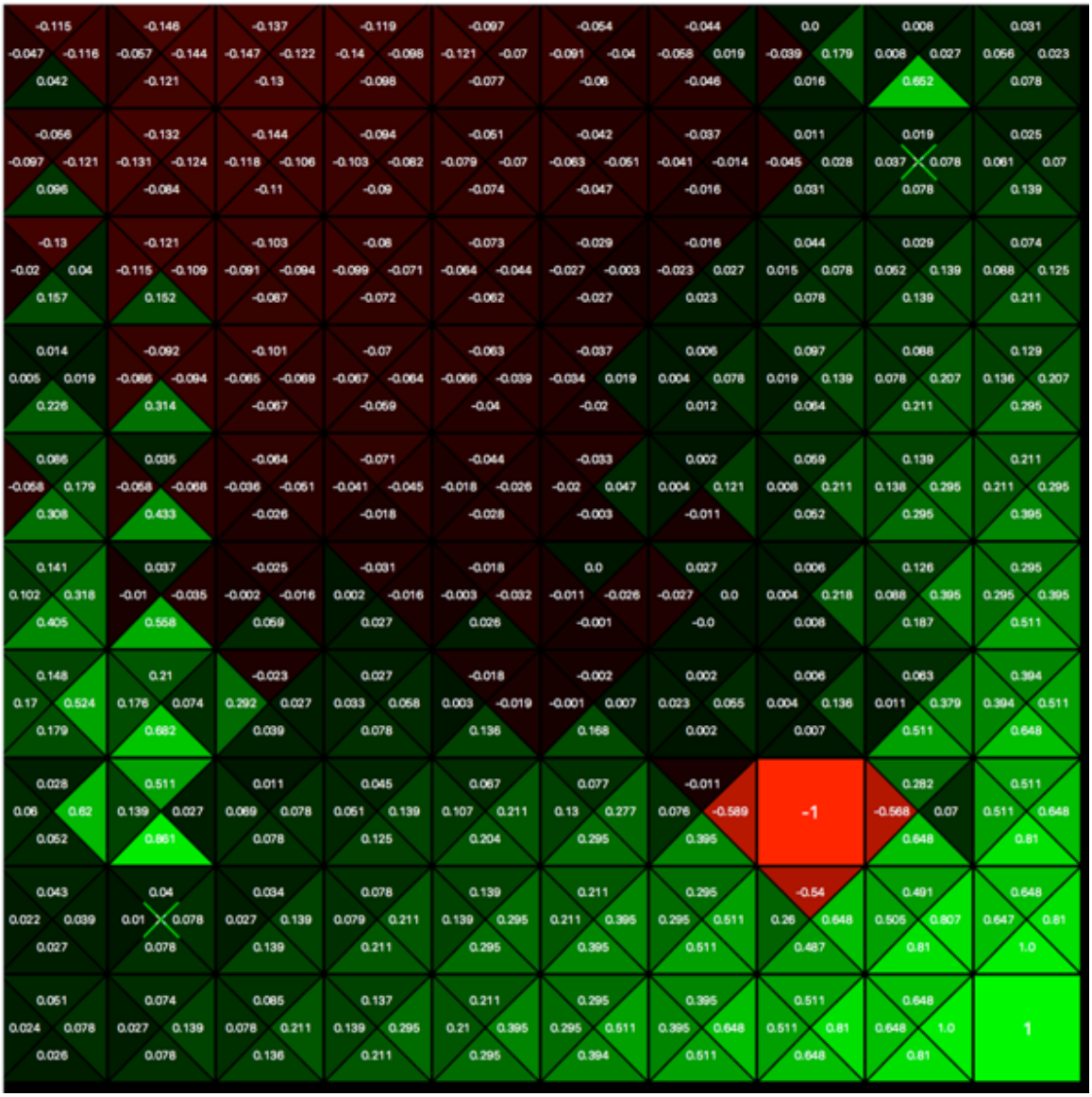

Figure 4.8: the action-values for gridworld 3.0 trained with 1000 episodes. The triangles in every state represents the actions the agent could take in that state (eg. the upper triangle represents the action going up). The color of the triangles represent the value of the action-value, negative values are red and positive values are green. The higher the absolute value, the brighter the color.

Now, we could start to modify the parameters in this experiment to see their influence. The main parameters we observed are: 1) the steps in loss function; 2) the amount of extra-reward; 3) the time scale factor γ; 4) the proportion of mini-batch and replay buffer. To compare the results, we use two indexes to judge the behavior of the agent with the parameters: 1) the average steps the agent needs for every episode; 2) the times agent received the extra-reward within 1000 episodes.

With these two targets, we pointed out good parameters we need: to help the agent to go to the good middle-points and get the extra-reward as quickly as possible.

4.2.1 Steps in Loss Function

Taking multi-steps into consideration is proved to be effective with complicated reward function. In this part we will do more experiments to explore the relationship between the the steps and the amount of the extra-reward. First we defined the loss function for taking K steps into consideration:

$$J_k(Q) := (\sum_{k=0}^{K} \gamma^k R(s_{t+k}, a_{t+k}) + \gamma^{k+1} \max_{a_{t+k+1}} Q(s_{t+k+1}, a_{t+K+1}) - Q(s_t, a_t))^2 \quad (4.3)$$

We did experiments with K=1 to 6. With each K, we firstly trained the agent with 1000 episodes and then tested with 1000 episodes. We did these loops 10 times and calculated the average for the targets: the average steps and times receive extra-reward. The outputs are shown in Figure 4.9 and Table 4.2:

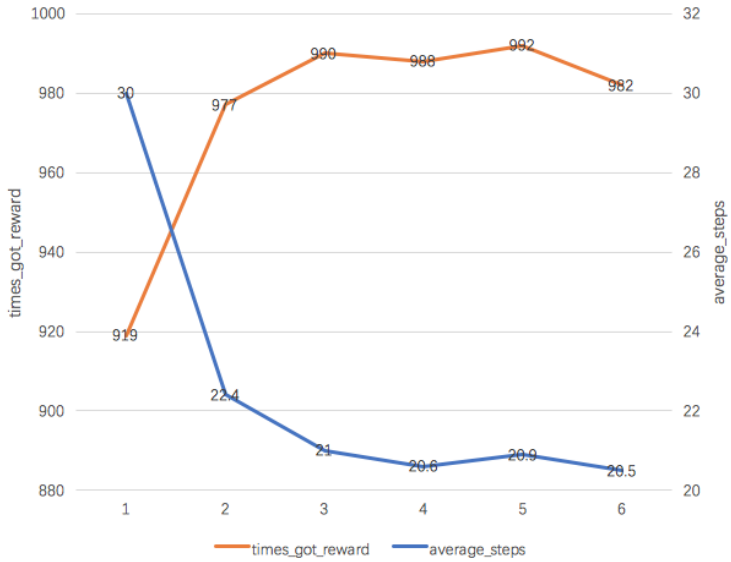

Figure 4.9: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.5$

Table 4.2: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.5$

Steps K	Average Steps Took	Times Received Extra-reward
1	30.0	919
2	22.4	977
3	21.0	990
4	20.6	983
5	20.9	992
6	20.5	982

It is showing that, by changing the steps K from 1 to 2, the steps the agent needs will decrease and the times it receives extra-reward will increase obviously. When we set K=3, the agent could almost get the extra-reward every episode (990 times in 1000 episodes) while taking 21 steps to reach the terminal state, which every close to the optimal solution taking 20 steps.

Then we took experiments by changing the number of extra-reward r_e. We chose 0.3 to do comparative experiments. The results are shown in Figure 4.10 and Table 4.3.

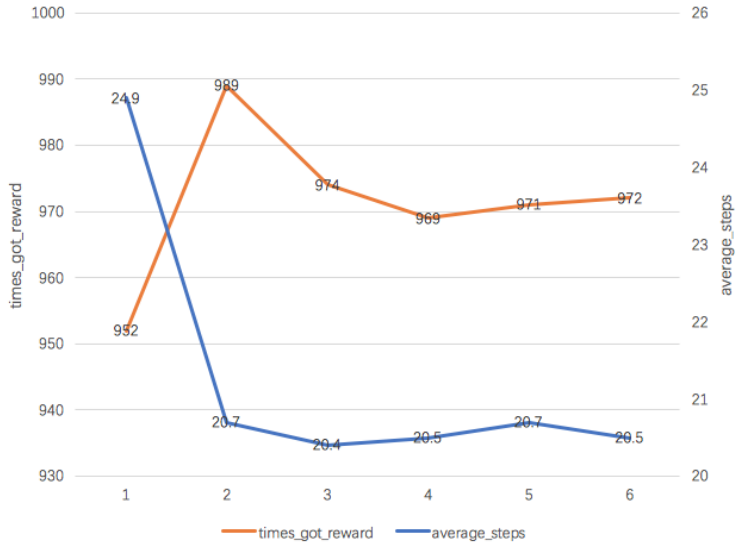

Figure 4.10: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.3$

Table 4.3: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.3$

Steps K	Average Steps Took	Times Received Extra-reward
1	24.9	952
2	20.8	987
3	20.4	974
4	20.5	969
5	20.7	971
6	20.5	970

We could see that, similarly to the situation $r_e = 0.5$, by changing the steps K from 1 to 2, the steps the agent needs will decrease sharply from 24.9 to 20.8, a result very close to the optimal. Meanwhile, the times it receives extra-reward will increase obviously from 952 to 987. However, when we keep adding K to 3 or more, the two indexes stop turning better but start to be turbulent.

This is the situation when we decrease the r_e from 0.5 to 0.3. Then we increase it to 0.8 the results are showing in Figure 4.11 and Table 4.4.

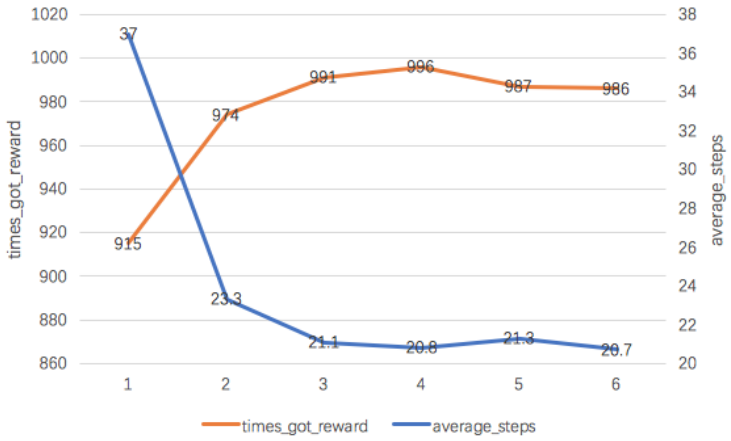

Figure 4.11: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.8$

Table 4.4: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 0.8$

Steps K	Average Steps Took	Times Received Extra-reward
1	37.0	915
2	23.3	974
3	21.1	991
4	20.6	994
5	21.3	987
6	20.7	986

Similarly, two indexes show that changing K from 1 to 2 could obviously increase the times agent receives the extra-reward while decrease the average steps it needs. But also, add one more step in loss function by changing K to 3 or more could this time slightly improve the behavior of the agent. At last, we changed the r_e to 1.0. And the results are showing in Figure 4.13 and Table 4.4. This time, the agent will receive an extra-reward with amount same as it receives when finish the task, which means somehow getting the extra-reward is as important as reaching the terminal state to finish the task. This setting seems to be not very reasonable, since then the extra-reward will mislead the agent. But the experiment results are interesting showing in Figure 4.12 and Table 4.5.

The results confirmed our idea that, with only one step further into consideration and facing some high rewarded states in state space, the agent could be easily misled.

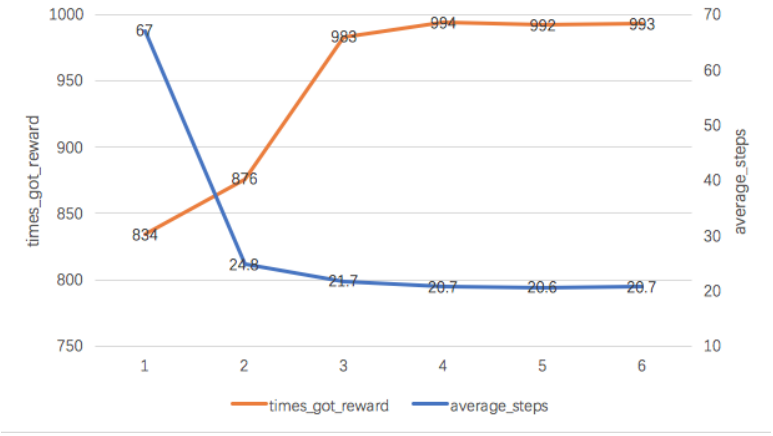

Figure 4.12: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 1.0$

Table 4.5: the average steps took and the times received extra-reward with different K when extra-reward $r_e = 1.0$

Steps K	Average Steps Took	Times Received Extra-reward
1	67.0	834
2	24.8	876
3	21.7	983
4	20.7	994
5	20.6	992
6	20.7	993

It needs 67 steps to somehow accidentally finish the task while got the reward for the lowest 83.4% of all the episodes. Under such extreme situation, again, changing the steps K in loss function from 1 to 3 could dramatically help the agent to solve the problem.

4.2.2 Time Scale Factor γ

Time scale factor in the loss function (4.3) is weighing how the agent compares the rewards close to it or far in the future. To make sure the algorithm converge, γ should be smaller than 1 and bigger than 0. We then did experiments with setting γ to different numbers. And we set K=4, $r_e = 0.5$. The results could be seen in Figure 4.13 and Table 4.6.

We could see that, if the γ is too small, the agent will take far more steps in every episode, because it will ignore the reward far in the future but only chasing the reward in front of it. But at the same time, it should not be set too big. We finally found that setting γ between 0.8 and 0.9 will be more reasonable.

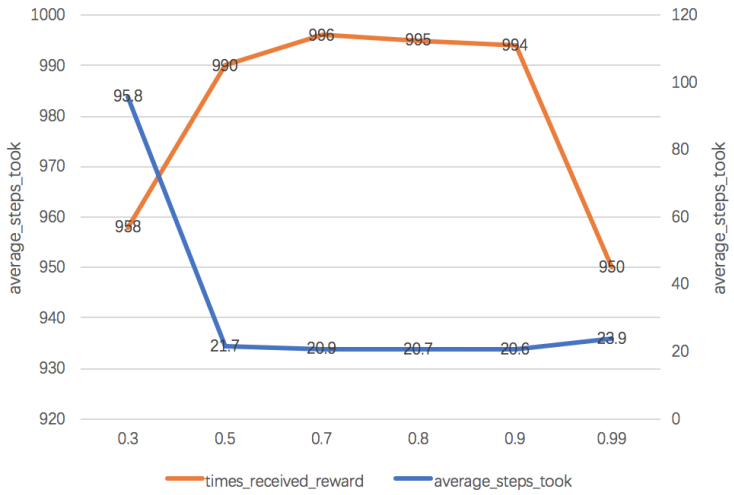

Figure 4.13: the average steps took and the times received extra-reward with different γ

Table 4.6: the average steps took and the times received extra-reward with different γ

γ	Average Steps Took	Times Received Extra-reward
0.3	95.8	958
0.5	21.7	990
0.7	20.9	996
0.8	20.7	995
0.9	20.6	994
0.99	23.9	950

4.2.3 Proportion of Mini-batch and Replay Buffer

In DQfD, we record the experience tuples $< s, a, r, s' >$ in replay buffer and sample mini-batches from it to train the agent. We chose different number of the proportion of mini-batch and replay buffer to do the experiment. In these experiments, we set the size of replay-buffer to be 2000 and then change to size of the mini-batch, the experiment results could be seen in Figure 4.14 and Table 4.7.

Table 4.7: the average steps took and the times received extra-reward with different proportion of mini-batch and replay-buffer

Mini-Batch/Replay-Buffer	Average Steps Took	Times Received Extra-reward
1/100	23.1	931
1/200	23.3	930
1/500	24.5	935
1/1000	24.9	952
1/1500	25.4	946
1/2000	25.9	928

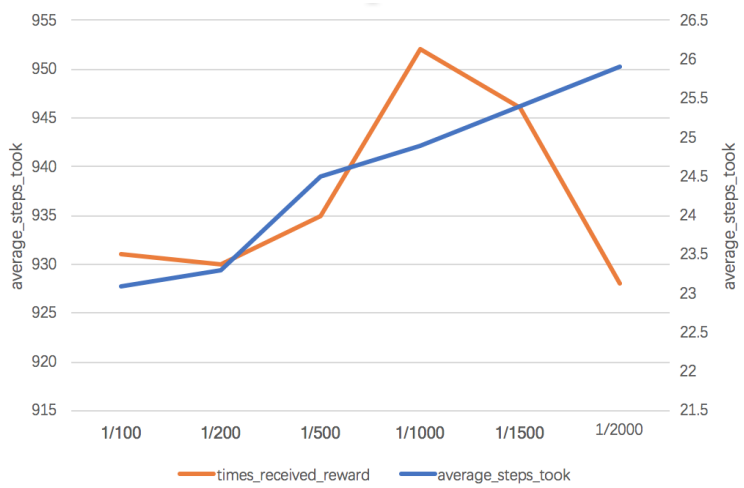

Figure 4.14: the average steps took and the times received extra-reward with different proportion of mini-batch and replay-buffer

We could see that, if the proportion of the mini-batch is too small, the agent may not be able to learn well from its observation. Therefore, when the proportion is under 1/1000, the smaller the proportion, the worse the two indexes. On the other hand, when the proportion is too big, the agent may over-fit some observed experiences, then act worse when the proportion increasing from 1/1000. In conclusion, we choose the proportion to be 1/1000.

4.2.4 Summarisation

In this part we experiment with different parameters in deep Q-learning. We found that, with higher amount of extra-reward, it will be easier for the agent to get the reward but also may mislead it. By changing the loss function taking more steps into consideration could solve this problem. We also proved that the time scale factor should not be too big or too small but set between 0.8 and 0.9. And the proportion of the mini-batch and replay buffer could also influence the behavior of the agent.

4.3 Grasps Planning and Sampling

Grasp planning is the method to evaluate many hand postures quickly with the general concept: try out lots of grasps really fast and see which work. Using a simulated environment allows us to do this part much faster than in real life, and also at a lower cost. Also, the quality metrics can give us assess on the grasps, more useful than just a binary success / fail outcome. With the quantised grasps, we could then generate our grasping demonstrations.

Grasp planning is one of the most useful tools in GraspIt!. It comes with a couple of grasp planners: 1) the Primitive-based Planner; 2) the Eigengrasp Planner.

In the first half part of this project, we use GraspIt! to plan and sample the grasps and we will disccuss the use of Dex-Net 2.0 in the second half.

4.3.1 The Primitive-based Grasp Planner

Primitive-based Grasp Planner uses primitive decompositions for the grasped object. It has a couple of restrictions: it only works on the Barrett hand, and only if the user also supplies a primitive approximation of the object to be grasped. The core idea behind primitive-based grasp planner is that, the user should choose a primitive object and then GraspIt! will automatically define some direction and distances with parameters set by user and carry out grasp planning, to limit the huge number of possible grasps.

First and foremost, each grasp carried out by the grasp planner will be in a reasonable grasp starting position. This starting position consists of a 3D palm position, a 3D orientation which is divided into an approach direction (2D) and a thumb orientation, and a hand preshape. The starting positions for sphere are shown in Figure 4.15. The balls represent starting positions for the center of the palm. Every long arrow shows the grasp approach direction (perpendicular to the palm face), which should pass through the center of the sphere. The short arrows show the thumb directions (always perpendicular to the approach) [48].

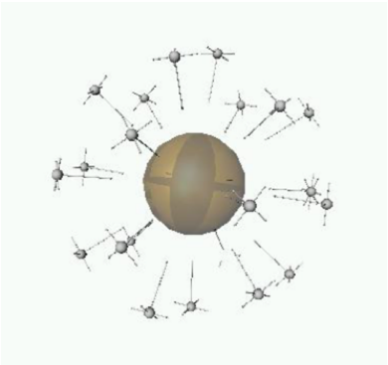

Figure 4.15: a Primitive Sphere for Grasp Planning

The starting positions are equally located above the sphere. The way we control the number of starting positions is by setting parameters to generate the grasping planner. For a sphere specifically, we need to set two parameters: the number of divisions of 360 degree and the number of grasp rotations. The number of divisions of 360 degree controls the sampling of both the azimuth and elevation angles. The number of grasp rotations controls how man grasps are planned by rotating the palm around an approach vector. We could both set the parameters or use the default ones in the window shown in Figure 4.16. After setting the parameters for starting positions, we could start the planner to grasp planning which will automatically do the grasps as well as evaluate them and finally output the results. We could use these results to train our model.

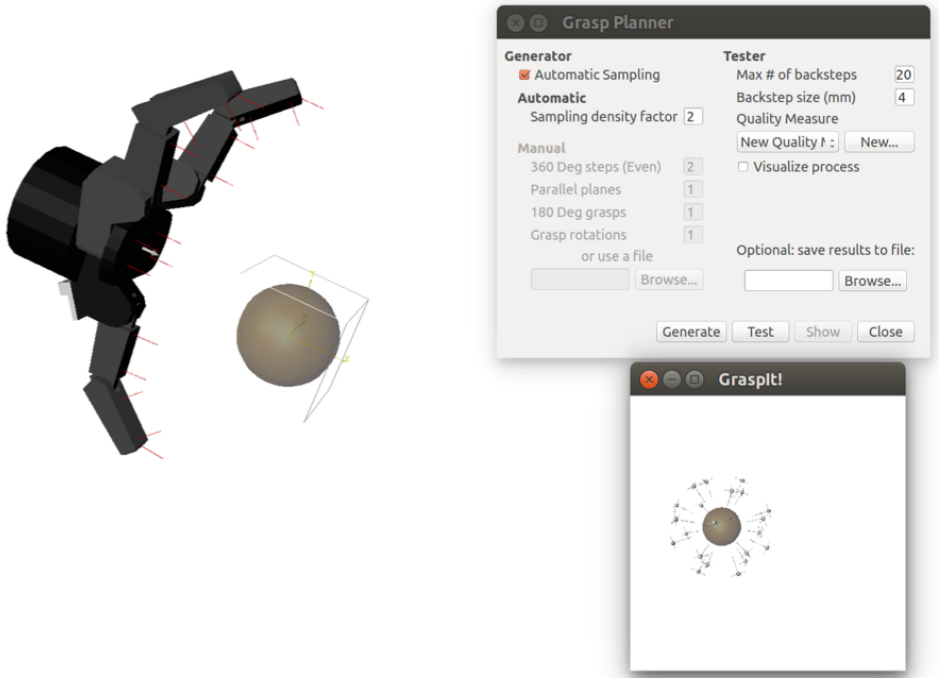

Figure 4.16: Setting primitive grasp planner

4.3.2 The Eigengrasp Planner

The two restrictions of primitive-based grasp planner limits it functionality severely. We need grasp planner for different robot hands as well as different objects. Not like primitive-based grasp planners using pre-set locations and directions to limit the number of searching, eigengrasp planner is based on the idea of eigengrasp.

Eigengrasps define a subspace of a given hands DOF space. Assuming the hand has d DOFs, each eigengrasp is a d-dimensional vector. A basis comprising b orthogonal eigengrasps can define a b-dimensional subspace. Additionally, this subspace needs an origin, which is also a d-dimensional vector. This gives the following options:

- define a hand posture using b eigengrasp amplitudes (as opposed to d DOFs);
- find the Eigengrasp subspace projection of a given hand posture which is not necessarily in this subspace.

The human hand model in GraspIt! for example, is provided with contacting points and eigengrasp directions matching those discovered through studies by Santello et al. [61]. The hand model used in that study had 16 DOF. Therefore, only the 16-DOF version of the human hand included with GraspIt! has all 6 eigengrasps discovered in the study, and then could be successfully executed the eigengrasp planer. Then later on provided 20-DOF hand model could also use the planner, but may could outcome with precise results.

However, when we try to import this human hand for eigengrasp planning, we found

that this model is broken (see Figure 4.17(a)). But with the idea of eigengrasp planning, we fixed it by ourselves (see Figure 4.17(b)).

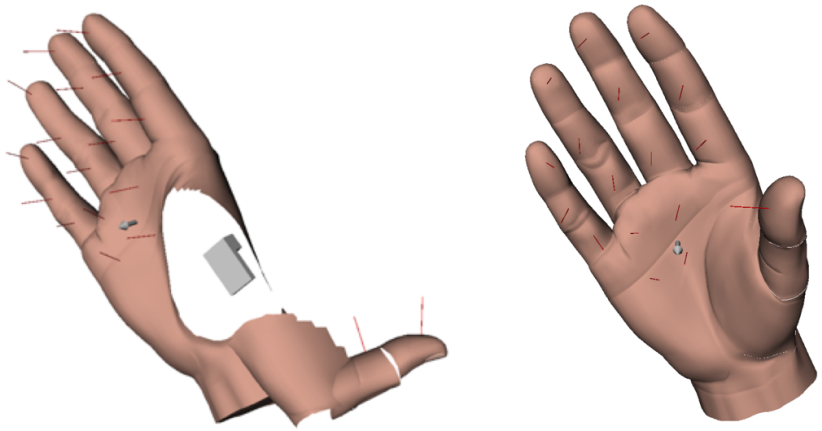

Figure 4.17: (a) broken human hand model; (b) fixed human hand model

Then we use this human hand model to do the grasp planning using eigengrasp planner. We use a sphere as the object to grasp. After 10000 times searching, the best results searched by GraspIt! using eigengrasp planner are shown in Figure 4.18(a).

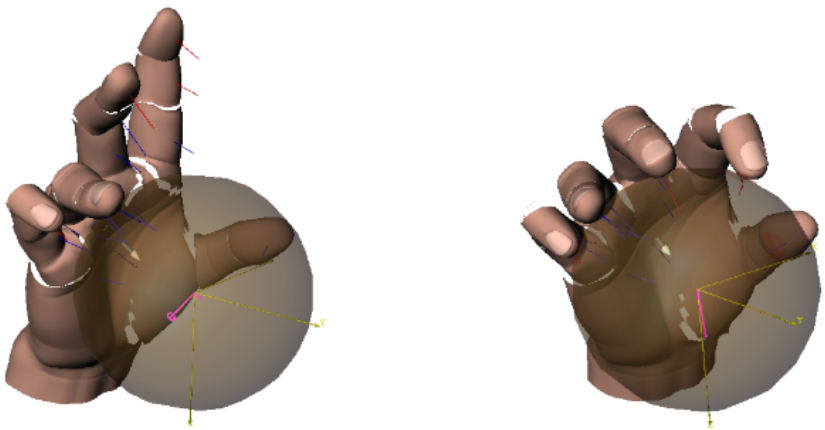

Figure 4.18: (a) results for eigengrasp planner; (b) adjusted grasping result

The gesture could be seen in Figure 4.18(a) is a very common gesture for us human-beings to grasp a sphere. But for an algorithm, it is not that obvious. During the visualised process of its searching, we could see that it started from some very bad gestures such as with some of the fingers bended or even with the back of the hand towards the sphere. And gradually, it adjusted the gesture with modifying the eigengrasp vectors.

Because the searching algorithm is using eigengrasp, the outcome gesture cannot

be a complete grasping gesture. We should then manual adjust a little, which is not difficult but just bend every finger until it touch the object. Then we will get the final grasping gesture (see Figure 4.18(b))

4.3.3 Summarisation

The primitive-based grasp planner is an easy-understanding and effective way to plan the grasps. But it is limited that we could only use it to plan the grasps by the Barrett Hand with primitive objects. Then we use the eigengrasp planner to make up the shortage of primitive-based grasp planner. With these two ways, we have generated several good gestures, which could be used in our latter experiments.

4.4 Simulation Environment Test Flight

Previously, we have built up our simulation environment MuJoCo Pro. In this section, we use it to implement some experiments to implement the functions we need.

4.4.1 Movement Controlling

MuJoCo Pro comes with several code samples providing useful functionality. One of them called simulate is quite elaborate and has realised most of the main functions of MuJoCo. At the very beginning we could simply do the modification on it after understanding its structure.

We firstly modify the simulate.cpp file by adding our own controller function, in which we use a redefined function to find the joints we want move. Here, we try to move the 3 joints control three fingers of the Barrett hand. Then, we add a small constant b on them to move the joints. Then we call the control in the main loop, therefore at each time-step the joints we chosen will be add the same constant b, and then move.

After changing the code, we could use the make file MuJoco Pro provided to compile the project and then run the excutable file named simulate with the Barrett Hand model file. OpenGL renderer will visualise our modified version of Barrett Hand, the Barrett hand with fingers opened and closed are shown in Figure 4.19(a) and 4.19(b) respectively.

4.4.2 Models Repairation and Optimisation

In this project, we are using two-finger model Baxter, three-finger model Barrett hand and five-finger model MPL in MuJoCo. Before the simulation experiments, we manually tested every model with every joints we need to control. But it is quiet disappointing that, every model has this or that bug, we fixed them one by one. Furthermore, we modified some of them to be more suitable for our project. We

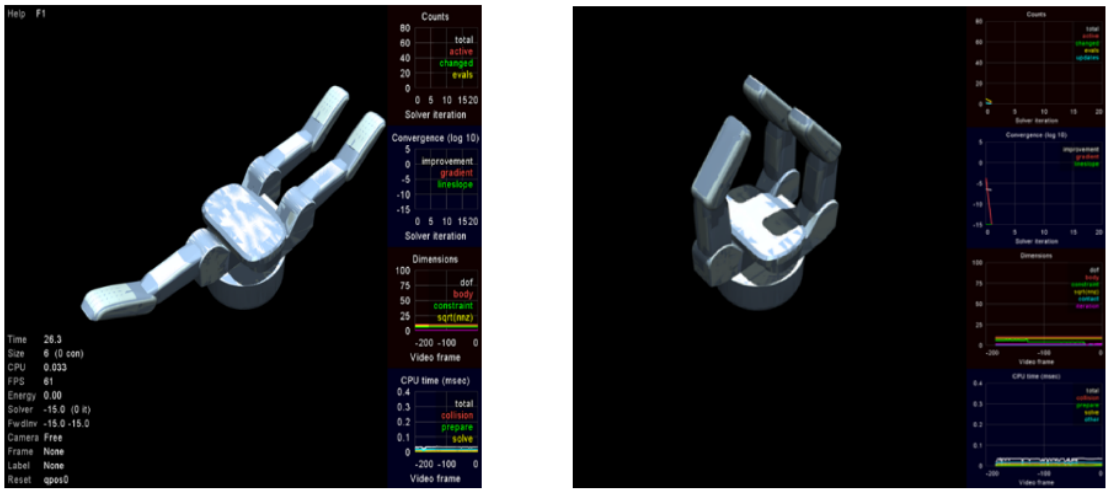

Figure 4.19: (a) opened Barrett hand; (b) closed Barrett hand

discussed the procedures of repairation and optimisation as follow.

The Barrett hand has unlimited joints. Thus every joint could rotate 360 degree. This problem results in that the fingers of the Barrett hand could penetrate itself when its joints rotate to unrealistic angles (see Figure 4.20).

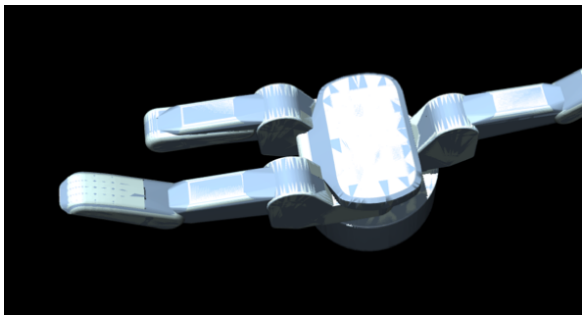

Figure 4.20: unlimited joints caused penetration

This problem has been fixed by modify the model files, in which we could add attributes to the joints. One core binary attribute for this problem is ”limited”. By setting it true, we could then specify the joint has limits and then use another attribute ”range”, to set the range for the joints in angle. We chose to set the first joint for every finger connecting it with the palm a range within 0 to 180 degree and the other two joints within 0 to 90 degree.

The Baxter model is too complicated. The model of the Baxter we found on-line is as a whole robot (see Figure 4.21(a)). Even if it is well-defined as a whole, the part of it that we are interested is only it griper. Using the whole robot to carry out the experiment could not only be a computational wasting but also bringing unwanted experiment designing difficulty.

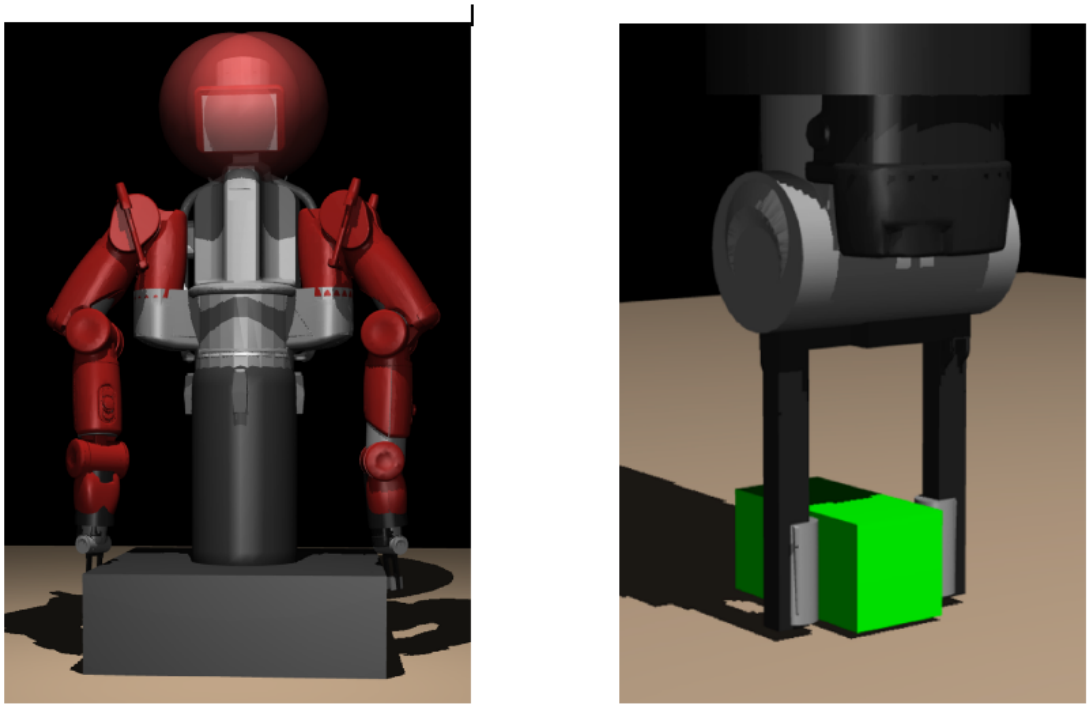

Figure 4.21: (a) the original Baxter model; (b) the Baxter griper model

Thus we imitated its original model to write our own one, the griper of the Baxter (see Figure 4.21(b)). In this model, we only implemented the waist and two fingers. Using this model, we are able to carry out all the experiments we need, by moving the griper as a whole or closing it jaw. If in the future we need to change the orientation of the griper, we could just add degree of freedom to it waist or add a forearm for this model.

The MPL hand is the best model we found for MuJoCo. Every joints is well defined, mesh files are fully equipped and the file code is clean and clear. But still, there is a severe problem with this model, it has no fraction. At the very beginning, we tried to manually control the MPL model to grasp a stick, but the stick slowly slipped out from the hand (see Figure 4.22).

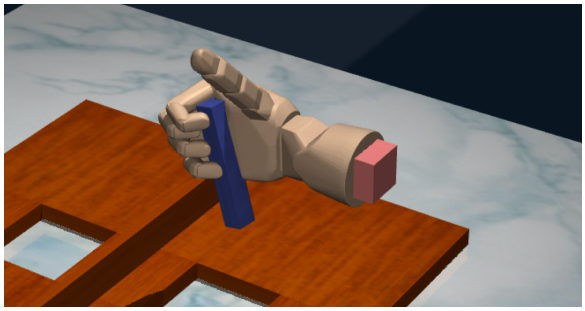

Figure 4.22: the stick slipping out from the MPL

Actually, we later on found that, almost every model has more or less this problem that, there is little fraction between the robot hands and the objects. We tried to use the Barrett hand to grasp a box, but failed due to the lack of fraction (see Fugure 4.23). The fraction is the core physical feature for our experiment, while a lot of setting in a simulation environment could influent it. It may because of either the surface of the robot griper or the object, which will be controlled by the mesh attributes. Or it could be the problem with force exerted by the joints. Also, it may be influenced by some environment parameters.

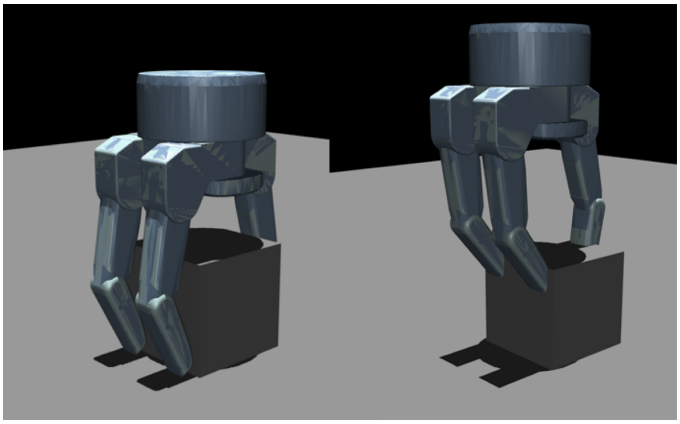

Figure 4.23: the box slipping out from the Barrett hand

With very carefully studied MuJoCo's documents, we found several parameters could influence the fraction of the model. We modified them as a whole to realise the fraction, thus list them all below with their functions.

- Contact type setting for environment. "solref" and "solimp" parameterise the function for all frictions in the environment. They will also be influenced by the "solver" setting.

- Contact controlling. Several attributes controlling the contact type: "contype", "conaffinity", "condim". The right setting for them all as a good combination will making the contact between griper and object with right frictional contact, opposing slip and rotation.

- Friction. It is controlled by attribute "friction" and it may seems to be the key, which contains the parameters for sliding friction, torsional friction and rolling friction. We could set them by ourselves, but usually the default settings are good enough.

- Object attributes. The "mass" and "density" may also influence this problem.

After setting them all with caution, all the gripers could then have fraction to successfully grasp objects.

4.4.3 Problems Discussion

Then it is the time for us to combine GraspIt! and MuJoCo together with deep Q-learning from demonstration (DQfD) algorithm. The idea is using GraspIt! to sample the grasping points on the surface of the object, and then use the output good grasping position to generate demonstrations. And use these demonstrations to pre-train the agent.

However, when we started to carry out experiment with this idea, we found us facing a lot of severe problems. The first problem came to us is that the state space for the gripers are too large. There are 8 degree of freedoms in the Barrett hand and 16 in the humanoid hand. If we combined them with the 3D position and use a quaternion representing the orientation of the hand, the state space will be far more complicated than the gridworld we tested.

On the other hand, in the GraspIt!, due to the complexity of the hands state space, if we want some meaningful output, we need to use very simple object, like a sphere shown above. Otherwise, the algorithm used by GraspIt! can so easily fall into local minimums and outcome with meaningless results, which cannot be used to create demonstrations later. But if the object is as simple as a sphere, the only information we could get from GraspIt! output is the distance between the griper and the sphere, as there is no difference between different orientations to grasp and if the direction of the griper is not aiming to the center of the sphere it is also not a good gesture for sure. Then, it became a problem to find of the right distance for a certain griper to grasp a sphere, which actually doesn't need any training in MuJoCo (see in Figure 4.24).

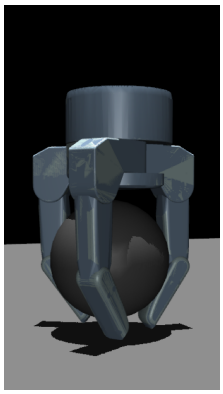

Figure 4.24: the Barrett hand in MuJoCo grasping a sphere with the output from GraspIt!

In general, it is not a problem with our structure but simply cause by the griper and algorithms in GraspIt!. If we are able to use a two-finger griper with a better grasping planning algorithm, we could then get rid of the high dimensional joints' rotations, but focus on the agent behaviour in looking for good positions on the surface of the objects. This led us for finding and taking advantage of Dex-Net 2.0, which will be discussed in the next chapter.

4.5 Summarisation

In this chapter, we introduced the early preliminary experiments we did for this project. We implemented the deep Q-learning algorithm by simplifying the task with gridworld model. And we gradually add the features we need for this project into the gridworld model making it able to represent complex task, especially for our discussion about loss function in deep Q-learning. We also did experiment with different parameters to investigate their influence and find the right settings for our project. Then we use two different grasping planner in GraspIt! to sample grasps. Next, we did some tests in MuJoCo and fixed its original models. At last, we conclude the problem with the structure that simpling in GraspIt! and training in MuJoCo.

Chapter 5

DQfD Experimental System

In the previous chapters, we prepared all we need: choosing the libraries and building the environment. We simplify the problem and did research about the parameters. Furthermore, we sampled the demonstrations and repaired and optimised the models we need for experiments. In this chapter, we will firstly introduce the second sampling simulation tool, Dex-Net 2.0 and then combine all parts we prepared together to form our experimental system, and introduce the core experiments we did using this experimental system we built.

5.1 Experiment Description

Previously we are using the structure that sampling in GraspIt! and training in MuJoCo. In this section we introduce the second structure in our project using Dex-Net 2.0 to corporate with MuJoCo.

5.1.1 Problem Description

The first structure with GraspIt! and MuJoCo turned out to be inefficient due to two main reasons:

1. All the grippers in GraspIt! have more than two fingers. It forces GraspIt! doing grasping planning in a very high dimensional state space. Even if in eigen-grasp planning, GraspIt! uses the idea of eigen-grasp to reduce the dimensionality, the dimension of its state space could still be higher than six.

2. The algorithm for primitive-grasping planning in GraspIt! limits the griper as well as the object; the eigen-grasping planning could easily fall into local minimums.

These two problems together forced us to choose very simple object like a sphere otherwise it will be too hard for our learning algorithm to realise. Therefore, in order to overcome this problem, we build the second structure with Dex-Net 2.0 to sample and still training in MuJoCo. This structure takes advantage of the good implementation in Dex-Net 2.0 with the two-finger griper Baxter.

5.1.2 Experiment Structure

Although it has been proved that Dex-Net 2.0's output has very high success rate in real-world grasping task, it may not find the points with highest grasp quality but just very good ones. It is reasonable to believe that, the best points are close to these good points. Therefore, in the second structure, we use Dex-Net 2.0 to sample with two-finger griper, the Baxter. The outcomes will be points in pairs on the surface of the object. Then we could use these points to generate our demonstration data with the algorithm DQfD. And finally carry out the experiment in MuJoCo using another Baxter model, or we could also use the MPL model, but only use the index finger and thumb.

5.2 Dex-Net 2.0

Another simulation tool for sampling we are using in this project is Dex-Net 2.0. As we introduced above, not like the two methods in GraspIt!, Dex-Net 2.0 uses GQ-CNN to generate the grasping points and it is proved to be very quick and accurate.

5.2.1 Preparing Database

Previously, we introduced the installation of the Dex-Net 2.0. But before we start our experiment with Dex-Net 2.0, we shall import a database to its workspace. Dex-Net 2.0 itself has a database with 10 objects it is good for testing but for our experiments we need far more objects ant then we need to find or generate our own database.

Dex-Net 2.0 is using HDF5 files as its database to store its models. It is not hard to find one on-line. The one we found is called dex-net-training-database, which contains models form two public mesh datasets with 1,500 3D object models. 1,371 models are from the first dataset with 50 category subset objects called 3DNet [73], and another one contains 129 laser scanned models called KIT Object Database [33]. Due to large capacity of this database, it is uploaded separately, we need download them in pieces and then use the command line to merge it by running:

$ cat hdf5.dexnet_2_database_parta* > dexnet_2_database.hdf5

to generate the .HDF5 file, which can then be opened with the Dex-Net CLI. Also, we could generate our own database using mesh files, this will not be discussed in the project.

5.2.2 Grasping Test

As we have imported a large database, we have thousands of models to choose for our experiments. Also, Dex-Net 2.0 has a few different ways to measure the grasps such as force closure, ferrari-canny matrix and robust-ferrari-canny matrix. We could have a taste by opening it example database, choosing a dataset , calculating the

matrix for a certain object and print the outcome. The grasping quality results for a vase and a lock using robust-ferrari-canny matrix could be seen in Figure 5.1.

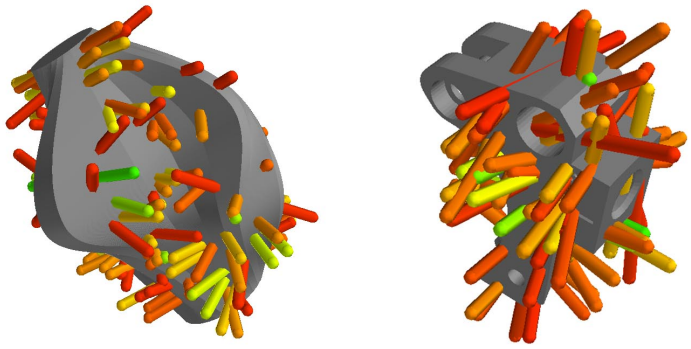

Figure 5.1: Dex-Net 2.0 output for a vase and a lock. The green, yellow and red bars represent good, bad and worse grasping positions respectively

We could see that, the output is showing the good grasping points at some positions that the local structure is flat or close to the center of the object, which confirms with our daily experience.

5.2.3 Grasping Planning

Then we could start our own grasping planning with the database we prepared. We choose 4 models in the dataset 3DNet: a coffee box, a rusk, a cereals box and a salt cube. We import them one by one into the workspace and use the rubust-ferrari-canny matrix to calculate the grasping quality (see Figure 5.2).

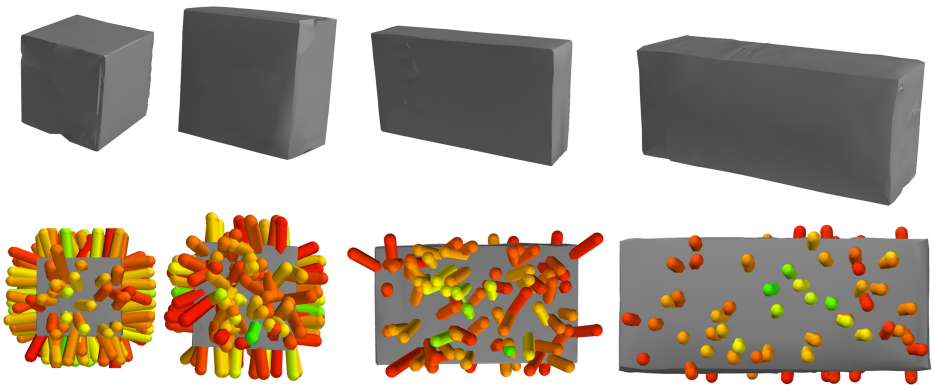

Figure 5.2: four models: a coffee box, a rusk, a cereals box and a salt cube with their grasping quality by rubust-ferrari-canny matrix

The reason for choosing these four models is that, they are all cube-like but also modeling from real-life objects. They are convenient to later on be modeled in our experimental simulation environment but also interesting to doing research about. We could see from the grasping quality results that, most of the good points are in the center of the object but still different from each other due to the difference in shape and size.

Then we choose the five points with highest grasping qualities to print out there locations on the objects, the results for the coffee box and rusk are shown in Figure 5.3.

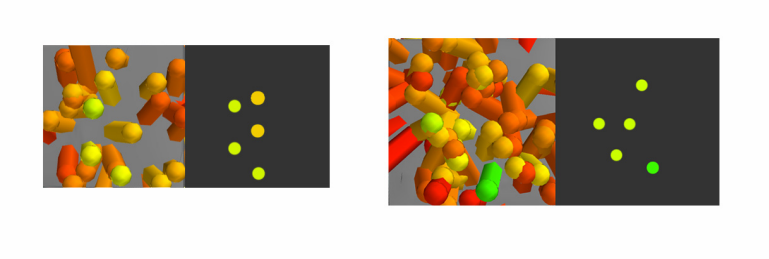

Figure 5.3: five points with highest grasping qualities for coffee box and rusk

Now we could see clear that, the positions with highest grasping qualities are mostly in the center of the object. Because the coffee box model is quite small, the five best grasping points are not be given high scores being printed yellow or orange. While there is a point with high score being print green on rusk model. It is not surprising that, this point is not right at the center of it but close to the bottom of it.

Then, we could start our training using these points as the good-middel-points we used in previous discussion, and create our demonstrations to implement the DQfD algorithm for robot grasping task.

5.3 Structure for Experiment

Now it is time for us to implement the whole learning experiment environment making very parts we prepared to form a complete project. To complete this task we need import our networking library ZeroMQ into the both the deep Q-learning program and the simulation environment MuJoCo to make them a Python client and a C++ server.

5.3.1 Python Client

As we discussed before, in our project, we are using TensorFlow in a Python program doing the deep Q-learning algorithm. Within a state s, the deep-Q learning agent calculates the action-value Q for every action, chooses the action a with highest action-value with ϵ-greedy idea, then sends this action command to simulation

environment and waits for the results. After receiving the results containing the information about the next state s' and the reward r, it will carry on the deep Q-learning calculation with the next step.

Now we should import the ZeroMQ library for realising the client. As we have already installed ZeroMQ and pyzmq, it could be done by directly import zmq. Then we create a ZeroMQ content for our networking requirements, and define a REQ (request) socket binding to port 5555:

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.bind("tcp://*:5555")
```

Then after our agent has decided which action to take, it could send encoded action command (such as using 0 to represent the action "move up") with this socket and start to wait for reply immediately:

```
socket.send(0)
message = socket.recv()
```

We could put these two lines of codes in a loop to keep sending action command and receiving the results.

5.3.2 MuJoCo C++ Sever

For the simulation environment part, it will also use the networking library ZeroMQ. But here, it will use another part of the library to work as a C++ sever. After receiving the request, MuJoCo needs to do the simulation and then sends back the results as the reply. To realise these process, we could first include the .hpp file, create a context and a REP (reply) socket binding the same port 5555 as we did in Python:

```
#include <zmq.hpp>
zmq::context_t context;
zmq::socket_t socket (context, ZMQ_REP);
socket.bind ("tcp://*:5555");
```

Acting as the sever, MuJoCo needs to keep waiting for the request. Then, after finishing the simulation, it needs to encode the results and send it back to the Python client:

```
zmq::message_t request;
socket.recv (&request); // Wait for next request from client
// Do the simulation here
zmq::message_t reply;
memcpy (reply.data (), result); // encode the result into reply
socket.send (reply);
```

Because we include the the head file zmq.hpp here we should be caution that it is should be properly built. As MuJoCo itself is built with CMake, we need to change the makefile for it adding a command -zmq.

5.3.3 MuJoCo Experiment Settlements

Then we could move on the set the experiment scene. We firstly import a wide plane as the experiment table. And then we import an cube setting it to different shapes referring to the models we sampled in Dex-Net 2.0. As we discussed before, we could use both the Baxter hand model or the MPL model as the griper to pinch in this experiment. After all these pre-setting, the whole scene could be seen in Figure 5.4(a) and (b) and for the Baxter griper and MPL respectively.

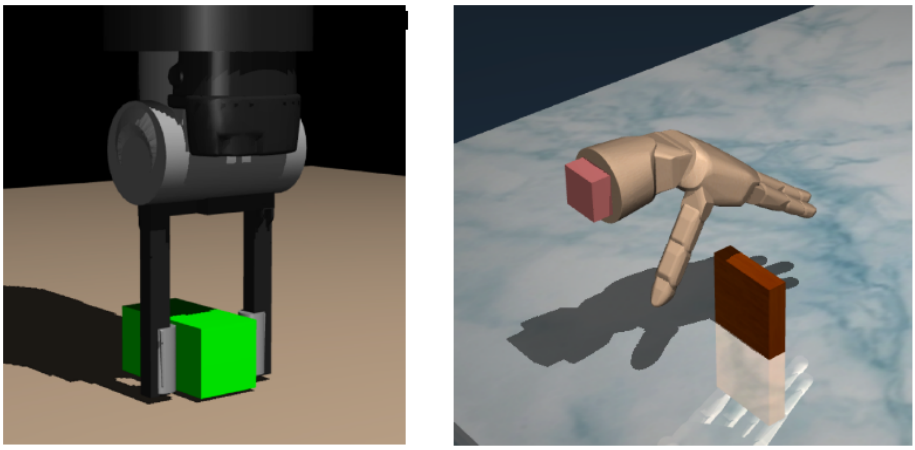

Figure 5.4: (a) the Baxter griper model; (b) the MPL model to pinch

Then we should set a start position for the griper, the origin of the coordinates. We use the point that griper's thumb contacts with its index when closed its fingers as the position of the agent. And then set the origin above the left corner of the object, with the height twice as the height of the object. We also set that moving along the width to its right corner is in the direction of x-axis, moving down towards the object is in the direction of y-axis (see Figure 5.5).

5.4 Discrete State Space Experiments

In this experiment, we are using DQfD algorithm, which is a kind of deep Q-learning. To train the neural network, we use the state of the agent, the position of robot hand as the input and output the Q-values for all actions at that state. We firstly divide the whole state space into $N \times M$ regions and describe it discretely and use a sparse matrix to represent the position of the agent. For example if the agent is at the up left corner, the matrix representing its state shall be a N by M matrix with every element equals to 0 expect the element at (0, 0) equals to 1; if the agent is at the down right corner, the matrix should have the only element at (N-1,M-1) equals to 1 and the rest

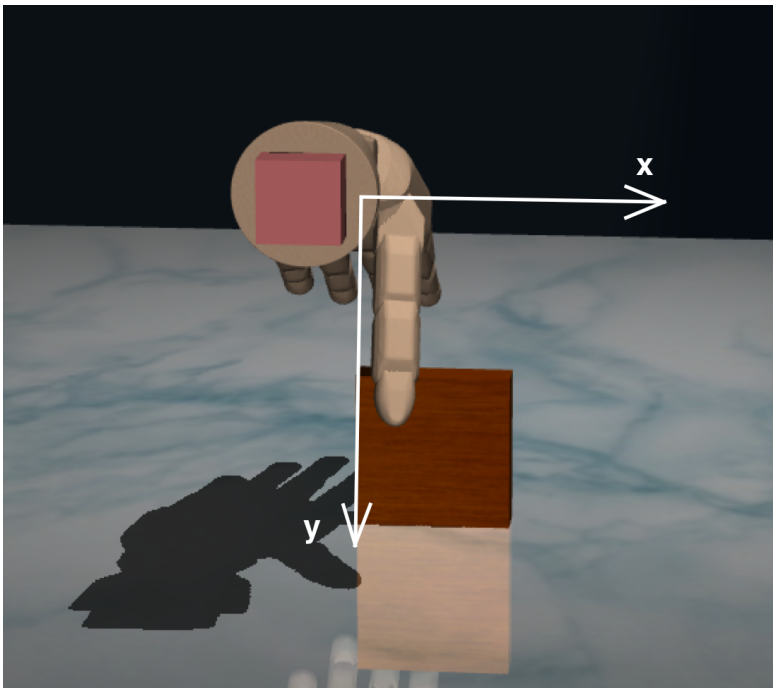

Figure 5.5: set the origin and the start position

0. There are many advantages of using a sparse representation. Sparse representations describe complicate information energy with minimal number of parameters using over-complete dictionaries [55], making it easier to process the information.

5.4.1 10×5 Situation

We firstly divide the state space into 10×5 small regions to do the experiment. Therefore, in the divided state space, the regions from row 0 to 4 is the space above the cube. If the agent try to grasp at these areas, it will definitely fail. And the regions from row 5 to 9 is the cube. If the agent try to grasp at these areas, it may succeed, which depends on the exact position. We don't use DQfD here yet, but just original deep Q-learning algorithm since this state space is very simple. At the beginning of each episode, we start the agent randomly in the state space, let it interact with the environment by its policy. After 500 episodes training, we print out all Q-values as shown in Figure 5.6. This time we have five action-values for every state, and we use the central square to represent the fifth action: the grasping action.

We could see that, after 500 episodes training, the agent has already generated a policy to move from the upper space to the lower space in order to be in a better position to grasp. Later, we use this policy to tested 100 times. Each time we started the agent randomly in the state space, and the agent successfully grasped the object 100 times, which is not surprising due to the simplicity of the state space.

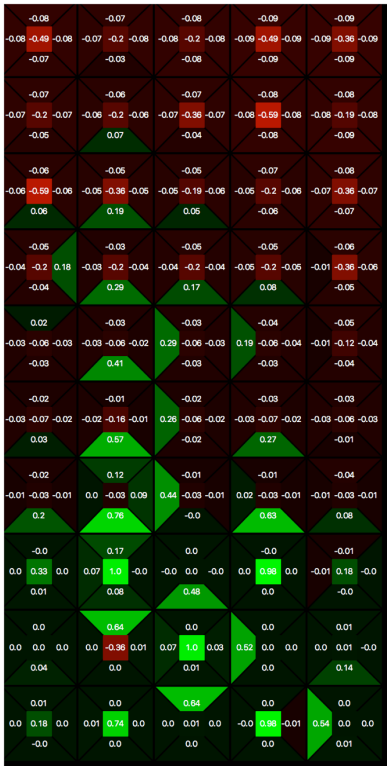

Figure 5.6: Action-values for the 10×5 discrete state space

5.4.2 20×10 Situation

Then we divided the state space into 20×10 regions, which means the regions from row 0 to 9 is the space above the cube, while from row 10 to 19 is the cube. Also, every step for the agent now is half as far as in the previous part. This time, the state space is big enough for us to use DQfD to compare with DQN. As we need demonstrations for DQfD, we use the similar method we introduced in section 5.3.1 to generate our own demonstrations:

1. Start the agent from good-middle points created from Dex-Net 2.0 at the beginning of each episode;

2. Train the agent with reinforcement learning for N episodes;

3. Record the agent's behavior with the tuple $< s, r, a, s' >$ after training for M episodes.

We chose N = 1000 and M = 100. Then we use these demonstration to pre-train our agent 500 times before it starts to interact with the environment. We compare the agent's behavior trained by DQfD and DQN, the action-values could be seen in Figure 5.7(a) and Figure 5.7(b) respectively. And we used these two agents to test for 1000 times. The agent trained by DQN has 81.8% success rate to successfully grasp, while the agent trained by DQfD has 99.6 success rate. Therefore, it is quite obvious that using demonstrations is efficient.

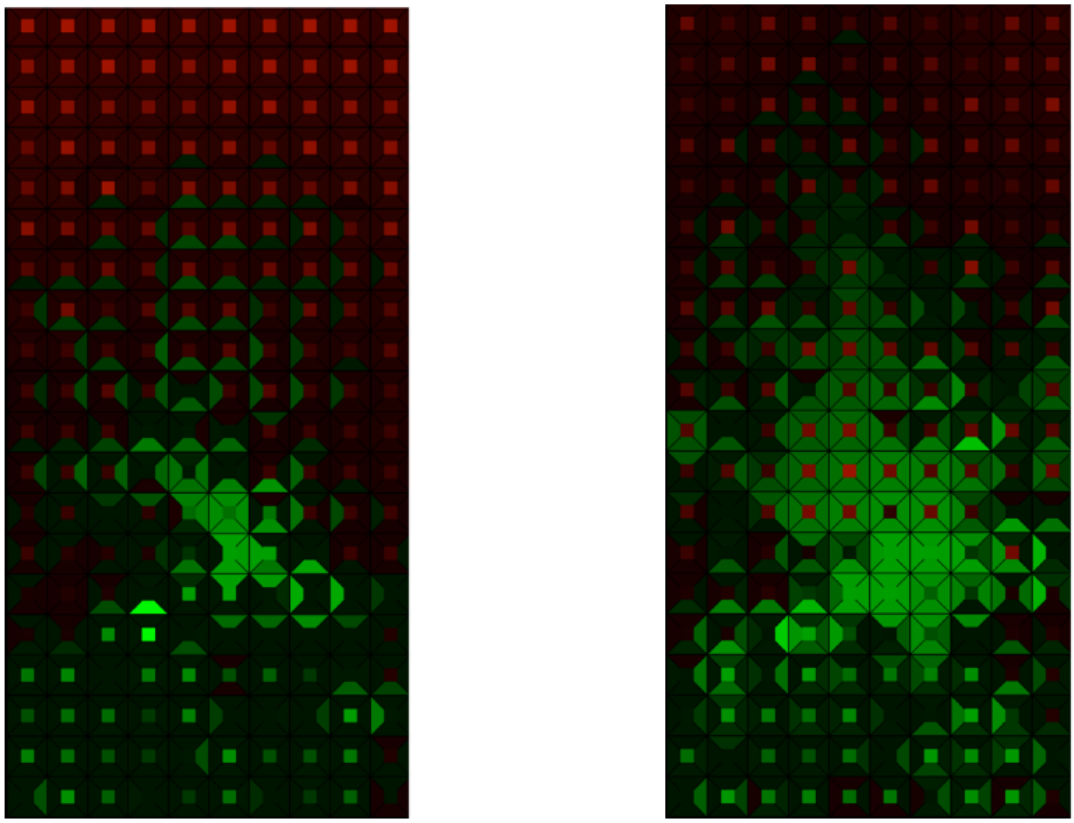

Figure 5.7: (a) DQN trained action-values; (b) DQfD trained action-values

We could firstly see from the figure that, even though the training time is the same 1000 for two algorithms, the agent in DQfD has generated more local details, especially in the middle of the the state space.

Then if we have a closer look at these two figures, we could found that, in Figure 5.7, DQN just generated some paths to move from the higher spaces to the lower spaces, the space closer to the cube. And this policy do help the agent in it to have a reasonable success rate around 80% to successfully grasp. On the other hand, the agent using DQfD has generated more paths.

Also, we could noticed that, in the higher space the DQfD's action-values are darker red. This is because the agent using DQfD spent fewer times grasping in the higher space, but move down instead. It proves that the agent will act more reasonably after pre-training.

5.4.3 40×20 Situation

Then we make one step further dividing the state space into 40×20 regions. This time we do experiments for different deep neural network models as well as different loss functions. First, we test two kinds of NN model and a CNN model. The first NN model has one hidden layer and the second has two hidden layers. We transform the 40×20 input state matrix into a 800×1 vector. And then at each hidden layer, there are 800 nodes. At last, there are still 5 output nodes.

In the CNN model, the input is 40×20 state matrix. It is a sparse matrix with only one element has value 1 and the rests are 0, therefore we don't need any preprocess. Then there are 5 feature maps in the first convolution layer. Each unit in a feature map has 25 inputs from a 5×5 area in the input matrix. The convolution layer is followed by a max-pooling layer. The third hidden layer has 10 feature maps receiving a 2×2 area from the previous layer. This second convolution layer is also followed with a max-pooling layer and finally at the top using a NN to output.

We trained these models and tested them after 1000, 2000, 5000, 10000 and 50000 episodes training (seen in table 5.1). We also print out the Q-values for NN (see Figure 5.8), 2NN (see Figure 5.9) and CNN (see Figure 5.10) respectively.

Table 5.1: The success rates for three different networks:%

Number of episodes	one hidden-layer NN	two hidden-layers NN	CNN
1000	24.1	25.0	45.0
2000	24.3	25.1	55.2
5000	40.1	59.6	66.8
10000	48.5	64.6	84.5
50000	99.9	99.8	99.7

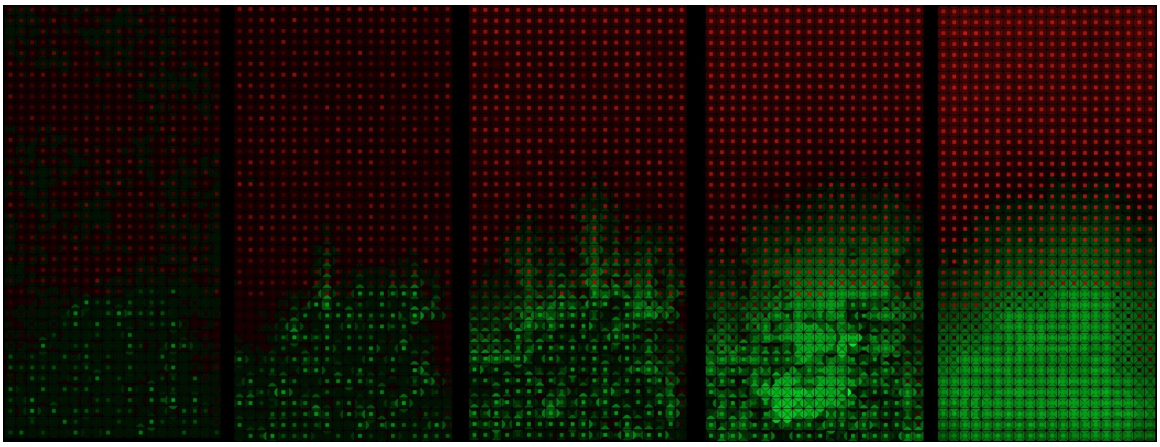

Figure 5.8: Action-values for the 40×20 discrete state space using NN

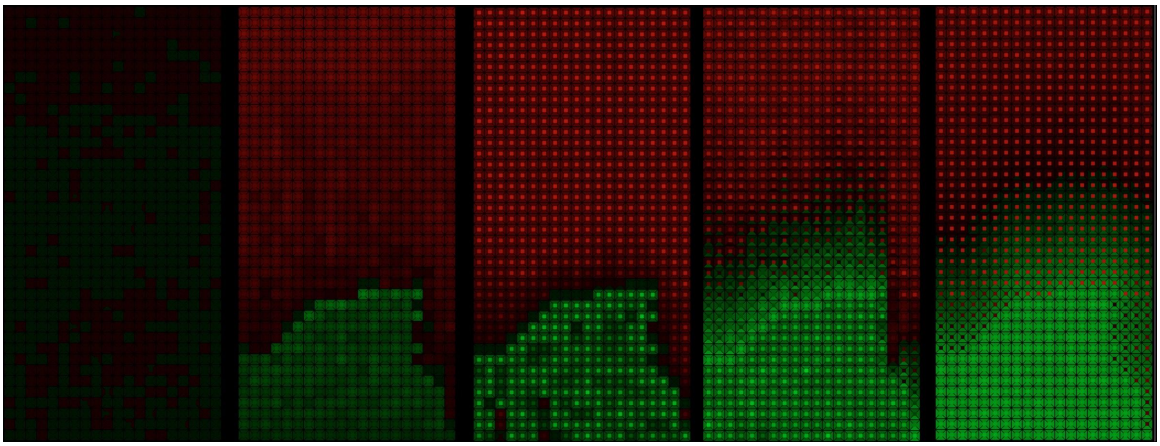

Figure 5.9: Action-values for the 40×20 discrete state space using 2NN

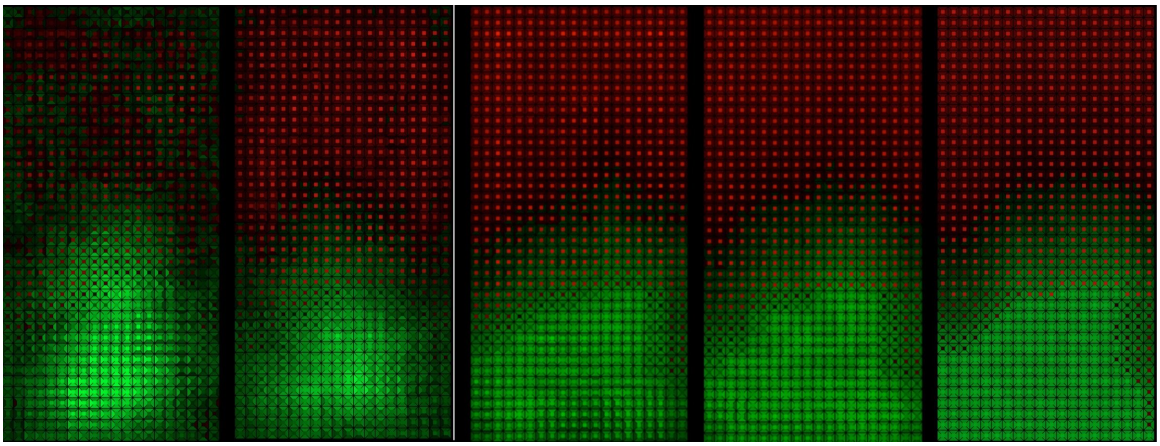

Figure 5.10: Action-values for the 40×20 discrete state space using CNN

We could see that, after 50,000 episodes training, three neural network models generated almost the same action-values, and all of them has almost 100% success rate to pass the test. It indicates that, with these action-values, the agent could always found the right position to grasp the object, no matter where it starts.

However, it is more interesting to observe the evolution of the action-values. As for the NN model with one hidden layer, we could in Figure 5.8 that, the action-values are changing point by point and in the end grow together to make the whole area. But for the other two models, it seems at the very beginning of training, the action-values are changing together. We could see that, with only added one more hidden layer, the 2NN could make obvious progress in testing. And also, its shape of the action-values generated much faster that with only 2000 episodes training, it generates a very clear boundary of an area that could succeed. And then with more training, it improves with more details.

As for the CNN model, even only trained with just 1,000 episodes, it has somehow already generated a police that the lower space has higher chance to successfully

grasp, thus we could see a very obvious green area. After 10,000 episodes training, the action-values are very close to the final ones, thus we could conclude that, CNN models could extract the information from the episodes much faster than the other two models.

Then we compared different loss functions referring to the multi-steps Q-learning we discussed above. Here, we used deep Q-learning with 2,3 and 4 steps in loss function, we call them Q(2), Q(3) and Q(4) respectively. We also compared the Monte Carlo method, as the algorithm to take whole episode to train. We list the success rates for Q(2), Q(3), Q(4) and Monte Carlo method in Table 5.2.

Table 5.2: The success rates for tests with two different networks:%

Number of episodes	Q	Q(2)	Q(3)	Q(4)	Monte Carlo
1000	24.1	42.9	42.8	40.1	33.0
2000	24.3	52.1	48.6	45.5	43.7
5000	40.1	62.2	73.4	65.8	64.0
10000	48.5	82.0	94.4	84.5	68.0
50000	99.9	100.0	100.0	99.9	99.9

We could see from the table that only with one step added, the Q(2) could preform much better than the original Q-learning. And then we output the action-values. Since we noticed that the action-values for the Q(2), Q(3) and Q(4) are very similar, we just print out Q(2) (see Figure 5.11) with Monte Carlo method (see Figure 5.12).

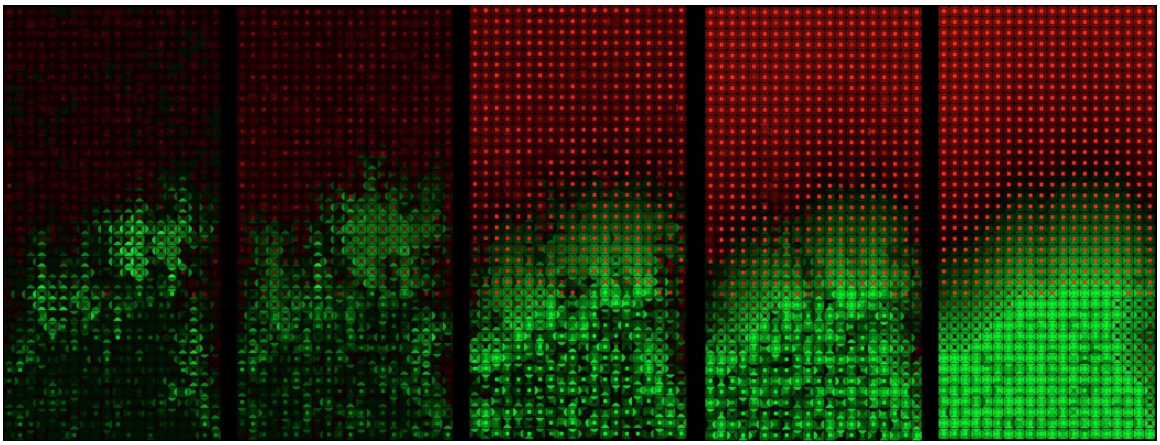

Figure 5.11: Action-values for the 40×20 discrete state space using Q(2)

Since the Q(2) has much higher success rate with only trained by 1000 episodes, we take a look at its action-values after 1000 episodes training. We could notice that, the biggest difference is that, it has a brighter area in the center of the state space, which means the action values here are higher than the ones train by original Q-learning. This is because Q(2) takes one step further, and then it is more easy for the state in the center of the state space to get information from the lower stats, the

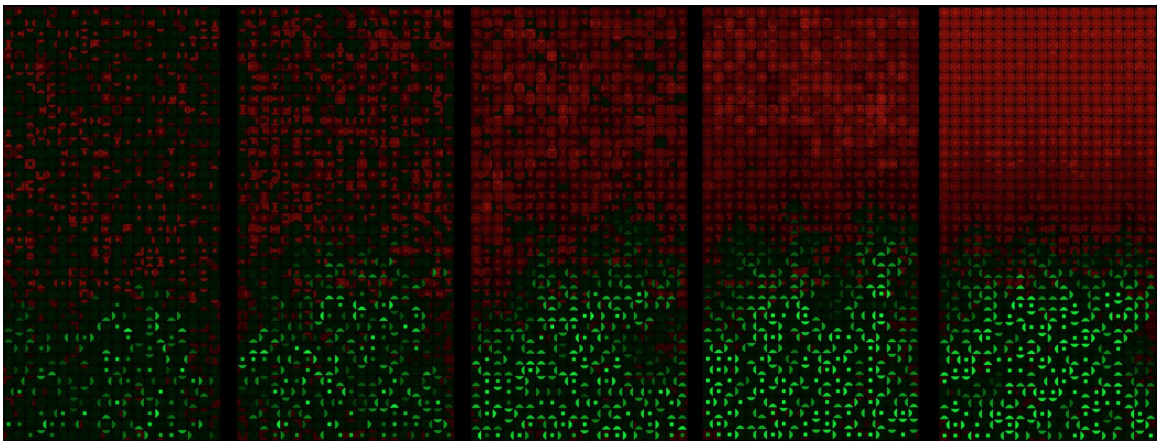

Figure 5.12: Action-values for the 40×20 discrete state space using Monte Carlo Method

states with higher Q-values that could successfully grasp the object.

The action-values generated by Monte Carlo method is much different from the others showing in Figure 5.12. Its action-values are like separated points. This is because every time the agent is use the whole episode to train.

5.4.4 Comparison of Deep Neural Network Models

In the first half of previous experiments, we compared different deep neural network models (see in Figure 5.13(a)). We could see that, with 50,000 episodes training, every model has the ability to generate the right policy with almost 100% success rate. But before that, especially when trained with around 10,000 episodes, CNN model performs better than others.

It may because CNN model could realise the location information form the in-

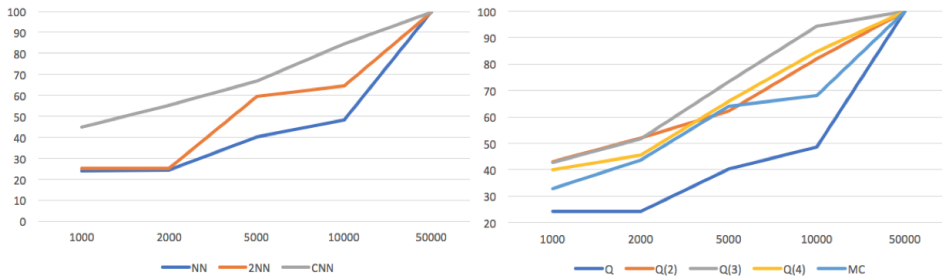

Figure 5.13: (a) comparing NN, 2NN and CNN; (b) comparing Q-learnings and MC

put, thus even with limited episodes to train, it could somehow generate the policy to generally move the agent to the down middle part of the state space before trying to grasp and then gaining a higher success rate.

In the second half we compared different Q learnings with Monte Carlo method

(see in Figure 5.13(b)), although Q(2) algorithm only takes one more step into consideration, its output is a lot improved. And then, Q(3) perform even better than Q(2). But it is not the case that taking more step ends in better outcomes. Q(4) is not better than Q(3). Monte Carlo method taking whole episodes to train, performs close to Q(2).

5.5 Continuous State Space Experiments

In the real world, the state space for a robot client is not discrete but continuous. After changing into using continuous state space, we will still train our agent will a certain step size, for example, we will still divide the whole state space into N$\times$M region to train the agent. But after training, we may be able to sample any points in this state space, which means we could start the agent from any state and it will still be able to use its policy to choose actions. For the best situation, it could still find a good position to successfully grasp the object.

To change our model to the form that could learn from continuous state space we have to change the structure of our neural network. First of all, we should change the input of the network. We can no long use sparse vectors or matrices to represent the input state but use two continuous variables x and y representing the coordinates of the agent to be the input state.

Then, it is for certain that, we need to add more hidden layers and/or more nodes for each layer in our neural network model as we now have far lower dimensional inputs. We firstly chose to use three hidden layers and kept adding the nodes until it could generate a policy to have a success rate higher than 75% after 50,000 episodes training. To reach this aim, we used 2000 nodes for each layer. The action-values after training are shown in Figure 5.13(a).

Then we chose to use four hidden layers. This time, each layer only needs 20 nodes. The action-values output from these two networks are shown in Figure 5.13(b).

5.6 Complex Object Experiments

All the situations we discussed above are using cube-like objects. What if we train our robot to interact with more complicated objects, with more local details in geometry? Can this structure with deep Q-learning algorithm still generate a policy to perform well in test? It is very interesting doing experiments to find answers these questions.

5.6.1 Find Models

As we have two simulation environments, one for sampling and one for experimenting, we need to find models for both of them. It is very easy to find the model we

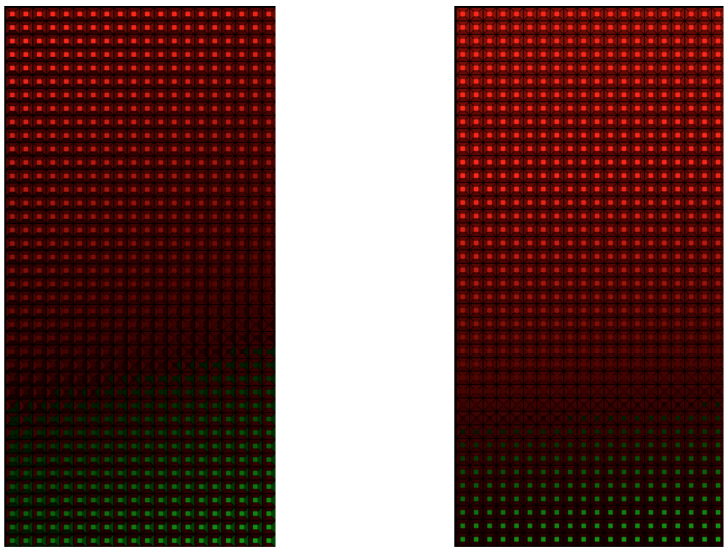

Figure 5.14: (a) action-values three hidden-layers; (b) four hidden-layers

want for Dex-Net 2.0 as we introduced before that we have already successfully imported a database with 1,500 models. But it is a little bit harder to find a similar one for MuJoCo. But MuJoCo has provided a method for us, that we could import mesh files to our models. Thus, we only need to export the mesh file form Dex-Net 2.0 and then import it to MuJoCo's model file, we could then have the same model in two environments. Then We used this idea to do an experiment with a hammer we found in Dex-Net 2.0's database (see Figure 5.14).

Figure 5.15: hammer model and grasping planing results

5.6.2 Experiment Results

In this experiment, we used the discrete state space and a NN with one hidden layer. And the original is above the hammer. the distance from O to the plan along the X-axis is as long as the hammer's length, therefore we divide the state space into 20×20 regions. The action-values generated after 1,000 episodes training are shown in Figure 5.15. This time, we also draw out the location of the hammer in the space for reference.

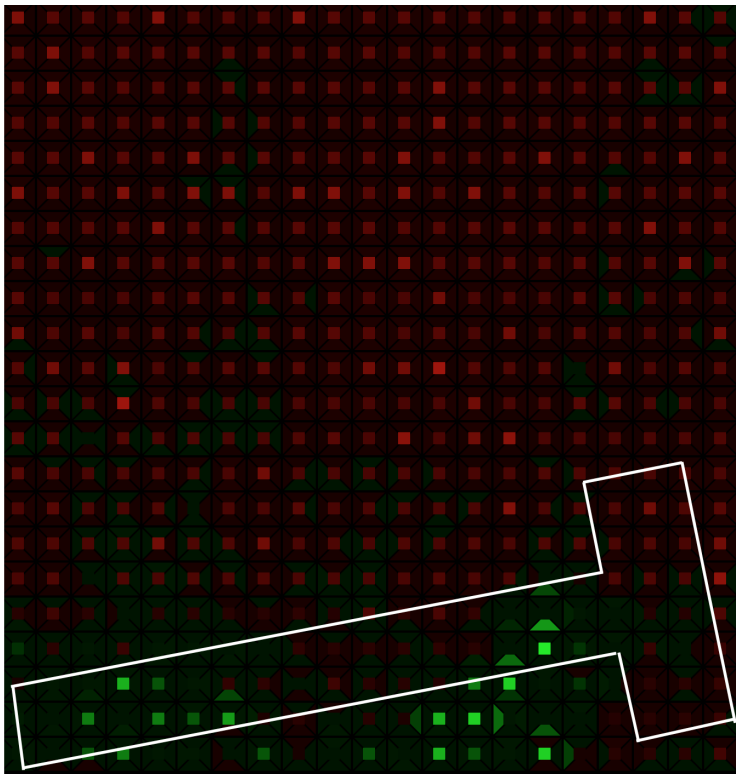

Figure 5.16: action-values for the hammer model

It is clear that, the agent has found out some point on the hammer itself that good for pinch grasping. But also, it found some better positions in the space to grasp that are under the hammer. It is very interesting that, in these positions, when the agent tries to pinch the hammer, in the simulation tools the MPL model is actually trying to hold the hammer. This is an extreme example for using DQfD with Dex-Net 2.0. Although Dex-Net 2.0 directly outputs some points with good grasping qualities, they may not contain the best ones, or the best positions may even not on the surface of the object at all: for a hammer like we used in this experiment, holding is a much better gesture than pinching. Therefore, we use DQfD algorithm to train the agent to learn to grasp the object carrying the information from Dex-Net 2.0, but not totally depending on it.

5.7 Summarisation

In this chapter, we introduced the core experiments we did in this project that to implement the idea of DQfD: using Dex-Net 2.0 for grasping planning to generate some good grasping points; using DQN to train agents locally around these good points and then record experience tuples as the demonstrations; using demonstrations to pre-train the agent before sending it to interact with the environment. And then we proved that DQfD algorithm converge faster than the original DQN algorithm and could adapt to the environment faster getting better results with less training episodes. We also compared different deep neural network models as well as differ-

ent loss functions. Then we realise the idea that to represent the input continuously and proved that it needs deeper neural network models. At last, we did experiments with more complicated and realistic object.

Chapter 6

Evaluation

In this chapter we evaluate our implementation via its advantage and disadvantages.

6.1 Advantages

- **Accuracy**. Dex-Net 2.0 is proved to be very accurate in grasping experiments with high success rate. Our method is based on Dex-Net 2.0, using the best outcomes of it to generate demonstrations to pre-train the deep neural network for a Q-agent. Also, our implementation can not only sample the space on the surface of the object like Dex-Net or other grasping planing algorithm, but also learning the states outside the object, the whole environment. Therefore, it is reasonable to believe that, this agent could be as accurate as the original Dex-Net algorithm or even more.

- **Making use of other agents' records or human experiences.** In this project, we are using different agents to generate demonstrations and to interact with the environment, which could be regarded as two different robots. One robot learns another ones' experience. By making other agents' records or human experiences into demonstrations, DQfD could make use of them to train its deep neural network without interacting with the environment. It could be useful when there is no accurate simulation but precise records or human experiences.

- **Reducing training time**. The DQfD algorithm reduces the training time from original DQN algorithm because the agent could perform well at the beginning of training. And our experiments have proved the same idea that, after pre-training, the agent could not only finish the task more quickly in the first few episodes, but also generate a good enough police within fewer episodes training.

- **Acting reasonably at the beginning.** Pre-training the agent can help it act reasonably at the beginning of getting in touch with the environment, instead of randomly trying actions with a random policy. Thus, if the training is car-

rying out with real robot and real models this strength can help prevent some severe damage.

- **Low hardware requirement**. Since we have proved that the whole project could be simulated, it requires little for the hardware. These experiments could be executed without real robot models or objects.
- **Generalisability** Not like supervised-learning, reinforcement learning doesn't require labels for training data. Therefore, this method could be used to train with any model. And we have proved that it has the ability to generalise with incoming episodes, and it is robust when we change our models form simple cube-like objects to more complicated ones.

6.2 Disadvantages

- **Calculation and space for demonstrations**. As we are using the DQfD algorithm, we need to train our agent with demonstrations. Both creating and using the demonstrations require computational capabilities as well as storage spaces, which is not preventable.
- **Requiring accurate models**. Since our experiments are taken in simulation environments with models, the functionality will depends on the training models accuracy. Without accurate models in all simulation environments, the trained agent can hardly behave well.
- **Continuous inputs**. When we change the robot state from discrete vectors or matrices form into continuous variables, the algorithm turned to be slow of understanding the input. It may need other network architectures or other training methods.
- **Time for training**. Even though the DQfD algorithm could reduce the training time due to the reasonable behavior of agent at the beginning of training. It is still time demanding for a the simulation tools to carry out grasping procedure before sending back the results to the learning algorithm.

Chapter 7

Conclusion

In this project we have shown that depp Q-learning from Demonstrations can be used to train the robot agents learning to grasp different objects. Simulation tools such as GraspIt! or Dex-Net 2.0 could sample the positions on the surface of the objects and output the grasping qualities of these positions, which could be use to generate demonstration for pre-training agents. It has been shown that, pre-trained agents could act reasonably at the beginning of interacting with the environment and also generate good policies with fewer training episodes.

In the rest of this chapter, we discuss some of the lessons we learnt during the project and we try to list some possible directions for future work.

7.1 Lessons Learnt

- **Reinforcement learning.** Before starting this project, deep reinforcement learning is just an amazing and powerful name beating Lee Sedol by playing Go. Then I started from watching DeepMind's lectures and writing my own programs, and gradually finished this project. The most of the knowledge in this area is still waiting for me to explore, I should set a bigger ϵ for them.

- **Deep learning**. Deep learning has already become the coolest area recently. Taking advantage of different deep neural network models and comparing the different abilities from them with a realistic project made this journey enjoyable and meaningful.

- **Dual systems working**. In this project, there are three different simulation tools, a machine learning library and a networking library and there are a many more libraries they need. Some of them require Linux system while others work better in Mac OS. Therefore, from the start of this project, we tried our best to build two series of libraries on both Mac OS and Linux. It added more work but gave us more freedom to choose and a backup for the project.

- **TensorFlow**. The installation of the GPU version TensorFlow caused some

problems in the beginning, especially when we tried to install CUDA for it on a Ubuntu in a Mac Book. The installation kept sending errors and warnings until we tried all the suggested methods could be very frustrating. Also, it took some time to get used to its structure and grammar. However, in the end, it showed its magic power and help us finished our work amazingly.

- **Third party tools and their models**. We used a few different third party tools in this project. They brought us huge convenience to accomplish this project, although we need some time to get used to their structures and especially their models, since it will be to too lucky to finish the experiments with demo models.

- **Sparse matrix**. It seemed to be very natural to use sparse matrix as the input state at the beginning of the experiments. Until we tried to use continuous variables represent input state, we noticed the difference between the input data representations, and then realised the efficiency comes with sparse matrix.

7.2 Future Work

- **Higher dimensional state space**. In this project, we have simplified the task by reducing the dimension of the state space. The robot hand is moving in a two-dimensional space. In the future, we should use higher dimensional space that, the robot agent should be moving in a three-dimensional space and be able to rotate.

- **More complicate gestures**. In this project, we used one action to represent the grasping gesture, which helped us focus on the whole process of the agent moving in the space and finding good points to grasp. It is the same idea used by Dex-Net using the Baxter. However, as the robot may have more fingers aiming to more complicated missions, we may need algorithm and training structure for more gestures.

- **More complicate objects**. In the most part of this project, we focused on the cube-like objects, because we were aiming to prove the feasibility of this structure. In the future, far more complicated objects, especially those modeled form real world objects should be used to do this experiment.

- **Deeper neural network models**. If the state space, robot gestures and the objects became more complicated, it will definitely need a deeper neural network model. It will be interesting and challenging to adjust that deep network as there will be more parameters and will need more training data. We believe in the end it will be able to accomplish amazing work.

- **Training with real robots**. This project is carried out all in the simulation environments. Actually the DQfD algorithm could be used with real robot agent. We could realise this idea by using a real robot in a experiment environment to take place of MuJoCo Pro. It will be very interesting to train a real robot,

and one of the advantages of DQfD will be more obvious that, the agent will behave reasonably at the very beginning.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016. pages 28

[2] David H Ackley and Michael L Littman. Generalization and scaling in reinforcement learning. In *Advances in neural information processing systems*, pages 550–557, 1990. pages 7

[3] IA Basheer and M Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000. pages 12

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012. pages 28

[5] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013. pages 1

[6] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957. pages 9

[7] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009. pages 15

[8] Donald A Berry and Bert Fristedt. *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability)*, volume 12. Springer, 1985. pages 8

[9] Jeannette Bohg, Antonio Morales, Tamim Asfour, and Danica Kragic. Data-driven grasp synthesisa survey. *IEEE Transactions on Robotics*, 30(2):289–309, 2014. pages 1, 20

[10] Benchmark Brown. *Electronic Image Bank*. " Times Mirror Higher Eduction Group, Inc.", 1995. pages 13

[11] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC press, 2010. pages 1, 17

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. pages 29

[13] Matei Ciocarlie, Corey Goldfeder, and Peter Allen. Dimensionality reduction for hand-independent dexterous robotic grasping. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 3270–3275. IEEE, 2007. pages 27

[14] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011. pages 15

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012. pages 28

[16] Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. 2011. pages 20

[17] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999. pages 30

[18] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *arXiv preprint*, 2015. pages 20

[19] Cliff Fitzgerald. Developing baxter. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013. pages 26

[20] 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*, pages 427–434, 2006. pages 20

[21] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996. pages 15

[22] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, et al. Learning from demonstrations for real world reinforcement learning. *arXiv preprint arXiv:1704.03732*, 2017. pages 2, 18

[23] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013. pages 30

[24] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009. pages 15

[25] Keith J Holyoak. Parallel distributed processing: explorations in the microstructure of cognition. *Science*, 236:992–997, 1987. pages 12

[26] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982. pages 12

[27] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996. pages 14

[28] Narasimhan Jegadeesh and Sheridan Titman. Short-horizon return reversals and the bid-ask spread. *Journal of Financial Intermediation*, 4(2):116–132, 1995. pages 20

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. pages 28

[30] Matthew S Johannes, John D Bigelow, James M Burck, Stuart D Harshbarger, Matthew V Kozlowski, and Thomas Van Doren. An overview of the developmental process for the modular prosthetic limb. *Johns Hopkins APL Technical Digest*, 30(3):207–216, 2011. pages 26

[31] Kyle Johns and Trevor Taylor. *Professional microsoft robotics developer studio*. John Wiley & Sons, 2009. pages 26

[32] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. pages 1, 6

[33] Alexander Kasper, Zhixing Xue, and Rüdiger Dillmann. The kit object models database: An object model database for object recognition, localization and manipulation in service robotics. *The International Journal of Robotics Research*, 31(8):927–934, 2012. pages 61

[34] Jens Kober, Erhan Öztop, and Jan Peters. Reinforcement learning to adjust robot movements to new situations. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 2650, 2011. pages 20

[35] 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*, volume 3, pages 2619–2624. IEEE, 2004. pages 20

[36] GL Kovacs. A robot laboratory in real and virtual environments. In *Logistics and Industrial Informatics (LINDI), 2012 4th IEEE International Symposium on*, pages 227–232. IEEE, 2012. pages 26

[37] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013. pages 15

[38] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. pages 16, 17

[39] Honglak Lee, Peter Pham, Yan Largman, and Andrew Y Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pages 1096–1104, 2009. pages 15

[40] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015. pages 2

[41] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016. pages 20

[42] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. *arXiv preprint arXiv:1703.09312*, 2017. pages 1, 21, 22, 25

[43] Takamitsu Matsubara, Jun Morimoto, Jun Nakanishi, Masa-aki Sato, and Kenji Doya. Learning cpg-based biped locomotion with a policy gradient method. *Robotics and Autonomous Systems*, 54(11):911–920, 2006. pages 20

[44] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. pages 12, 13

[45] Olivier Michel. Cyberbotics ltd. webots: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004. pages 26

[46] Andrew T Miller and Peter K Allen. Examples of 3d grasp quality computations. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1240–1246. IEEE, 1999. pages 20

[47] Andrew T Miller and Peter K Allen. Graspit! a versatile simulator for robotic grasping. *IEEE Robotics & Automation Magazine*, 11(4):110–122, 2004. pages 24

[48] Andrew T Miller, Steffen Knoop, Henrik I Christensen, and Peter K Allen. Automatic grasp planning using shape primitives. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 1824–1829. IEEE, 2003. pages 21, 51

[49] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. pages 18

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. pages 12, 17

[51] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):14–22, 2012. pages 15

[52] ML Mynski and SA Papert. Perceptrons: An introduction to computational geometry. *MA: MIT Press, Cambridge*, 1969. pages 12

[53] Margarita Osadchy, Yann Le Cun, and Matthew L Miller. Synergistic face detection and pose estimation with energy-based models. *Journal of Machine Learning Research*, 8(May):1197–1215, 2007. pages 15

[54] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 763–768. IEEE, 2009. pages 20

[55] Yagyensh Chandra Pati, Ramin Rezaiifar, and Perinkulam Sambamurthy Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on*, pages 40–44. IEEE, 1993. pages 66

[56] Jing Peng and Ronald J Williams. Incremental multi-step q-learning. *Machine learning*, 22(1):283–290, 1996. pages 12

[57] Lerrel Pinto and Abhinav Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 3406–3413. IEEE, 2016. pages 2, 20

[58] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE, 2013. pages 26

[59] Frank Rosenblatt. Principles of neurodynamics. 1962. pages 12

[60] David E Rumelhart, John L McClelland, PDP Research Group, et al. Paralleldistributedrocessing: Exploration in the microstructure of cognition. *Foundations. ambridge, Mass.: MIT Press*, 1986. pages 13

[61] Marco Santello, Martha Flanders, and John F Soechting. Postural hand synergies for tool use. *Journal of Neuroscience*, 18(23):10105–10115, 1998. pages 52

[62] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015. pages 15

[63] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3626–3633, 2013. pages 15

[64] Kihyuk Sohn, Dae Yon Jung, Honglak Lee, and Alfred O Hero. Efficient learning of sparse, distributed, convolutional feature representations for object recognition. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2643–2650. IEEE, 2011. pages 15

[65] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. pages 6

[66] 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*, volume 3, pages 2849–2854. IEEE, 2004. pages 20

[67] Anand Thati, Arunangshu Biswas, Shubhajit Roy Chowdhury, and Tapan Kumar Sau. Breath acetone-based non-invasive detection of blood glucose levels. *International Journal on Smart Sensing & Intelligent Systems*, 8(2), 2015. pages 13

[68] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012. pages 25

[69] WT Townsend. Mcbindustrial robot feature articlebarrett hand grasper. *Industrial Robot: An International Journal*, 27(3):181–188, 2000. pages 26, 27

[70] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. pages 8

[71] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989. pages 12

[72] Paul John Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974. pages 13

[73] Walter Wohlkinger, Aitor Aldoma, Radu B Rusu, and Markus Vincze. 3dnet: Large-scale object class recognition from cad models. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5384–5391. IEEE, 2012. pages 61