Imperial College
London

MEng Individual Project
Final Report

Imperial College London

Department of Computing

Teaching Multi-fingered Robotic Hands to Grasp Simple Objects using Tactile Feedback-driven Deep Reinforcement Learning

Author:
Miklos Kepes

Supervisor:
Edward Johns

Second marker:
Stefan Leutenegger

June, 2017

Abstract

Humans' advanced object manipulation abilities play an essential part in our everyday lives and unmatched by other animals and machines as well.
Despite the fact that dexterous robot manipulation has been widely researched for decades, results have been limited by computational complexity and the challenges of high-dimensional state spaces of robot hands with many degrees of freedom (of which the human hand has 27).
Recent developments in machine learning algorithms such as deep learning and reinforcement learning introduced new possibilities in robotics and grasping research. With reinforcement learning, robots can learn tasks with a minimal amount of human intervention, motivated to execute the proposed task by a reward signal rewarding correct and penalising undesired behaviour. A recent reinforcement learning algorithm, a variant of deep Q-learning was successfully applied to learn playing and winning several Atari 2600 games, occasionally even outperforming human level [1].
Although when grasping we rely on our tactile senses as much or even more as our vision, modern research focuses less on tactile feedback and more on vision based grasping.

In this project we apply deep Q-learning to experiment with two and three finger precision grasping tasks and lifting objects relying solely on tactile feedback.
We also introduce our own improvements to deep Q-learning speeding up training times by more than 60%
We show how our agent can complete simple grasping tasks and explain why we had difficulties with more complex problems. In our evaluation we discuss our results, its strengths and weaknesses and compare it with other work in the field.

Acknowledgements

I would like to say thank you to Ed, my supervisor, for proposing this super-exciting project and for his constant guidance.
I would also like to thank my Mum and Dad for their love and support, my dearest siblings, Tomi and Mili and all my family members.
I wish to express my gratitude to my best friends Dávid and Manó for their help and valuable suggestions and to my dear friends at Imperial, too as well as my lovely flatmates.
I would also like to say thank you to my tutor, Maria for her support throughout my years at Imperial.

Contents

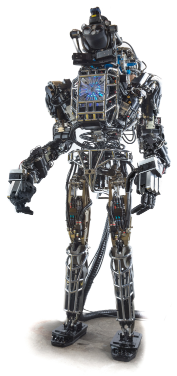
Chapter 1

Introduction

In this project we research tactile feedback-driven robotic object manipulation, grasping in particular, applying the technique of deep reinforcement learning. For our experimentation we use simulation with a five-fingered hand model and simple objects such as spheres and cubes. Unlike most existing solutions we solely rely on haptic feedback from touch sensors placed on the fingers and joint positions data for our state-representation instead of vision and vision based machine learning. We show that the Deep Q-Learning algorithm that has been very successful recently can be applied to simple grasping problems in this setup.

1.1 Motivation

Robots are becoming more and more part of our everyday life and found in all sizes and shapes performing tasks in the fields of military, medicine, agriculture, household, nanotechnology and many others. Latest applications include developing robots that clean blood vessels [2], cars that can drive completely autonomously [3], and household robots capable of cleaning the grill or mow the lawn.


(a) Boston Dynamics Humanoid robot [4]

(b) Boston Dynamics BigDog [5]

(c) Waymo self-driving car [6]

Figure 1.1: Various robots

Even in the age of self-driving cars, grasping is a major challenge in robotics and robotic object manipulation. Despite the fact that we are able to build and program robot hands that perform tasks with superhuman precision, their movements and trajectories are usually manually programmed. While they excel at that one certain sequence of movements, they cannot be applied to any other kinds of tasks failing in the case of previously unseen scenarios. These types of robotic hands are usually blind in the sense that they do not use any feedback from the environment. Robotic hands used in car factories is a good example of this.

Autonomous object manipulation, on the other-hand, aims to be more general, capable of automatically forming the necessary grasping pose and planning movements to complete a given task or given set of tasks without having the trajectory and joint manipulations pre-programmed. Some methods approach this challenge analytically, describing the pose, geometry, and movement planning with equations, other solutions attempt to learn from grasp experience data to choose the right grasp for the given problem. It is a promising field of research still in its early stages mainly due to the large state-space of robots using many degrees of freedom (the human hand has 27 of them), as well as the limitations of the algorithms. Still we see rapid development from day-to-day, and new, more efficient machine learning algorithms are proposed to overcome the above mentioned difficulties.
Humans and primates have very good grasping and fine motor skills, in high correlation with the developed brains, superior to other animals. Playing the piano, sewing, or drawing all require rather precise and well-coordinated controlling and large brain power. Researching this field, therefore, is not only useful in the applied areas but can get us closer to understanding and mimicking human intelligence and behaviour as well.

When manipulating objects, we use our vision in the first place to locate the target object, and, of course, visual feedback can also help to confirm that we hold it the way it is required . Hand-eye coordination [7], the study of coordinated movements of the hand and eye, describes how we depend on visual landmarks to plan our hand movements as well as the importance of vision in object manipulation to other uses.
However it is not only vision, that we need when actually grasping and lifting an object, but we use our very precise muscle movement capabilities or fine motor skills and kinaesthetic and tactile sensory apparatus as well. Although vision gives us constant feedback while moving our hands toward the target, we do not actually need it too much once we have located target with the eyes and memorised it. We are able to move our hands to the location and pick up objects with closed eyes and with astonishing accuracy. To get hold of the target we use our haptic intelligence which helps us sense the weight of the object, our fingers can read the texture, shape, and combine these and other properties to adjust our movements accordingly.

Figure 1.2: Lifting a sphere

Recent developments in Deep Learning [8, 9] and Deep Reinforcement Learning [1, 10] made it possible to train large, multi-layer neural networks on normal computers and learn from high-dimensional data efficiently. With these improvements deep neural networks can be used for object recognition, predicting the right grasping position, or determining the next movement of a robot hand based on the current state of its environment.

Despite the fact that the field of robotic object manipulation is well-researched, most modern machine learning based research places greater emphasis on computer vision e.g. Convolutional Neural Networks [8] for object manipulation than tactile sensory input.

The above mentioned facts motivate us to explore how tactile information i.e. touch sensor feedback complemented with position data can be used to control a software agent trained with deep reinforcement learning to plan and execute the right actions by moving the finger joints to complete simple grasping tasks. We will teach a simulated robotic hand to learn grasping and lifting small objects of different shapes and sizes.
We conduct our experiments using simulation, which makes it possible to train our software agents at an accelerated rate and even in parallel.

1.2 Objectives

The primary objectives of this project are the following:

Review relevant research Research available literature, review similar implementations, read relevant papers and research Deep Reinforcement learning and Deep Q-learning in particular, and how it can be applied to multi-finger grasping, and tactile-based grasping.

Investigate how Q-learning works with tactile-based state representations Although there is research with setups and goals similar to this project [11], we investigate how Q-learning can be applied to train a simulated, multi-fingered robotic hand to learn grasping and lifting objects of different sizes and shapes, based on tactile and position information alone, not relying on vision at all.

Compare performance and results with existing solutions Discuss how the object manipulation capabilities of our implementation compares to implementations of other techniques and algorithms.

Research different training algorithms Although in this project we focus on Q-learning, it would be interesting to see how other, more recent Deep Reinforcement Learning algorithms such as [12] and the asynchronous advantage actor-critic (A3C) algorithm [10] could be applied for this task.

Explore the possibility of applying curriculum learning Our aim is to experiment with curriculum learning, i.e. investigate whether and how the experience learnt by simpler models can be extended and used by more complex models thereby gradually building up solutions for more complex tasks.

Transfer model to real-life robot-hand We also consider experimenting with the option of applying our model trained with simulation to real-life grasping tasks using a real-life robot arm. Even if this is not possible, the simulator trained model can be used as an initialised model to a real-life training setup.

1.3 Challenges

Open-ended Because of the numerous types of grasping tasks and setups, the large number of possible experiments makes it hard to decide on what the focus should be.

State-of-the-art results are still at an early stage compared to human grasping Even though multi-fingered robotic manipulation is a popular area of research, due to the curse of dimensionality training large degrees-of-freedom (DOF) hand models is still quite limited.

Sensitivity to initial setup and hyperparameters There are a large number of hyperparameters and settings for the deep reinforcement learning framework and the simulation which can hugely influence and determine the results. Although the goal is to make the models as general as possible, finding hyperparameters is still one of the major challenges of modern machine learning algorithms.

Initial finger positions and object placement are also crucial to increasing the chances of getting good results.

Hardware and time requirements Despite becoming significantly more efficient in recent years data-driven machine learning algorithms still take quite long to train, from a few hours to sometimes more than a day in the case of this project. Generally the best solution for this problem is to implement distributed versions of these algorithms which is challenging, especially when the computational resources are limited.

1.4 Contributions

Object manipulation with Q-learning and haptic feedback In this project we show (Chapter 6) that it is possible to train a software agent to learn simple object manipulation tasks such as grasping and lifting using deep Q-learning with input states containing only tactile and joint position data.

Curriculum learning with Deep Reinforcement Learning Our experiments prove that applying a simple form of curriculum learning to gradually increase complexity is a viable option for tasks like this.

Optimising Q-learning with separate ϵ per joint We present a minor alteration of Q-learning that – similarly to curriculum learning – can significantly speed up training time (see Section 6.15).

Q-learning with simultaneous actions We also introduce another small change to the Q-learning algorithm in order to be able to move all fingers simultaneously. Although this modification has not improved the learning speed significantly, it is a viable and useful option for controlling the joints in the fingers.

1.5 Structure of the report

Our report breaks down to six chapters. Chapter 2 describes the background for the experimentations, including deep reinforcement learning and the relevant algorithms, object manipulation techniques and grasp taxonomy. In Chapter 3 we present and explain our design choices, e.g. software and the applied algorithms. Chapter 4 specifies the setup for our experimentation, including a 3D hand and object models used for grasping and the Prioritised Q-learning algorithm applied to this specific task. In Chapter 5 we introduce the technical details of our program that runs the experiments. The experimentation and the results are detailed in Chapter 6, Chapter 7 gives an evaluation of the project while Chapter 8 consists of our conclusions and ideas about future work.

Chapter 2

Background

In this section we describe the background that is necessary to understand how to apply deep reinforcement learning to solve the grasping task by a simulated robotic hand.
First we discuss different grasping techniques and place our technique into context. We also present a grasp taxonomy so we can refer back to it later when talking about different types of grasps.
Then we provide a short summary of deep learning (Section 2.2) and reinforcement learning (Section 2.3) and then of how the two technologies are combined in deep reinforcement learning (Section 2.4).
We briefly discuss curriculum leaning in Section 2.6.
Finally, we list a short summary of research papers related to this project and used for, or referenced in the implementation.

2.1 Robotic Grasping

In this section we detail the history of, and different approaches to, robotic grasping throughout the last decades. Robotic (multi-finger) grasping [13] and dexterous object manipulation are popular and well-researched areas with a wide range of options to address their challenges.

What is a grasp? Shimoga [14] defines a grasp as a system whereby a desired object is gripped by the fingers of a robot or human hand. Sometimes grasp refers to the manipulated grasp as well.

Shimoga [14] also describes four mutually independent properties that should be possessed by the grasp: dexterity, equilibrium, stability and some kind of dynamic behaviour to be able to grasp and there are different algorithms focusing on achieving each.
There are two main approaches to grasping problems, namely, analytic and data-driven. Bohg et al. [15] provide an overview of data-driven grasp synthesis [1], while also presenting a summary of analytical approaches and the relationship of the two. A more general, but thorough, analytical overview of robotic manipulation can be found in in the book by Murray et al. [16].

[1]"grasp synthesis refers to the problem of finding a grasp configuration that satisfies a set of criteria relevant to the grasping task" [15]

Analytic approaches In the case of analytic methods [17, 18], grasp synthesis is defined as a constrained optimisation problem requiring geometric, kinematic and/or dynamics formulations. [19, 15].

To achieve successful grasping a large number of constraints need to be satisfied, hence the complexity of analytic approaches. However most algorithms only focus on either the hand model or the task constrains and referred to as force-closure [20] or task-oriented [21, 22]. There are numerous solutions within these categories, for instance a example of analytic approach described in [23] is based on a fast force optimisation algorithm with the Lagrange Multiplier Method.

Errors in the models of the robot's kinematics and dynamics and noise in the sensor values can make analytic models inaccurate causing it more difficult to complete a grasp.

Analytical methods dominated until the 2000-s, however with the raise of Modern Artificial Intelligence data driven approaches became more widely applied.

Data-driven approaches Data-driven or empirical approaches, on the other hand, avoid complex mathematical and physical formulations and rely on sampling grasp experiences from a database generated by human demonstration or other methods and ranking them according to some empirical metrics. They use perceptual information such as feature extraction, semantics, classification, and others to solve a given grasping task. They can develop a grasping technique by learning and generalising from data or by using some heuristic, shown in Figure 2.1. The database used for learning is usually generated by human demonstration, trial and error, or assumed to be a labelled data set.

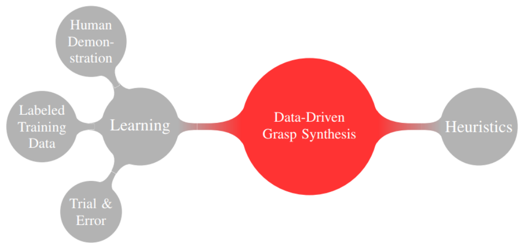


Figure 2.1: Data driven grasp synthesis [15, Fig. 2.]

Data-driven methods are further categorised by Bohg et al. [15] to problems with known objects, familiar object and unknown objects requiring different approaches.

To summaries, the main advantages of data-driven solutions are less computational complexity, fewer assumptions, and less fragility, while it usually requires large amount of data.

Hybrid approaches It is not rare to use both analytic and data-driven methods together, for instance in [24] a hybrid methods is presented for task-specific dexterous in-hand object manipulation.

In the last 15 years data-driven approaches gained more and more ground. Recent algorithms begin to involve deep learning (Section 2.2), for instance for predicting successful grasps, object recognition and pose estimation [25, 26, 27].

Although there has been plenty of examples of grasping algorithms based on vision [28], tactile information [29] or both [30], a recent variant of deep neural networks, Convolutional neural networks [8], became the primary choice for many vision-based data-driven solutions. For example in [31] the authors use Convolutional Neural Nets to estimate the ideal grasping pose for a given task.

Another machine learning technique, reinforcement learning, which is widely used in robotics, working well with a large range of robotic tasks, gained popularity for grasping tasks as well. For instance it is often used together with human demonstration [32]. With reinforcement learning we can define tasks by implementing a reward function rewarding the behaviour we want to see and penalising unwanted actions.

Deep learning can be combined with reinforcement learning [25], [1]. Deep neural networks can handle well the large state-space of a robot hand with many degrees of freedom, and can be used to predict the next action to take by a robot to successfully complete a task. To teach the network what actions are to be taken in what states of the hand, reinforcement learning is an ideal choice.

2.1.1 Grasp taxonomy

There have been several attempts to classify human grasping types into categories from various fields, however, probably the most consistent and complete taxonomy published in 2015 [33] under the title ”The GRASP Taxonomy” and – as the authors write – it is based on the GRASP project funded by the European Commission. We will refer to this taxonomy when we mention different types of grasps throughout this report.

In the paper they differentiate 33 different types of grasps along the following dimensions:

Opposition type Oppositions occur between hand surfaces and are further categorised based on their direction.

- Pad opposition: along a direction parallel to the palm
- Palm opposition: along a direction perpendicular to the palm
- Side opposition: along a direction transverse to the palm

For illustration see the second-level columns of Figure 2.2

The virtual finger assignment The virtual finger (VF) refers to a subset of fingers when they work together as a functional unit, applying similar forces to achieve a certain task. See third-level columns of Figure 2.2

Power, precision, and intermediate grasp Categorising grasps based on how much power or precision they utilise. See the main columns of Figure 2.2

The position of the thumb Either abducted or adducted. See rows of Figure 2.2

The authors also mention the increasing importance of haptic feedback in human-computer interaction.

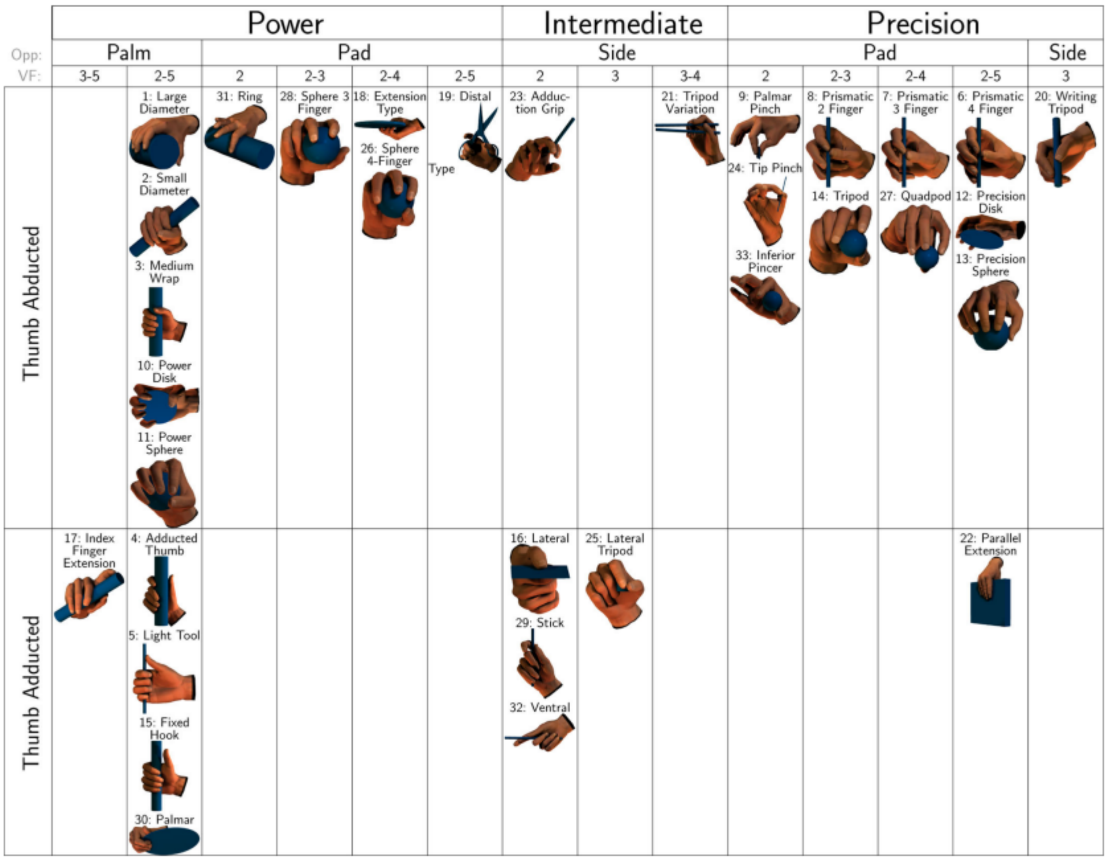


Figure 2.2: GRASP taxonomy [33, Fig. 4.]

There are only 17 cells and thus distinguished grasp types on Figure 2.2. This is because the main difference between grasps within the same cells are due to the shape of the manipulated object and not the grasping technique.

2.2 Deep Learning

Deep Learning [34] is a machine learning technique that uses multi-layer artificial neural networks capable of finding patterns in large datasets by approximating the underlying structure of the data. The popular training algorithm, backpropagation (2.2.5), can iteratively adjust the parameters (often referred to as weights) of the network based on the difference between the expected and actual results, until that difference is minimal.
There are different neural network architectures and algorithms, making it possible to be trained in a supervised or unsupervised way, resulting in predictive and generative models. Deep Neural Networks and its variants, such as Convolutional Neural Networks (CNN-s) [8] – used for image input – General Adversarial Networks (GAN-s) [35] – for unsupervised learning – and Long short-term memory (LSTM) [36] – a recurrent variant, capable of remembering and forgetting input for various time periods – are widely used for classification and other problems such as voice, face and image recognition tasks, image generation from text [37] and many others.

When we talk about deep learning we refer to a certain type of artificial neural network architecture that has multiple so-called hidden layers and the algorithms used to train it for a given task, but let us see what neural networks really are!

2.2.1 Neurons and Neural Networks

The idea of artificial neural networks is inspired by biological neural networks of the brain, which can be found in every mammal, avian and Cephalopoda. Both biological neural networks and artificial neural networks consist of a large number of neurons and the connections between them. The working principle behind the two, however, is substantially different. In the case of biological neural networks electric current flows through neurons. Neurons receive current through their dendrites (there could be 10,000 of them for a single neuron) and their firing potential in the nucleus increasing with the total amount of current flowing in. Once it reaches a threshold potential the neuron fires sending its current through its axon and the synapses – the points connecting the axon with other neurons' dendrites and communicating with them through complex chemical reactions. There are approximately 100 billion neurons in a human brain which can be categorised into groups based on their functionality (or their connections). Groups communicate with each other through some kind of synchronised oscillation [38]. Researchers do not quite agree on the way the information is encoded in the brain. There are multiple possibilities such as rate coding, temporal coding and population coding, further discussion, however, is out of the scope of this project.

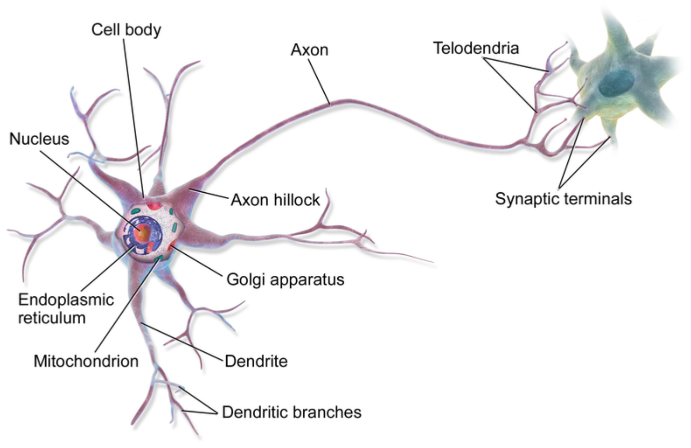


Figure 2.3: Neuron, with its axon, dendrites, nucleus and synaptic terminals annotated [39]

2.2.2 Artificial Neurons

The model of artificial neurons and artificial neural networks (ANN) is not nearly as complex, and actually quite different from its biological counterparts. The building blocks of ANN-s are artificial neurons.
An artificial neuron is a mathematical unit which takes the weighted sum of its inputs, applies a so called activation function to it and the neuron 'fires' if this 'activation' is larger than a given threshold value. For instance

$$\text{output} = \begin{cases} \text{fire} & \text{if } \boldsymbol{w}^\top \boldsymbol{x} > t \\ \text{do not fire} & \text{otherwise} \end{cases} \tag{2.1}$$

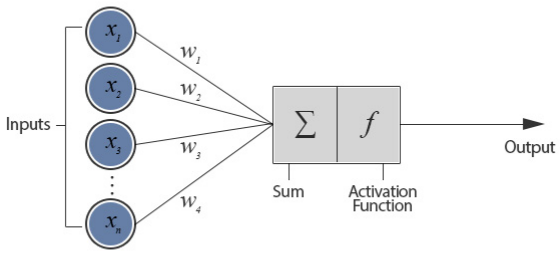


Figure 2.4: Artificial neuron [40]

Where

- $\boldsymbol{x}$ is the vector of input data
- $\boldsymbol{w}$ is the vector of weights
- t is the threshold value

This is the model of a binary neuron, since the output is either 'fire' or 'do not fire'. In practice, the threshold is usually represented by a negative bias value i.e. $t = -b$ and therefore we can write

$$\text{output} = \begin{cases} \text{fire} & \text{if } b + \boldsymbol{w}^\top \boldsymbol{x} > 0 \\ \text{do not fire} & \text{otherwise} \end{cases} \tag{2.2}$$

Also, instead of having a binary output, it proved useful just to apply a certain activation function to $b + \boldsymbol{w}^\top \boldsymbol{x}$

$$h(\boldsymbol{x}) = g(a(\boldsymbol{x})) = g(b + \boldsymbol{w}^\top \boldsymbol{x}) \tag{2.3}$$

Where

- h represents the neuron as a function
- g is the activation function (detailed later) and its output is the activation of the neuron (resembling biological neurons' membrane potential)
- and $a(\boldsymbol{x}) = b + \boldsymbol{w}^\top \boldsymbol{x}$ is called the pre-activation

Activation functions The usual choices for value functions are generally one of the following:

Linear A linear neuron or perceptron [41] simply takes the sum of its inputs, adds the bias value and then outputs the resulting number.

$$g(a) = a \tag{2.4}$$

Sigmoid One of the most popular activation functions, squashing its input to be between 0 and 1.

$$g(a) = \frac{1}{1 + \exp(-a)} \tag{2.5}$$

where $\exp(x) = e^{-x}$

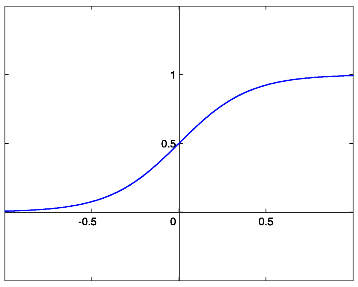


Figure 2.5: Sigmoid function [42]

Hyperbolic tangent (Tanh) This function also has a bounded range between -1 and 1

$$g(a) = \frac{\exp(2a) - 1}{\exp(2a) + 1} \tag{2.6}$$

Rectified linear (ReLU) A simple, yet widely used activation function is the rectified linear unit, it is only bounded from below, by 0.

$$g(a) = \max(0, a) \tag{2.7}$$

What can artificial neurons do? Artificial neurons can be used for simple decision making, also known as classification. The dimensions of the input vector represent different aspects of a problem or question, the weight parameters weight these aspects in accordance with their importance and the output is a yes or no decision or a probability, however the user decides to interpret it. A single neuron can classify problems where the data is linearly separable, while organising multiple neurons in a network makes it possible to model more complex underlying functions.

2.2.3 Artificial Neural Networks

One successful structure of neural network architecture is one whereby neurons are organised into layers with inter-layer connections only, without any intra-layer connections. That is, there are no connections between neurons within the same layer. Also, neighbouring layers are usually fully connected, there is a connection between every two neurons of two neighbouring layers. This kinds of neural nets are called feed-forward networks.

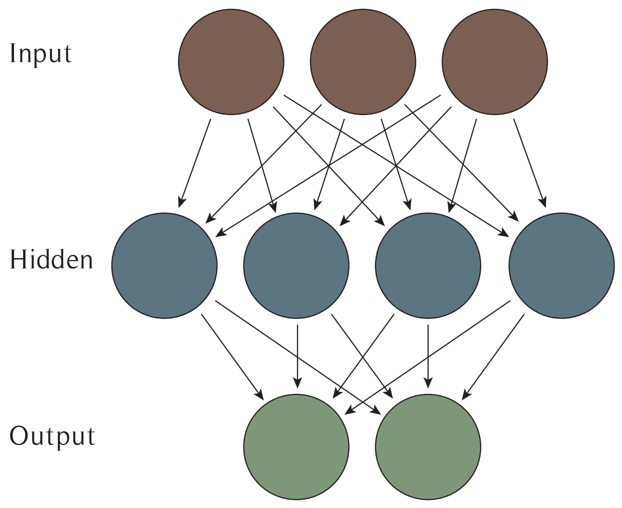


Figure 2.6: Feed-forward neural network with single hidden layer.

Feed-forward Neural Network computation In multi-layer networks the activations of the previous layers constitute the input for the hidden and output-layers. Intermediate layers are called hidden layers because we do not 'see' their inputs and outputs. Input is fed to the input layer, with one neuron for each dimension of the input. It is then propagated through the network by computing the activation of each neuron of each layer, which is then the input of the next layer. Note that layers can have different activation functions within a single network.
For classification tasks the number of output neurons is usually the number of classes which the input is supposed to be categorised into. The output neuron with the highest activation represents the predicted category. Output activities can be normalised across all output neurons as well, so that they add up to one, thus one can interpret the output as a probability distribution.

2.2.4 Representation capabilities of Neural Networks

It is known that a single neuron can classify linearly separable problems and neural networks with a single hidden layer are capable of modelling any continuous function, while multi-layer networks are powerful enough to represent any function by distorting the input space so that it becomes linearly separable [34]. The weights of neural networks represent features and in multi-layer networks, the nearer the layer is to the output, the higher dimensional features it represents. That is how we arrive from a small curve in the left corner of an image to identifying a face, taking face recognition as an example.

2.2.5 Training Neural Networks

Above, we saw the simplified description of how neural networks compute their output values. However, the main challenge in neural network programming is finding the proper weight parameters of the connections. The algorithms used to do this are called training algorithms. When we train a neural network in a supervised way, we feed it a large set of training examples consisting of input output pairs. During training we measure the difference between the network's actual output and the expected output from the supervision signal. This measurement is referred to as loss, error or cost function and our goal is to minimise it. The smaller the error, the better the network has learnt the features of the training set.

Loss functions During training we optimise the loss functions (also called cost, error, or objective function), representing the error between the predicted and actual values of the network. There are several different types of them, usually chosen based on the type of the problem.
For training with backpropagation we require that the loss is written as an average:

$$l = \frac{1}{n}\sum_x l_x \tag{2.8}$$

Some of the widely used loss functions are:

- The mean squared error (MSE) function

$$\begin{aligned} l(f(\boldsymbol{x}), \boldsymbol{y}) &= \frac{1}{2}\sum_j (y_j - f(x_j))^2 && (2.9) \\ &= \left\| \boldsymbol{y} - f(\boldsymbol{x}) \right\|_2^2 && (2.10) \end{aligned}$$

  where

  - $f(x)$ is the prediction of the network
  - and y is the target

- The cross-entropy function

$$l(f(\boldsymbol{x}), \boldsymbol{y}) = -\sum_j [y_j \ln f(x_j) + (1 - y_j)\ln(1 - f(x_j))] \tag{2.11}$$

Gradient descent A popular way to minimise the loss function is using the first-order optimisation method called gradient descent. Gradient descent works by taking the derivative of the error/loss function at an initial point in the parameter space, then seeking out the direction of the steepest descent to the next point, and adjust the weights along this direction repeating this until the error gradually reaches a local minimum.
Below we see the equations for a single gradient descent update:

$$\Delta = -\nabla_\theta l(f(x^{(t)}; \theta), y^{(t)}) - \lambda\Omega(\theta) \tag{2.12}$$

$$\theta = \theta + \alpha\Delta \tag{2.13}$$

Where

- l is the loss function
- θ is the matrix of neural network weights
- α is the learning rate, a small constant
- $-\lambda\Omega(\theta)$ is a regularisation factor, called momentum, ensures that gradient descent does not stuck in small local minima.

Backpropagation As long as the loss function is not sufficiently small we need to keep adjusting the weights. For networks with hidden layers the backpropagation algorithm is used for this task. The idea is to update weights of the neurons in proportion with the amount they contribute to the final error. The error or loss is computed for the output layer first and by repeatedly applying the chain rule the error function gradients are propagated backwards in the network, with respect to which the gradients of the weights can be computed and the weights updated.
For a more detailed description of backpropagation see [34]

2.3 Reinforcement Learning

In this section we briefly describe reinforcement learning, and the background for Deep Q-learning. This section is mostly based on the book by Sutton and Barto [43] and on the Reinforcement course [44].

2.3.1 Description

Reinforcement Learning (RL) is a sequential decision making, machine learning technique different from both unsupervised and supervised learning. The concept of reinforcement learning comes from behavioural psychology, referencing the study of human and animal learning that is controlled by rewarding right behaviour and penalising unwanted behaviour.

RL focuses on how it is possible to learn from interacting with the environment. If we assume an initially unknown environment then we have to explore it and gain experience in order to improve our knowledge of this world, our interaction with it and thus of our survival chances. Throughout this exploration we obtain good and bad experiences (or anything in between) and naturally we would like to relive the good ones and learn how to achieve them, while we want to avoid the bad ones. When things get more complex, we might find that some good experiences can only be obtained by first experiencing some bad ones. In such cases we have to decide whether the that good experience is worth enough to suffer through the bad ones.

To formulate the reinforcement problem we thus define a (software) agent which interacts with an environment. This interaction is modelled as a sequence of observations and actions in time. At every time-step the environment is described by its current state which is observed by the agent, assuming the environment is observable and the state of the environment is equivalent to the state of the agent.

Along with every state the agent also receives a reward from the environment representing the 'goodness' of this new state. The more reward an agent gets the more good experience it has. It is no surprise then, that the goal of the reinforcement learning task it to maximise the accumulated overall reward.
Figure 2.7 depicts how the agent interacts with the environment. It repeatedly observes the environment by receiving a state $s \in S$, and a scalar reward r at every time-step t, $0 \leq t \leq T$. It then decides on the next action $a \in A$ to take. The action can change the environment, altering its state which is then observed by the agent again. Here S and A are the set of all possible states and all possible actions respectively and T is the length of the given time sequence. To choose actions, the agent uses a policy π, which maps states to actions, or

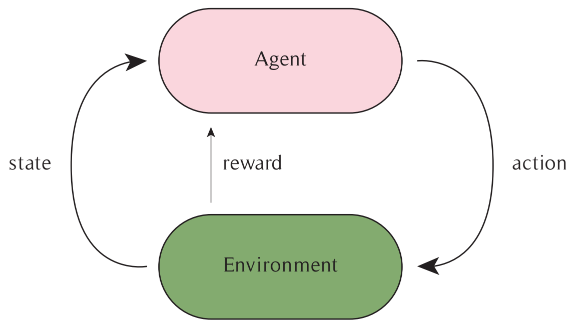


Figure 2.7: Simple model of agent-environment interaction.

action probability distributions, and defines the agent's behaviour.

RL is neither supervised nor unsupervised like most machine learning algorithms, because it is not trained with labelled data as in supervised learning, it has to find the best solutions by discovering through trial and error, but unlike unsupervised learning it does not have to find the features in the training data without any clue in an unsupervised manner, instead it is driven by a reward signal.

This model is extremely flexible and there are many different algorithms and implementations for various RL tasks. In order to understand more about them we first need to introduce a few definitions.

2.3.2 Rewards and returns

Interacting with the environment the agent gains experience of the form:

$$s_0, r_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, ... \tag{2.14}$$

If it is possible to break down the task into separate sequences of episodes, it is called an *episodic task* while a single sequence of interactions for $t, 0 \leq t \leq T$ is called an *episode*.

$$s_0, r_0, a_0, s_1, r_1, a_1, ..., s_t, r_t, a_t, ..., s_T, r_T, a_T \tag{2.15}$$

We can sum r_t for all t to get the total reward for a given episode, however in practice we define the *long-term* or *cumulated reward* or *return* as the *total discounted reward* from time-step t:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma r_{t+3}^2 + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.16}$$

Where $\gamma \in [0, 1]$ is the *discount factor*, a hyperparameter of the RL task, it ensures that future rewards are worth less than immediate rewards, preventing infinite loops and encouraging short and efficient solutions. It also resembles human and animal behaviour preferring immediate rewards.

Rewards help the agent to adapt its strategy suggesting actions that will maximise the long-term reward R. The fact that sometimes a complex sequence of actions is needed to obtain a reward (delayed rewards) makes reinforcement learning quite challenging.

2.3.3 Formalising RL problems as Markov Decision Processes

If the reinforcement Learning task satisfies the Markov Property (see 2.17) and the environment is fully observable then it can be formulated as a Markov Decision Process (MDP), a decision making model.
Markov Property: the probability of state in time-step $t+1$ given the state of time-step t equals to the probability of state at $t+a$ given all states from 1 to t

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_1,\ldots,s_t) \tag{2.17}$$

that is, given the previous state, all preceding states are irrelevant or, in other words, given the present we do not care about the past. The present state contains all the necessary information from the history of events, so the history can be forgotten.

MDP-s are ideal to formally describe an environment for reinforcement learning and consist of the following components (assuming finite MDP-s):

- States $s \in S$, where S is a finite set of states
  describing the state of the environment in time-step t
- Actions $a \in A$, where A is a finite set of actions
- Transition probabilities $P^a_{ss'} = p(s_{t+1} \mid s_t, a_t)$
  Probability of transition from state s to state s' when the agent takes action a
- Discount factor $\gamma \in [0,1]$
- Reward function $R^a_{ss'} = r(s,a,s')$, or $r_{t+1} = r(s_{t+1}, s_t, a_t)$
  Reward received when transitioning from state s to state s' following action a

The reinforcement learning task can be either

model-based Cases where the *MDP* is known, also called *planning*. An MDP in known if we know its transition probabilities $P^a_{ss'}$. An agent can improve its policy without interacting with the environment.

model-free Cases without a known, or too large *MDP*. This is the more frequent scenario, environment is initially unknown, the agent improves its policy by interacting with the environment.

2.3.4 Policies

A policy, representing the agent's behaviour, maps states to actions or action probability distributions (i. e. the probabilities for each action with which it should be followed).

That is a policy π can be

Stochastic $\mathbf{a} \sim p_\pi(\mathbf{a} \mid \mathbf{s})$
Returns the probability with which the given action should be executed in the given state, or

Deterministic $\mathbf{a} = \pi(\mathbf{s})$
Simply tells the next action for a given state

Policies can be

Evaluated Usually referred to as **Prediction**. It is about evaluating a given policy.

Optimised Usually referred to as **Control**. It is about finding the best policy.

Generally, the goal of the reinforcement learning task is to find an optimal policy π^* that helps the agent to choose the actions which will maximise its cumulative reward.

Not all types of reinforcement learnings are policy-based, however, there are policy-free value-based algorithms as well.

2.3.5 Value function and Action Value function

Given a policy, we can estimate the values of states and state-actions pairs. This is one of the main components of reinforcement learning.

We define the *value function* or *state-value function* for policy π as the expected future reward in a given state telling the 'goodness'/'badness' of that state. The formula is:

$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^t r_{t+k+1} \mid S_t = s] \tag{2.18}$$

where R_t is the discounted total return and r_{t+a} is the immediate reward.

Furthermore, let us also define the *state-action* or *state-action value* function, which tells us how good it is to be in a given state and take a given action following a policy π.

$$Q^\pi(s,a) = \mathbb{E}[R_t \mid S_t = s, A_t = a] = \mathbb{E}[\sum_{k=1}^{\infty} \gamma^t r_{t+k+1} \mid S_t = s, A_t = a] \tag{2.19}$$

which is the expected return when in state s, taking action a and following policy π.

There are multiple types of algorithms to compute V^π and Q^π, some of them estimates V^π and $Q\pi$ based on experience, others compute the real expected values assuming the MDP model is known.

Bellman equations and self-consistency Many of the algorithms exploit a fundamental property of the state and action-value functions, so that both of them can be decomposed into two components: immediate reward and discounted value or action-value of the successor state:

$$V^\pi(S) = \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s] \tag{2.20}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[r_{t+1} + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{2.21}$$

To see that this holds we have (for the state-value function):

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t \mid s_t = s] \tag{2.22}$$

$$= \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s] \tag{2.23}$$

$$= \mathbb{E}_{\pi}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s] \tag{2.24}$$

Then, applying the expectation we get:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \Big[\mathcal{R}^{a}_{ss'} + \gamma \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1=s'}]\Big] \tag{2.25}$$

$$= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \Big[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')\Big] \tag{2.26}$$

Also,

$$V^{\pi}(s) = \sum_{a} \pi(s,a) Q^{\pi}(s,a) \tag{2.27}$$

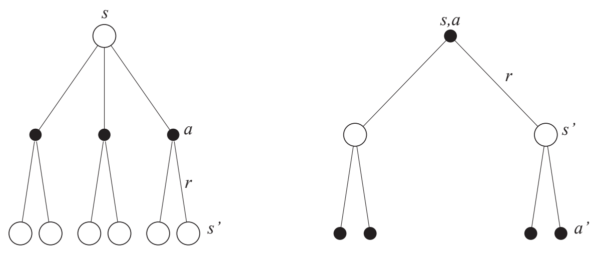


Figure 2.8: Backup diagram for V^{π} and Q^{π}.
Circles represent states, shaded nodes actions, and arcs rewards

We can use backup diagrams such as on Figure 2.8 to trace the paths from states or state-action pairs to the successor states. It represents the most important step of reinforcement, that of learning to transfer information back from successor states to the current state. This operation is also called update.

In matrix form:

$$\boldsymbol{v}_{\pi} = \boldsymbol{R}^{\pi} + \gamma \boldsymbol{P}^{\pi} \boldsymbol{v}_{\pi} \tag{2.28}$$

and it can be solved directly:

$$\boldsymbol{v}_{\pi} = (\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi})^{-1} \boldsymbol{R}^{\pi} \tag{2.29}$$

The Bellman equation is linear and self-consistent, therefore in theory it could be solved directly. However, due to the large state space of most reinforcement learning problems, in practice we usually rely on iterative algorithms to find the solution. We classify these algorithms into three methodologies:

- Dynamic programming
- Monte-Carlo evaluation
- Temporal Difference learning

Optimal value and action value functions Now let us also define the optimal state-value and optimal action-value functions as the maximum value and action-value functions over all policies

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.30}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.31}$$

We say that the MDP is solved when we have found the optimal value function. So how do we find the optimal policy achieving an optimal value function?
One option is finding the optimal state-action values $q^*(s,a)$ and then we can have a policy

$$\pi^*(a \mid s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in A} q^*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.32}$$

Bellman optimality equations

$$V^*(s) = \max_{a} \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^*(s')) \tag{2.33}$$

And the optimal state-action function:

$$Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | S_t = s, A_t = a] \tag{2.34}$$

$$= \sum_{s} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')) \tag{2.35}$$

Unlike the Bellman Equations, the Bellman Optimality Equation is non-linear and there is no general closed form solution for it, so here we need iterative solutions as well, such as

- Value Iteration
- Policy Iteration
- Q-learning
- Sarsa

2.3.6 Planning and Dynamic programming

Let us first examine the scenario when the MDP in perfectly known and there is no need for interacting with the environment. In this case we can use a certain type of algorithm, Dynamic Programming (DP), for evaluating and finding optimal policies. Although DP algorithms are computationally expensive and the perfect MDP model assumption also limits the applicability of this methods, it is essential to understand it in order to understand other RL methods.

Dynamic programming is an algorithm for solving complex problems by breaking them down into simpler subproblems. It is only applicable if the problem satisfies the following two properties:

- Optimal substructure: optimal solution can be decomposed into subproblems. In other words we can find a solution by repeatedly solving subproblems and using these solutions to get an overall result.
- Overlapping subproblems: sub-solutions can be reused. That is, besides breaking down the original problem to sub-problems, the solutions can be reused by multiple sub-problems higher up the recursion tree.

MDP-s satisfy these properties and therefore we can use DP for both

Prediction Find value function V^{π} for a given MDP and policy π

Control Find optimal value function V^{*} and optimal policy π^{*} for a given MDP

Let us see the DP algorithms for prediction and control assuming a known MDP.

Policy Evaluation This algorithm evaluates a given policy for a given MDP by repeatedly applying the Bellman expectation equation (2.26) until V converges to V^{π} that is $V \approx V^{\pi}$. Iteratively updating V this way is guaranteed to converge, for proofs see [43].

Algorithm 1 Iterative policy evaluation [43, Ch. 4]

```
Input: $\pi$, the policy to be evaluated
1: Initialise an array $v(s) = 0$, for all $s \in S^{+}$
2: repeat
3:     $\Delta \leftarrow 0$
4:     for each $s \in S$ do
5:         $temp \leftarrow v(s)$
6:         $v(s) \leftarrow \sum_{a} \pi(a \mid s) \sum_{s'} p(s' \mid s, a)[r(s, a, s') + \gamma v(s')]$
7:         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
8:     end for
9: until $\Delta < \theta$                    ▹ A small positive number
Output: $v \approx v^{\pi}$
```

Policy Iteration Now, what happens when instead of evaluating an optimal policy we have to find one? One solution is to use the policy iteration method. The idea is to repeatedly evaluate and improve, then again evaluate and improve the policy until convergence to optimal policy π^{*}. We saw above how to evaluate a policy, now we need a method to improve it. Such is the greedy policy which always takes the action that puts the agent in the neighbouring state with highest state-value.

General Policy Iteration It is a good strategy to decompose RL tasks into two components: policy evaluation and policy improvement (learning and control). This has the benefit that we can use any two independent algorithms for evaluation and improvement.

- Policy evaluation: estimate V^π
- Policy improvement Generate $\pi' \geq \pi$

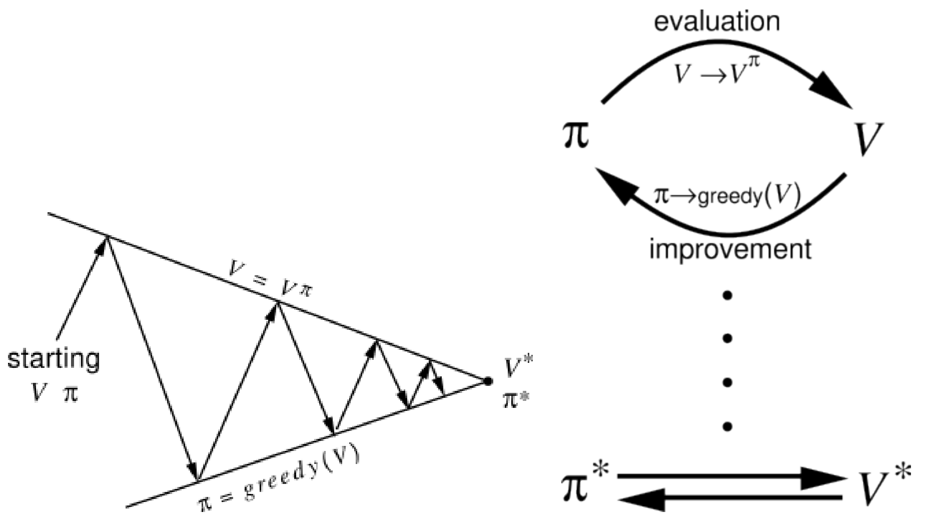


Figure 2.9: Value and policy functions interact until they are optimal and thus consistent with each other [43]

Value Iteration The idea behind value iteration, another method to find an optimal policy π^* given an MDP, is iteratively applying the Bellman Optimality Equation (2.33).

Algorithm 2 Value iteration [43, Ch. 4]

```
1: Initialise array $v$ arbitrary
2: repeat
3:     $\Delta \leftarrow 0$
4:     for each $s \in S$ do
5:         $temp \leftarrow v(s)$
6:         $v(s) \leftarrow \max_a \sum_{s'} p(s' \mid s,a)[r(s,a,s') + \gamma v(s')]$
7:         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
8:     end for
9: until $\Delta < \theta$                                    ▷ A small positive number
Output: A deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s'} p(s' \mid s,a)[r(s,a,s') + \gamma v(s')]$
```

Value iteration is policy free, an intermediate value function does not correspond to any policy. It combines policy evaluation and improving in each of its iterations.

Since dynamic programming assumes that we know the full model of MDP it is not suitable for most real-life problems. Model-free algorithms can learn by interacting with the environment and estimate the state-value or action-value functions from experience.

2.3.7 Model-free learning

In this section we will describe model-free reinforcement learning methods, which can estimate the value function of unknown MDP-s as well.

Monte-Carlo methods Monte Carlo (MC) methods learn directly from episodes of experience by defining the value as the empirical mean return of complete episodes. There are different MC algorithms such as

- First-Visit: we compute the empirical mean for every state starting from the first-time it is visited in the given episode.
- Every-Visit: we compute the empirical mean for every state starting from every occurrence of that state in the given episode
- Incremental: We update the value function after each episode in the following way:

  $$v(S_t) = V(S_t) + \alpha(R_t - V(S_t)) \tag{2.36}$$

  where α determines the rate of forgetting past episodes. R_t: actual return

Temporal-Difference learning TD methods learn directly from episodes of experience as well, however unlike MC methods they learn from incomplete episodes by bootstrapping (i.e. they update their value estimates based on other value estimates).

$$V(S_t) = V(S_t) + \alpha(r_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.37}$$

where

- $r_{t+1} + \gamma V(S_{t+1})$ is referred to as estimated return, or TD target, and
- $r_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called TD error

TD is an online algorithm, it can learn from experience after every step, MC on the other hand has to wait until the end of each episode.

2.3.8 Model-free control

In model-free control we optimise the value function of a given MDP. It is useful when either the MDP model is unknown, but experience can be sampled, or MDP model is known, but it is too large to use and it makes more sense to take samples from it. Model-free control algorithms can be classified into two categories:

- **On-policy**: methods, which attempt to evaluate or improve the policy that is used to make decisions
- **Off-policy**: methods, that evaluate or improve a policy different from that used to generate the data

Policy improvement

- If have a greedy policy over $V(s)$ (a policy that always chooses the action with maximum expected reward) then we require to know the model of the MDP:

$$\pi'(s) = \arg\max_{a \in A}(R_s^a + P_{ss'}^a V(s')) \tag{2.38}$$

- A greedy policy over $Q(s,a)$ however is model-free:

$$\pi'(s) = \arg\max_{a \in A} Q(s,a) \tag{2.39}$$

Therefore model-free methods usually work with the state-action value function instead of the state value function. If the state-action values are known,

ϵ-greedy Exploration A popular policy is the ϵ-greedy policy where we greedily select the maximum action with a high probability but there is also a small ϵ probability, equally for every other action, to be chosen:

$$\pi(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a^* = \arg\max_a Q(s,a) \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases} \tag{2.40}$$

Where $\epsilon \in [0,1]$ and $|A(s)|$ is the number of available actions in state s.

The ϵ-greedy policy keeps exploring new actions while following a greedy policy most of the time. This way it ensures that all actions are selected from time-to-time, which is essential for convergence.

Q-Learning Q-learning [45] is an off-policy TD control algorithm that is guaranteed to converge to the optimal Q function Q* independently of the policy being followed. In Q-learning to find the optimal Q values we update the state action-value function iteratively the following way:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.41}$$

Where α is the learning rate, it determines how quickly old episodes should be forgotten.

The pseudo-code for Q-learning:

Algorithm 3 Q-learning [43, Ch. 6.]

```
1: Initialise $Q(s,a)$ arbitrarily
2: repeat                                                  ▷ (for each episode)
3:     Initialise $s$
4:     Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
5:     Take action $a$, observe $r$, $s'$
6:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
7:     $s \leftarrow s'$
8: until $s$ is terminal
```

Algorithm 4 shows the DQN algorithm as it appears in [43].

Each step of an episode consists of an experience of the form (s, a, r, s'), and an episode contains a sequence of such tuples, representing the steps starting from an initial state up to a terminal state.
Since Q-learning is an off-policy algorithm it has two policies: a behavioural and a target policy. During policy improvement, given we are in state s we choose action a with the behavioural policy which is greedy w.r.t. $Q(s, a)$, for instance $\epsilon - greedy$. The target policy – the one being – improved is also greedy w.r.t. $Q(s, a)$, as we can see above: $\pi(S_{t+1}) = argmaxQ(s_{t+1}, a)$. This way the algorithm learns about the greedy strategy (target policy) while following the behavioural policy that ensures exploration.

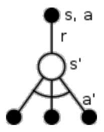


Figure 2.10: Backup diagram for Q-learning [43] representing how the state-action value is updated based on the next state and best action from that state

2.4 Deep Reinforcement Learning

For simple problems it is possible that we store all the state-values or state-action values in a table, and compute them one-by-one. However, for more complex problems we have to face the curse of dimensionality, the number of states can be far too large to process with today's computer platforms. For instance in the case of a robot hand with five fingers, each with three joints, even if we assume that each joint takes only 90 discrete values instead of the continuous space, we would have $(90^3)^5 = 90^{15} \approx 2 * 10^{29}$, and we have not even taken into account that the hand itself needs to be moving. One solution to overcome this problem is using linear or sometimes non-linear function approximation methods such as linear features, decision trees, nearest neighbours, or deep neural networks (DNN). Artificial Neural Networks (Section 2.2) are capable of generalising from training data to handle unseen data. A neural network can be used to predict the state action values of unseen state action pairs, if it is first trained on many examples.

For instance in the case of Q-learning we could train a network (a Deep Q-network) as a function approximator that maps state-action pairs to their values (Q-values).

2.4.1 Stochastic gradient descent

In order to understand deep Q-learning described in the following sections, we need one more concept to explain. In Section 2.2.5 we presented gradient descent, a method to optimise the loss function of a neural network.

Now let us define the loss function for a given reinforcement learning problem as

$$l(\boldsymbol{\theta}) = \mathbb{E}_{\pi}[(V_{\pi}(s) - \hat{V}(s, \boldsymbol{\theta}))^2] \tag{2.42}$$

where

- $\boldsymbol{\theta}$ is the neural network's weight matrix
- $V_\pi(s)$ is our true state value function
- $\hat{V}(s,\theta)$ is the estimate state value function by the neural network

and its gradient ∇l as

$$\nabla l(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_\pi[(V_\pi(s) - \hat{V}(s,\boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}}\hat{V}(s,\boldsymbol{\theta})] \tag{2.43}$$

The goal is to find a parameter vector $\boldsymbol{\theta}$ that minimises the mean-squared error between the approximate value function $\hat{V}(s,\boldsymbol{w})$ and true value function $V^\pi(s)$
With gradient descent we adjust θ in the direction of the steepest descent

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha E_\pi[(V_\pi(s) - \hat{V}(s,\boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}}\hat{V}(s,\boldsymbol{\theta})] \tag{2.44}$$

where α is the step-size parameter or learning rate.
Note that we ignored the $\frac{1}{2}$ as it is irrelevant to the gradient update.

Stochastic gradient descent on the other hand samples the gradient instead of doing a full update:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha(V^\pi(s) - \hat{V}(s,\boldsymbol{\theta}))\nabla_{\boldsymbol{\theta}}\hat{V}(s,\boldsymbol{\theta}) \tag{2.45}$$

which is – on the long run – equivalent to gradient descent with full updates. The advantage of this solution is that the parameters can be updated without computing the expected value.

2.4.2 Deep Q-Learning with Experience replay

This variant of Q-learning is called Experience replay [46] and a popular choice for tasks with huge state space. A more recent variant was introduced in [1] by Google DeepMind [47], where it was used to learn and play Atari 2600 games with superhuman efficiency. The authors have shown that the algorithm is general enough to play six of the games with the same set of hyperparameters (except for a few minor exceptions) and architecture.

In standard Q-learning we update the neural network with targets computed from the actual (s,a,r,s') tuples in every step. This has multiple disadvantages, for example it breaks the assumption that the input data samples are independent and identically distributed, which is usually the assumption for deep learning algorithms. Also the data distribution is not constant as it should be, but transforms as the agent develops, not ideal for deep learning either. These issues can be somewhat mitigated by introducing a so called replay memory, that stores (s,a,r,s') tuples as they are incoming, instead of learning from them immediately. The memory has a fix size (e.g. $1,000,000$) and once it is full, and new transitions keep being added the oldest ones get removed. From time to time (for instance every fourth step) the agent samples from this memory, choosing experiences randomly with uniform probability, and uses this batch of samples of a predefined size (usually 32 or 64) to update the weights of the network. This way experiences can be reused multiple times during training, reducing the number of examples that need to be generated. Furthermore, the high correlation between consecutive transitions is no longer a problem since experiences get randomly chosen from a larger pool, reducing the variance of the updates.

Schaul et al. [48] mention furthermore, that experience replay has some biological motivations as well. Neuroscientists observed evidence of sequences of prior experiences being replayed in the hippocampus of rodents during their awake, resting and sleeping periods.

Summary of steps [44]:

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random minibatch of transitions from D
- Compute Q-learning targets w.r.t. old, fixed parameters θ_{i-1}
- Optimise MSE between Q-network and Q-learning targets
$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s'\ D_i}[(r + \underset{a'}{max} Q(s', a', \theta_{i-1}) - Q(s, a, w_i))^2] \tag{2.46}$$
- Using stochastic variant of gradient descent

Algorithm 4 Deep Q-learning with Experience Replay [1, Alg. 1.]

```
 1: Initialise replay memory $\mathcal{D}$ to capacity $N$
 2: Initialise action-value function $\mathcal{Q}$ with random weights
 3: for episode = $1, M$ do
 4:     Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 5:     for $t = 1, T$ do
 6:         With probability $\epsilon$ select a random action $a_t$
 7:         otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 8:         Execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$
 9:         Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10:         Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
11:         Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$
12:         Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal} \phi_{j+1} \end{cases}$
13:         Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
14:     end for
15: end for
```

Algorithm 4 shows the DQN algorithm as it appears in [1].

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot);s'\sim\epsilon}\Big[\Big(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\Big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big] \tag{2.47}$$

2.5 Improving DQN

Although Deep Q-learning with Experience Replay proved to be an algorithm capable of playing Atari 2600 games better than humans in some cases, it does have a few drawbacks such as rather long training times, and also it suffers from certain instability issues. In the following sections we will discuss a few methods to improve DQN.

2.5.1 Using a target network

In the DQN algorithm the targets or updates for the network were computed as

$$Q(s,a) \to r + \gamma \max_a Q(s',a)$$

However, it was shown in [49, 50] that introducing a second, so called target network can make the training more stable. This target network is used to generate the target Q-values, and it is the copy of the original network, except for the fact that its weights are fixed for a certain number of steps and only updated from time to time, copying the weights of the other network. The reason this improves the training is because when we use only a single network the estimated target will be constantly shifting at every training step, since $Q(s',a)$ and $Q(s,a)$ are very close to each other. The update period of the training network should be large enough to allow the original network to converge.

$$Q(s,a) \to r + \gamma \max_a Q_{target}(s',a)$$

.

2.5.2 Double Q-Learning

According to the authors of a paper introducing a variant of Q-learning called Double Q-learning [50], the deep Q-learning algorithm presented in the 2013 paper [1] tends to overestimate action values in some cases. In their paper they show how this can influence the results in a negative way and present a solution to the problem.
In addition to using a separate target network they propose a further adjustment. They argue that the overestimation happens because of the $\max$ operator in

$$Q(s,a) \to r + \gamma \max_a Q(s',a) \tag{2.48}$$

If Q(s', a) is maximal for a certain a, but this is because the value is noisy and not precise, the error will be propagated and influence other states. The authors say that regardless whether the reason of the error in the estimation is because of environmental noise, function approximation, non stationary or other origin, it introduces an upward bias. These overestimations would not cause a problem if they were uniform, since then all states would be overestimated equally. However, the authors show that this is not the case and suggest to overcome this by using separate Q-functions for selection and evaluation. Luckily, we already have two networks so we can utilise the target network and alter the updates in the following way:

$$Q(s,a) \to r + \gamma Q_{target}(s',a) \arg\max_a Q(s',a)) \tag{2.49}$$

We choose the action with maximal Q-value estimated by the original network, then compute the Q-value of this action with the target network, which estimation then is applied in the training step, moving the $Q(s,a)$ towards $r + \gamma Q_{target}(s',a) \arg\max_a Q(s',a))$.
The paper also states that the decoupling of the Q-functions selecting and evaluating the action in the target is not completely independent, since we periodically copy the weights of the original network to the target network. This solution is still sufficient to mitigate the problem of overestimation, without introducing additional computational complexity.

2.5.3 Prioritised Experience Replay

As described in [1], the standard DQN algorithm samples from the replay memory uniformly, not taking into account how much can be learnt from the given sampled experiences. The authors mention that this could be possibly improved by applying a more intelligent sampling method.
In a 2016 paper, an improved version of experience replay was presented by Schaul et al. [48], where experiences were sampled with their significance taken into consideration. This is useful because the agent cannot learn from all the experiences with the same efficiency, some are more important, still in the original DQN algorithm they are replayed the same number of times as any other transition. The writers describe a stochastic prioritisation technique determining which experiences to replay more frequently. It can improve learning tasks where receiving positive rewards has a rather low chance and is often preceded by a large number of negative or zero reward steps.
They say that the ideal criterion by which the importance of an experience could be measured would be the amount that the agent can learn form a transition in its current state, and suggest this can be approximated by giving priorities to transitions using the magnitude of their temporal-difference (TD) error $|\delta|$, which shows how 'surprising' a certain transition is. For Double DQN (DDQN) the TD error is defined as

$$|\delta| = |Q(s,a) - [r + \gamma Q_{target}(s',a) \arg\max_a Q(s',a))]| \tag{2.50}$$

One problem is that if transitions are sampled purely in proportion with their TD error, then the ones with initially low errors will be likely never to be sampled, even if their TD error would increase if they were replayed, furthermore approximation noise can also bias the priorities and results. The proposed solution to this is to ensure that even low-priority transitions are sampled at least once by interpolating between uniform random sampling and greedy prioritisation. Thus the sampling probability of a transition, introduced by the authors is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.51}$$

where $p_i > 0$ is the priority of transition i, and α influences the amount of prioritisation used, that is, if $\alpha = 0$ we have uniform sampling.
In the paper there are two variants for determining p_i, proportional and rank based, here we will only present the former one, since they were shown to produce similar results (even though the rank based approach is more robust in theory).
Proportional prioritisation defines the priority of a transition, p_i as

$$p_i = |\delta_i| + \epsilon \tag{2.52}$$

where ϵ is a small positive value, so that the transition can be sampled even when its δ is zero.
As described in the paper, one efficient implementation featuring proportional prioritisation could utilise the 'sum-tree' data-structure (a tree with every parent node's value being the sum of the values of its children), where priorities are in the leaf nodes, supporting efficient sampling and updates.

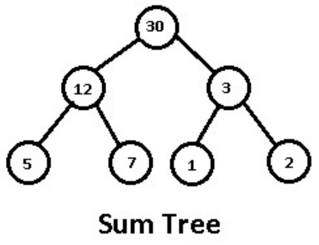


Figure 2.11: SumTree data-structure [51]

Importance Sampling Using priority sampling introduces a bias. Quoting from the paper: *"The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed)."*
The suggested solution for this problem is using (weighted) importance sampling (IS) weights.

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta} \tag{2.53}$$

In practise it is common to anneal β from a β_0 to 1 as the training proceeds, with β reaching 1 by the end of the training, since unbiased weights are most important for convergence towards the end of the training.

Algorithm 5 Double DQN with proportional prioritization [48, Alg. 1.]

```
Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget T.
 1: Initialise replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
 2: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
 3: for $t = 1$ to $T$ do
 4:     Observe $S_t$, $R_t$, $\gamma_t$
 5:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
 6:     if $t \equiv 0 \mod K$ then
 7:         for $j = 1$ to $k$ do
 8:             Sample transition $j \sim P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$
 9:             Compute importance-sampling weight $w_j = \frac{(N \cdot P(j))^{-\beta}}{\max_i w_i}$
10:             Compute TD-error $\delta_j = R_j + \gamma_j Q_{target}(S_j, \arg\max_a Q(S_{j-1}, a)) - Q(S_{j-1}, A_{j-1})$
11:             Update transition priority $p_j \leftarrow |\delta_j|$
12:             Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
13:         end for
14:         Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
15:         From time to time copy weights into target network $\theta_{target} \leftarrow \theta$
16:     end if
17:     Choose action $A_t \sim \pi_\theta(S_t)$
18: end for
```

Algorithm 5 shows the Double DQN algorithm as it appears in [48].

2.6 Curriculum Learning

Curriculum learning [52, 53] is another machine learning technique inspired by biology – human and animal learning, to be more precise. Bengio et al. [53] argue that humans and animals can learn much more efficiently if the training material is organised in a meaningful way, has a good structure and introduces complexity gradually. They also how this could be used in machine learning. They then introduce curriculum learning as a global optimisation strategy for non-convex functions, showing that in some cases this technique can find better local minima, and even converge to the global minimum faster than other methods. They compare curriculum learning to the unsupervised initialisations or pre-training of deep neural networks [54] which – although no longer necessary nowadays – is a similar way of 'starting small' in order to find better local minima.

In a more recent paper [55] we see a variant of reinforcement learning where curriculum learning is actually part of the parameters of the learning problem. Agents generally are trained for a certain task specified by a single reward function, however Held et al. [55] argue that if we want agents to be capable of learning more complex tasks – for example lifting objects and taking them to the other end of the room – we need a more flexible way of specifying goals. Their proposition is to use Goal Generative Adversarial Networks [56] (Goal GAN) to generate goals to achieve in the current state of the environment and feed these goals as part of the input state vector to the agent. The goal is to maximise the average success rate of the agent over all possible goals. This way goals get gradually more difficult as the agent learns as part of this automatic curriculum generating technique.

2.7 Summary of related work

Playing Atari with Deep Reinforcement Learning [1] In this paper by Google DeepMind [47] the authors describe how they managed to build the first deep learning model that could learn control policies directly from high-dimensional sensory input using reinforcement learning. They implemented a framework that could successfully learn how to play different Atari 2600 games on a human level. They used Deep Q-Learning with convolutional neural networks, and trained the system with the Experience replay algorithm [46] that is used in this project as well.

Deep Reinforcement Learning with Double Q-learning [50] An extension to DQN is introduced in this paper to overcome a drawback of the original Q-learning algorithm (Algorithm 3) namely that it tends to overestimate Q-values in some cases. As a solution a second Q-network is introduced which is used to compute the targets and updated with the weight of the other Q-network only from time to time. In this paper we can see how this solution mitigates the problem of biased Q-values.

Prioritised Experience Replay [48] The experience replay algorithm [46, 1] samples from the replay memory with uniform probability even though not all experiences are equally valuable. Prioritised experience replay replays those experiences more often that have larger TD-errors and therefore more important to learn from for the Q-network.

Curriculum learning [53] This paper introduces experiments with curriculum learning and deep learning to show that it is possible to learn more complex problems when the training data is organised in a meaningful structure and presented to the learning algorithm gradually with increasing difficulty.

Automatic Goal Generating for Reinforcement Learning Agents [55] A recent paper showing how reward functions can be generated automatically by Goal Generative Adversarial Networks and used as a parameter of the current state in order to learn more complex tasks with reinforcement learning within a single training.

In-Hand Robotic Manipulation via Deep Reinforcement Learning [11] The authors of this paper use the same technique, Deep-Q Networks, for teaching in-hand manipulation to a simulated robotic hand, or Modular Prosthetic Limb (MPL). Both the task and the techniques used are very similar to the task of this project, the main difference is that like in [1] the Deep-Q network used image data as its input unlike our solution, which relies on position and touch sensor data.
This paper is quite brief, but it provides a very good overview of the problem and the architecture of the framework that solves this challenge.

Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates [12] This paper features some other variants of Deep Reinforcement Learning algorithms that are also based on off-policy training of deep Q-functions, namely Deep Deterministic Policy Gradient and Normalised Advantage Function algorithms. They show that these methods, unlike previous solutions, can be efficiently used without compromising the autonomy of the learning process for robotic manipulation both in simulation and in real life. They successfully teach real life robots to open doors.

Learning Dexterous Manipulation Policies from Experience and Imitation [57] In this work, besides deep neural networks, some other methods such as nearest neighbours, imitation are explored for object manipulation tasks both in simulation and in real life situations. It is more relevant to this project in the sense that they also rely on tactile and proprioceptive feedback when training and using the deep neural network.

Associating Grasping with Convolutional Neural Network Features [58]] This paper describes a method of pre-shaping for a human like robotic-hand using convolutional neural networks. Although this is quite different from the main objective of this project, since it relies on visual information, while in this project we work with position data and haptic feedback, the two technologies complement each other in a very nice way, and the technique it uses could be interesting for the extensions part of this work.

Learning to Grasp Everyday Objects using Reinforcement-Learning with Automatic Value Cut-Off [59] This is an older paper from 2007, and while they don't use deep reinforcement learning (they keep track of Q-values dynamically), they do use a variant of Q-learning. Their technical descriptions (for instance for the reward function, actions, evaluation) could also be helpful for this project.

Towards Learning Hierarchical Skills for Multi-Phase Manipulation Tasks [60] The authors of this paper show how the manipulation task can be decomposed into a sequence of phases. They use different methods for training the robot for the different phases, so they combine learning from human-demonstration with reinforcement learning. Their work could be useful if we will experiment with curriculum learning.

Learning Robot In-Hand Manipulation with Tactile Features [29] In this paper reinforcement learning is applied to teach a robotic hand the repositioning of objects in the robot's hand, and they use tactile feedback to achieve this. Although the objective is quite different from this project's objective, it provides an interesting example of object manipulation based on tactile/haptic sensing.

Chapter 3

Design choices

In this section we present our design decisions and briefly explain the reasons behind using the software and tools we use.

3.1 Simulation

Simulations, although aim to mimic real-life, often require different strategies and algorithms and have several advantages and disadvantages over real-life experiments.

Pros

- Simulations can run at an accelerated speed
- They can handle errors well, errors and accidents do not result in large expenses. For instance teaching a robot to walk on two legs with reinforcement learning is more risky in real life, since the robot can break because of repeated failures.
- Also, resetting the environment is easy to do with simulations, e.g. the manipulated object can be placed back to its original place automatically, unlike in real life.
- Multiple simulations can be run simultaneously, resulting in even faster learning. This – of course – is also possible in real-life but usually is more expensive.
- Finally it is possible to access and use the simulator any time of the day, which is usually not the case with a real-life robot.

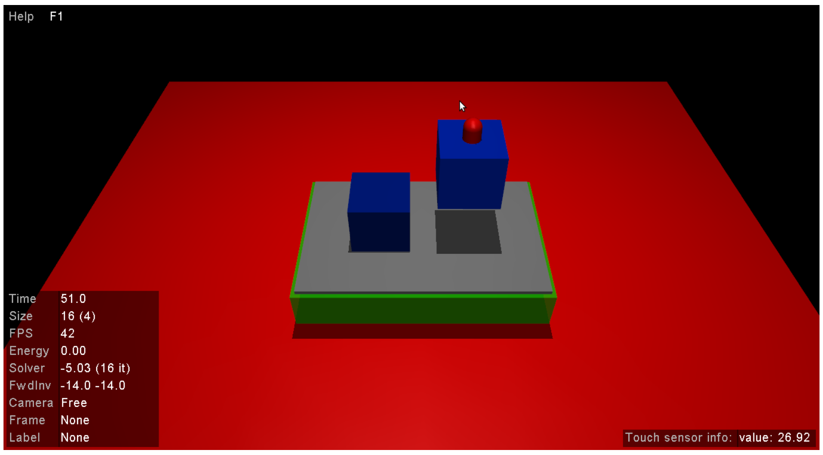
Cons

- Even the best simulations are simplified, cannot take all real-life like constraints into account. To speed up simulations we often have to sacrifice even more of the accuracy.
- Following from the previous point, it is not guaranteed that a model working in a simulation would work in real life.

In this project we conducted experiments relying exclusively on simulation because of the limited time range, easier accessibility and other advantages of simulation.

3.1.1 MuJoCo

MuJoCo (**Mu**lti-**Jo**int dynamics with **Co**ntact) [61] is an advanced physics engine with a simulator API widely used in research. MuJoCo was developed by Emo Todorov from Roboti LLC and it outperforms many other simulators, especially in the fields of robotics and grasping simulation. Models for MuJoCo can be implemented in XML format. The user can define bodies, give them different geometric shapes and specify the connections or joints between them. Unlike game engines, in MuJoCo two bodies have 0 degrees of freedom between them until we add joints manually (optionally ball, slide, hinge or free-shaped). We can also attach different kinds of sensors onto objects. The model can be controlled via a C++API, where we are allowed to specify the position/velocity/acceleration of the joints directly, or define actuators.



```
Time     51.0
Size     16 (4)
FPS      42
Energy   0.00
Solver   -5.03 (16 it)
FwdInv   -14.0 -14.0
Camera   Free
Frame    None
Label    None
```

Touch sensor info: value: 26.92

Figure 3.1: MuJoCo interface and interacting bodies

[62] presents a performance comparison of MuJoCo and other physics engines such as PhysX [63], ODE [64], DART [65], Bullet [66] and Havok [67] showing that MuJoCo is the fastest among these for high DOF (35) grasping simulation.

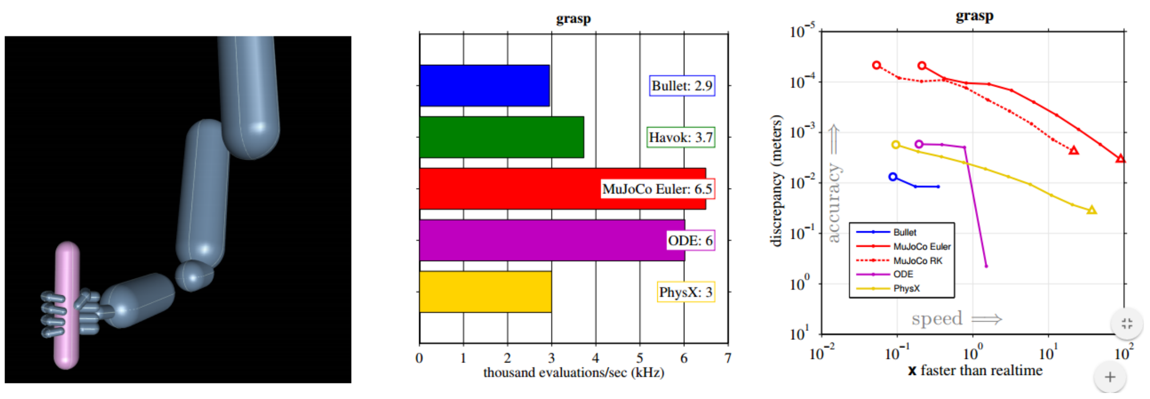


Figure 3.2: MuJoCo's performance compared to alternatives for grasping. Second figure shows the raw speed of the engine, and the third one the speed-accuracy trade-off [62, Fig. 3.]

Although MuJoCo is a relatively new software it is widely adopted and used by software companies like Google DeepMind [47]. It has a wide range of settings and has a good online documentation [68], however, MuJoCo being a new simulator intended for professional use by researchers, there are no tutorials on the internet (with the exception of its official online forum) that can offer fast solutions for all range of problems (like, for instance, in the case of Gazebo [69]).

3.2 Programming languages used

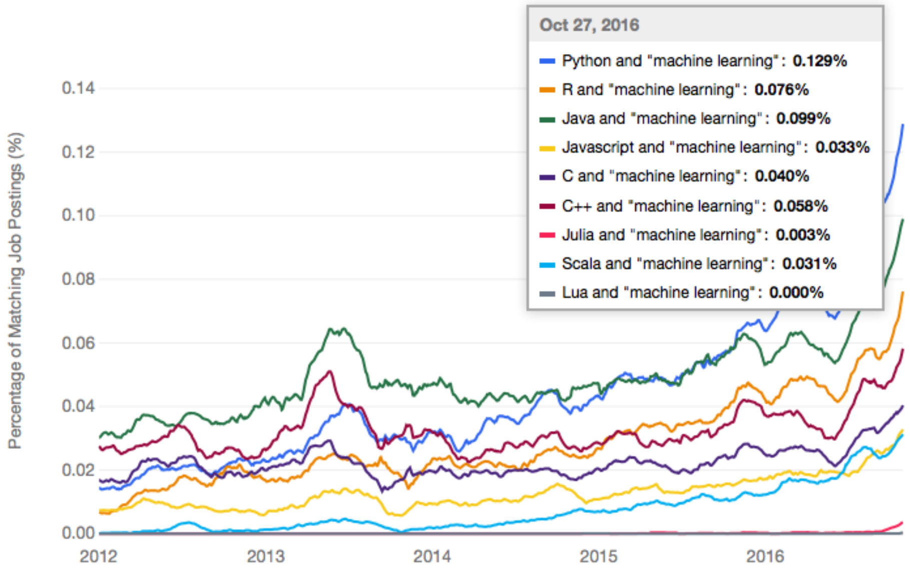


Figure 3.3: Most popular languages for machine learning job postings from 2016 [70]

Python 2.7 We decided to implement the deep reinforcement framework learning in Python [71] as Python is one of the most popular languages amongst data scientists and machine learning researchers. It has extensive library support, with wide range of tools that can be useful for machine learning problems, for instance NumPy [72] and Keras (3.3).

C++ As MuJoCo's API is written in C/C++, the simulation/environment side of the reinforcement learning framework was also developed in C++.

XML Also, as mentioned earlier (3.1.1), MuJoCo models are generally specified in XML, with the robothand models we use from MuJoCo's website being no exceptions.

3.3 Deep Learning libraries

TensorFlow TensorFlow [73] is an open source machine learning library originally developed by the Google Brain team of Google in 2015. It uses data-flow graphs to efficiently execute numerical computations. Its flexible graph-based API makes it possible to use on all kinds of devices, desktop or server environments, laptops and smartphones, devices with CPU or GPU. It is generally used for neural network computations with a large range of options.

CNTK CNTK [74] or The Microsoft Cognitive Toolkit is a deep-learning toolkit by Microsoft. Its first production release was in June 2017 and it is known to be performing generally much faster than its alternatives. Microsoft promises improvements both in speed and accuracy compared to Tensorflow. In its Version 2.0 release Keras support was added as well.

Keras Keras [75] is an open source, simplified and quite easy-to-use, high-level API (or wrapper) for TensorFlow written in Python. It can be used to build deep neural networks in only a few lines of code. It is designed to be minimal and intuitive, the user can put together the building blocks of a complex network without having to worry about the complicated backend implementation. For most applications with simple neural networks Keras is quite sufficient and provides all the necessary functionalities. We decided to use Keras since the deep neural networks we have are quite simple (no special tweaking, like customised loss functions were needed), furthermore, in this project we focus on reinforcement learning and not on deep learning.

3.4 Deep Reinforcement Learning Algorithms

DDQN with Prioritised Experience Replay The deep reinforcement learning algorithm primary used in this project will be DDQN with Prioritised Experience Replay described here (Algorithm 5). This is one of the most up-to-date (published in November, 2015), and efficient function approximation based methods for reinforcement learning with huge input state space. It is widely used for similar areas of research.

Asynchronous Advantage Actor Critic (A3C) Although DDQN with PER is indeed one of the best RL methods, the most state-of-the-art, policy gradient based algorithm was published by Google DeepMind in 2016 described in this paper [10]. It has slightly better performance than DDQN with PER, and the algorithm itself is less complex. It, however, requires to run multiple agents simultaneously. We do not discuss A3C further in this report.

This project focuses on deep Q-learning, however, we implemented A3C as well but unfortunately had no time for testing it.

Chapter 4

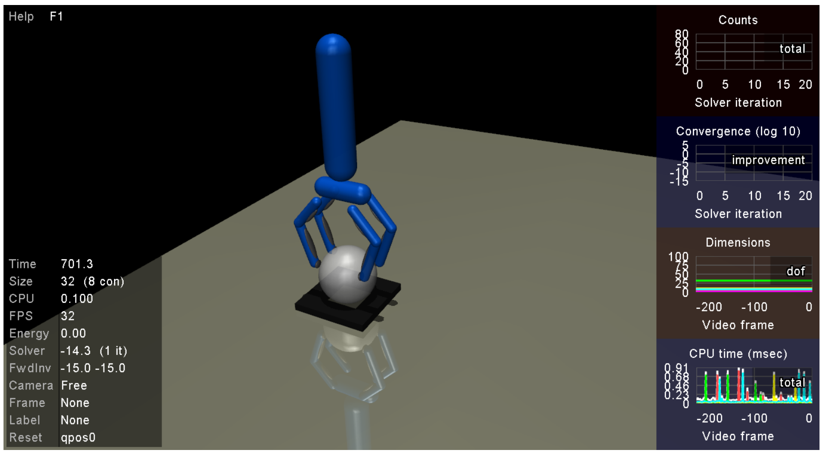
Setting up the environment for experimentation

In order to apply deep reinforcement learning on the problem of grasping we first need to build an environment that can accommodate our experiments.
In this section we present the robothand models we used for the experimentation together with their physical properties, as well as the models of the manipulated objects. Moreover, we will describe how deep reinforcement learning can be applied to this specific problem by showing how we implemented the components of the reinforcement learning algorithm.
Finally, we will present the hardware we used for the experimentation and compare typical running times.

4.1 Models

4.1.1 Gripper

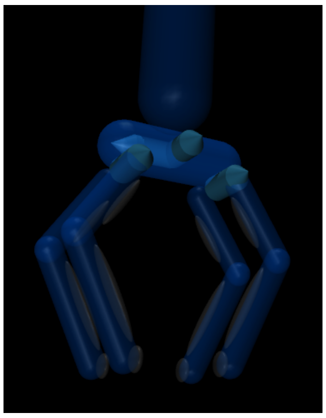
Throughout the experiments we used two different kinds of robot hand models, a gripper and a five-finger hand. Although we focus on multi-finger grasping we use the gripper model to test how our environment works, and we also conduct some simple lifting experiments with it in Section 6.5.

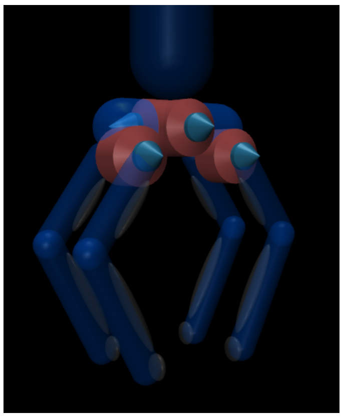


```
Time     701.3
Size     32  (8 con)
CPU      0.100
FPS      32
Energy   0.00
Solver   -14.3  (1 it)
FwdInv   -15.0 -15.0
Camera   Free
Frame    None
Label    None
Reset    qpos0
```

Figure 4.1: Four finger gripper

Figure 4.1 shows our four-finger gripper. It has two fingers on both left and right sides of its wrist. The fingers on both sides are coupled, therefore they move together controlled by the same joint and are totally interdependent.


(a) Joints (light blue)


(b) Joints (light blue) and actuators (red)

Components This model consists of four parts: the arm, the wrist attached to the arm, two claws attached to the wrist with both claws composed of two fingers.

Joints and DOF There are two hinge joints between the wrist and the forearm, *flexion/extension* and *abduction/adduction*, and two further hinge joints where the claws are fixed to the wrist. See Figure 4.2a
Furthermore, the arm can be moved in all three dimensions of the space. Therefore the model has four joints + three additional degrees of freedom, seven degrees of freedom altogether.

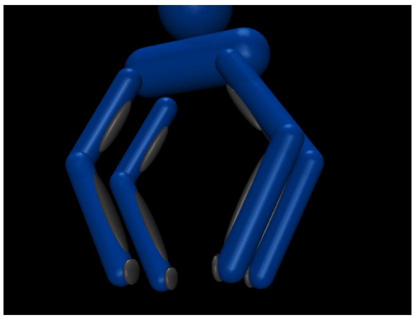
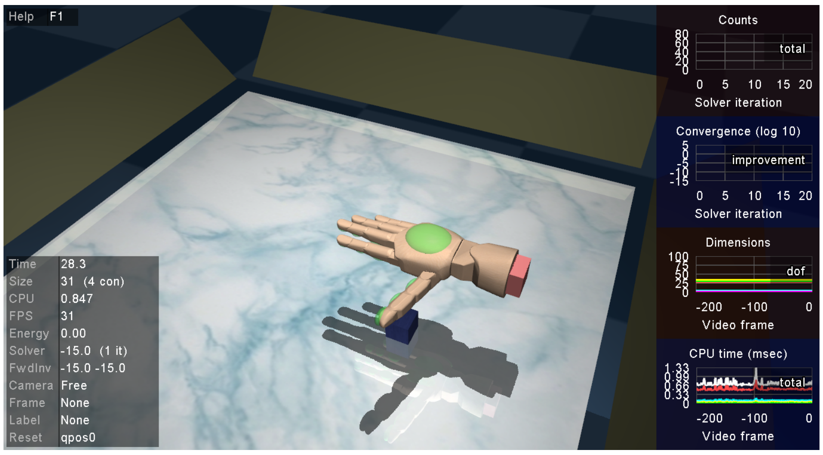


Figure 4.3: Gripper with touch sensors (grey)

Sensors Two touch sensors are placed on each of the fingers, one at the tip of the finger and one on the inner side of the lower part of the claws. (There are two additional touch sensors on the claws' upper part, however they remained unused), see Figure 4.3 Each joint has a one position sensor as well to report its positions (or angle) as part of the state vector which is then used by the agent.

Controls In order to move the joints we use position actuators, one for each, determining the angle of the given joint, see Figure 4.2b.

4.1.2 Hand



```
Time     28.3
Size     31  (4 con)
CPU      0.847
FPS      31
Energy   0.00
Solver   -15.0  (1 it)
FwdInv   -15.0 -15.0
Camera   Free
Frame    None
Label    None
Reset    qpos0
```

Figure 4.4: Hand

The other model is a five-finger hand resembling a real human hand, see Figure 4.4. This model, called Modular Prosthetic Limb, originally designed by the John Hopkins Applied Physics Laboratory [76], was downloaded from the resources page of the MuJoCo website [57].

Components The model is composed of the following parts: forearm, wrist, palm and five fingers (thumb, index, middle, ring, and pinky).

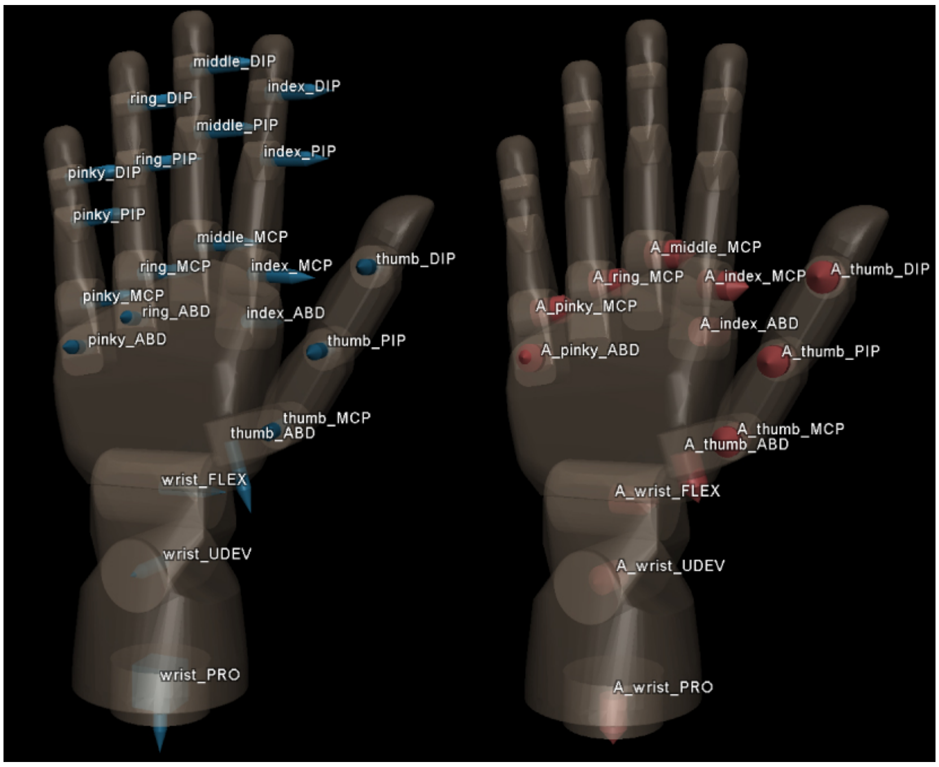


Figure 4.5: MPL with joints and actuators annotated [68, chap. 6]

Joints and DOF Table 4.1 shows the number and list of joints that can be found in each component.

Name	Number	Joints
Wrist	3	pronation/supination, ulnar/radial deviation, flexion/ extension
Thumb	4	abduction, MCP, PIP, DIP
Index finger	4	abduction, MCP, PIP, DIP
Middle finger	3	MCP, PIP, DIP
Ring finger	4	abduction, MCP, PIP, DIP
Pinky	4	abduction, MCP, PIP, DIP

Table 4.1: List of joints

where

- **metacarpophalangeal joint (MCP)**: the joint at the base of the finger
- **proximal interphalangeal joint (PIP)**: the joint in the middle of the finger
- **distal interphalangeal joint (DIP)**: the joint closest to the fingertip [77]

Note: in our MPL model we have separate abduction and MCP joints instead of a single MCP joint.

Therefore our model has 19 finger joints, and 3 joints in the wrist, moreover – just as in the case of the gripper model – this arm can be moved in all three dimensions freely, as well, therefore it has 25 degrees of freedom.

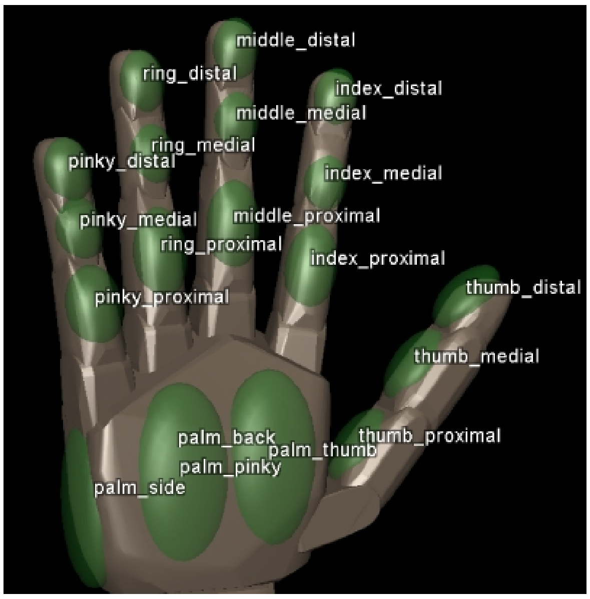


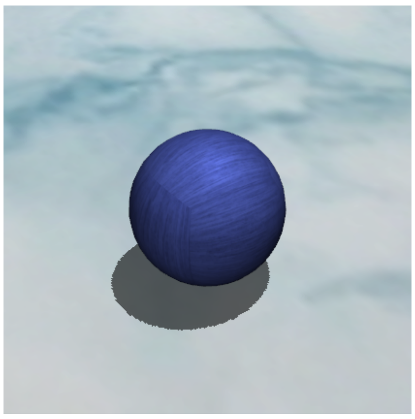
Figure 4.6: MPL with touch sensors annotated [68, chap. 6]

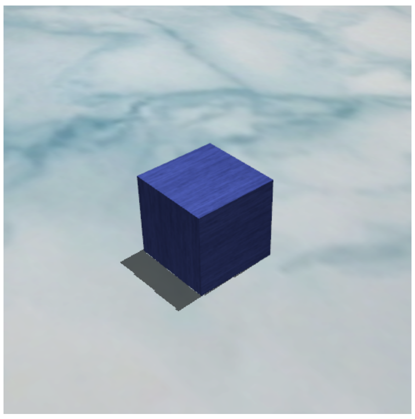
Sensors There are four touch sensors on the palm of this model, two on the front, one in the back and one on the side. Also, each finger has three of them on their digital bones: proximal, medial, and distal, see Figure 4.6

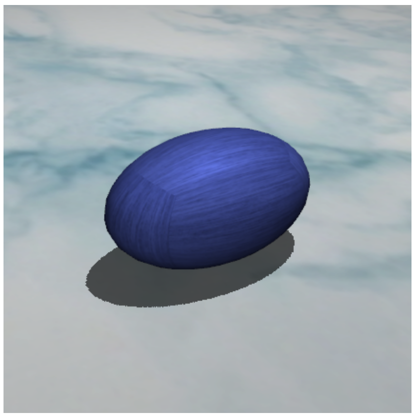
Controls (MPL motors) Unlike in the case of gripper, not all of the joints of this model are controlled by separate controllers. There are 13 position servos to move the 22 joints of the hand, three for the wrist and ten for the fingers. As described in the HAPTIX chapter of the MuJoCo documentation book, some of the joints have their own motors, while others share one [68, chap. 6].
Similarly to the other model, this one can be freely moved in three-dimensions as well.

4.1.3 Shapes

We experimented with several object of different shapes and sizes (Figure 4.7). All of the models are simple, convex objects, ideal for two, three and four finger prehension.


(a) Sphere


(b) Cube


(c) Ellipsoid

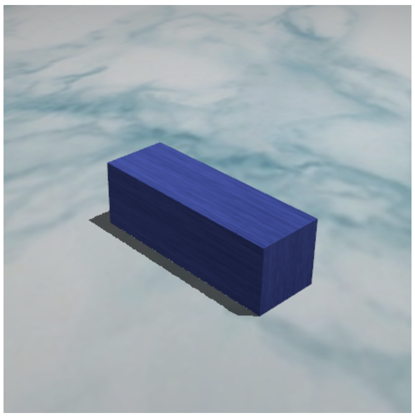

(d) Cuboid

Figure 4.7: Object candidates

4.2 Our Deep Reinforcement Learning Framework

In this section we describe how deep reinforcement learning and the algorithms presented in Chapter 2 are applicable to simple grasping tasks by describing our agent, environment, reward function, and other components of DRL.

4.2.1 State space representation

It is the state based on what the agent chooses for its next action, therefore it is a crucial component and its representation matters a lot. If the state does not contain enough information of the environment, the agent might fail to learn the given task. For our experiments the state vector includes each joint position value (i.e. the angle of each joint) and the touch sensor values, however it does not contain the position of the manipulated object.

We will also experiment with excluding joint position data from the state vector relying only on tactile information and compare what the agent could learn.

In practice, state values are often squashed between -1 and 1 because neural networks are known to be more stable and learn better that way. In our case these values are already quite small, definitely between $-\pi$ and π for the joint data. And we cut off touch sensor data with values above 20 therefore touch sensors have a range of $[0, 20]$, however most of the time it is below 1 as well.

Important note: the state of the agent and of the environment are usually assumed to be the same for most problems. In our case, however, they are slightly different. Our environment state also includes the position vector of the object, while the agent's state only contains the joint angles and touch sensor values. We will be referring to the latter by state throughout this report, except when we compute the reward $r(s, a, s')$ in which case s also includes the object position since it is used to determine the reward. However, it is very important that the agent only sees the state without the object position data.

Terminal states Terminal states are states with no transition to another state. If the agent gets into a terminal state the current episode ends. For our experiments the terminal states are mostly equivalent to goal states.

4.2.2 Action space representation

The robot hand is manipulated by actions chosen by the agent's policy. Each joint, and the arm itself have three corresponding actions altering their angles with a constant value c:

- increase angle: $angle(joint_n, t+1) = anlge(joint_n, t) + c$
- leave angle unchanged: $angle(joint_n, t+1) = anlge(joint_n, t)$
- decrease angle: $angle(joint_n, t+1) = anlge(joint_n, t) - c$

Action Name	Effect
a_1	decrease angle of joint 1
a_2	leave angle of joint 1 unchanged
a_3	increase angle of joint 1
a_4	decrease angle of joint 2
$\vdots$	$\vdots$
a_n	increase angle of joint m, where $m = (n/3) + (n \mod 3))$
$\vdots$	$\vdots$
a_{N-2}	decrease arm height
a_{N-1}	keep arm height unchanged
a_N	increase arm height

Table 4.2: Actions

We also conduct some experiments where we have only two actions per joint: increasing and decreasing the angle, and compare the results to see if that extra action is necessary in our original setup.
We set our neural network architecture in such a way, that there is one output neuron corresponding to each action, or more precisely, each output neuron represents one state-action or Q value, as described in [1].
The function this architecture represents differs from the original $Q(s,a)$ function (Section 2.3.5) in the sense that it accepts states as inputs instead of state-action pairs, and it returns the Q-value for all possible actions. This way we do not need to compute the Q-values separately for each state-action pair, our neural network computes all of them in one go, thereby saving expensive computation time. So instead of

$$Q(s,a) \rightarrow \text{value}$$

we have

$$Q(s) \rightarrow \text{array of values (one for each action)}$$

Furthermore, we will also experiment with altering this model in a way where instead of selecting only one action at every time step (by choosing the action corresponding to the highest Q-value), we take the $\arg\max$ of each output neuron trio corresponding to the same joint resulting in a vector of actions (with length equalling to the number of joints) at every step. Then, we can execute all actions in the vector simultaneously and the fingers will be able to move together.

4.2.3 Agent

Our goal is to teach our agent various object manipulation tasks. The agent's function is to observe the environment and to take actions: flex/extend the fingers, increase/decrease the height of the arm. After every action it observes the new state of the environment, which is a set of touch sensor and joint position values.
When learning our agent has two policies.

- One is for exploring and gaining new experience of the form (s_t, a_t, r, s_{t+1}) (state, action taken in that state, reward received, new state). It is the so called ϵ-greedy policy, see Section 2.3.8.
- The other policy learns from the experience gained by the ϵ-greedy policy, and it simply takes the action corresponding to the maximum Q-value in every step, see Algorithm 2.5.3.

Once training is finished the agent uses its learnt policy to choose the next action based on the current state. For instance, if it can see from the state that none of the touch sensors are on and the arm height is high it would take an action that lowers the arm (until touch sensor values increase),

4.2.4 Environment

Our environment is the simulation, usually consisting of an object placed on a tabletop and the robot hand floating above the table. A legitimate question would be: why does not the arm belong to the agent? Is it truly a part of the environment? We could say that the arm itself belongs to the agent, however, the observations it makes through its touch sensors and joint position values are of the environment. Note that the agent can compute its own joint angles (by keeping track of the changes made to them) as long they can move freely, but once it meets an obstacle a joint angle will not change as much as the taken action would require it to. This is also true when for instance the hand holds the object while gravity is pulling it down.

4.2.5 Reward function

One of the most challenging part is to choose a sufficient reward function as efficient as possible, since we define the task for the agent through the reward function, and it is the only feedback to know whether the agent is doing the right actions or not. We could reward for instance the agent for touching the object, and reward it even more when it starts lifting it. Steps without any object manipulation, on the other hand, could be penalised and thus discouraged. For more details see Section 6.4, where we detail how exactly we chose our reward functions.

4.2.6 Function Approximator

Due to the huge state space – all joint position combinations multiplied by the 3D position of the arm multiplied by all the touch sensor value combinations – it is impossible to map states directly to actions (using a table), we need a function approximator, see Section 2.4. In our case, our function approximator is non-linear, a deep neural network with multiple hidden layers. It can learn to generalise from experience to previously unseen input. That is, if we train our neural network with a large number of various experiences, it will learn to map states it has not seen before to optimal actions.

4.2.7 Policy

As a reminder: a policy $\pi(s)$ decides on the next action to take by the agent based on the current state of the environment, for more details see Section 2.3.4. Q-learning, being an off-policy algorithm has two policies: one exploring, generating experiences, and one that is being improved. We want the agent to learn a policy that makes it take the sequence of actions that can successfully complete the given object manipulation task. While training it follows a behaviour policy ϵ-greedy in this case.

4.2.8 Algorithm breakdown

As in every reinforcement task, we have an agent interacting with an environment, observing the states of this environment, and taking actions that can change this environment. The goal is to learn a policy which, when followed by the agent, produces the largest possible (or sufficiently high) cumulative reward. If the reward function and thus the problem is well defined, then the agent should be able to learn to successfully execute the proposed task. Below, we go trough the workflow presenting and explaining how the Prioritised Experience Replay algorithm (Algorithm 5) works in practice.

In training mode

- Agent requests the state of the simulator, which is a set of joint position and touch sensor values.
- The agent then uses its ϵ-greedy policy to find the next action.
  - With $1-\epsilon$ probability the agent feeds the received state to its Q-network whose output is the Q-values of the actions, and then the action with the largest Q-value is taken. Or
  - with ϵ probability a random action is chosen uniformly from all possible actions.

  Each joint has three actions corresponding to them, increase/decrease/do not change its angles.
- The agent sends this action to the simulator where it is executed and a new state is sent back, furthermore a reward $r(s,a,s')$ (function of state, action, new state) is computed and a flag signalling whether the given new state is terminal or not.
- The agent saves (s, a, r, t, s') or (state, action, is state terminal, reward, new state) tuples (called experiences or transitions) in its replay memory. In our implementation the replay memory is based on a 'sum-tree' data structure, when a new experience is added to the memory it gets a priority assigned to it, which is initially the maximum priority in the tree so far (assuring that every experience is replayed at least once).
- From time to time the agent samples a batch of experiences from the memory. The sampling method is described in Section 2.5.3 in more detail. The batch will contain experiences of the form (S,A,R,T,S'). In order to train the Q-network we have to feed it an input and a target. The input will be S and the target will be $Q(S)$ with $Q(S,A)$ changed to
  - $R+\gamma Q_{target}(S', \arg\max_a Q(S',a))$ if S' is terminal (i.e. T is true)
  - R otherwise.

That way, we improve $Q(S,A)$, so that it will be more accurate.
We also compute the TD-error δ as

$$\begin{aligned}\delta &= R + \gamma Q_{target}(S', \arg\max_a Q(S', a)) - Q(S,A) && (4.1)\\ &= Target - Q(S,A) && (4.2)\end{aligned}$$

Then we have

$$|\delta| = |Target - Q(S,A)| \tag{4.3}$$

And use this error to update the priority p of this experience in the replay memory.

$$p = |\delta| + \epsilon \tag{4.4}$$

Furthermore we compute the weight w for experience j using Equation 2.53. After this we update the Q-network with the above defined input, target and weights.

- Occasionally we merge the Q-network with Q_{target} by copying the weights of the formal to the latter.

In testing mode

- Agent requests state of the simulator.
- Q-network computes next action based on the state.
- Agent executes this action.
- Repeated until a terminal state or the maximum number of steps is reached.

In Figure 4.8 we present a summary of the architecture of the Deep Reinforcement Framework.

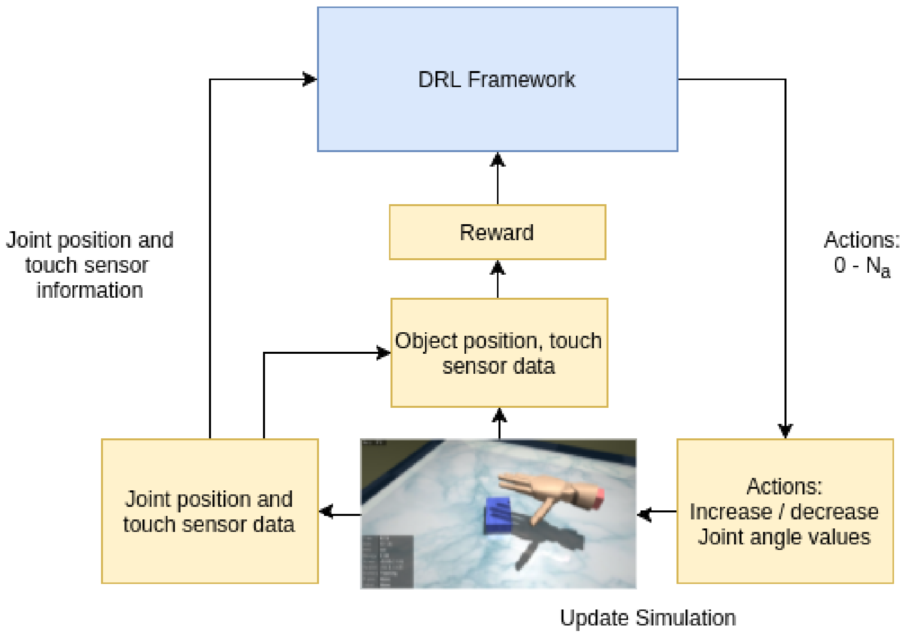

DRL Framework
Reward
Object position, touch sensor data
Joint position and touch sensor information
Actions: 0 - N_a
Joint position and touch sensor data
Actions: Increase / decrease Joint angle values
Update Simulation

Figure 4.8: The DQN computes all Q values for the current state, we take the best action, and execute it. Following that we capture the new state of the simulation represented by the positions of the joints, and the signal from the touch sensors placed on the fingers. We also compute the reward from the new state based on the position of the object we manipulate. Design based on [11, Fig. 1.b]

4.3 Hardware requirements

As machine learning projects generally do, this one relies on large computational power as well, mostly because of training deep neural networks. However, the simulation also requires good graphical resources. Tensorflow has versions optimised for both CPU and GPU, MuJoCo is optimised for CPU, therefore the need for powerful CPU, and possibly GPU. Experience replay stores large amount of data in memory, so large memory is a priority, too. We do not do heavy disk reads and writes so for disk both ssd and hdd are suitable. Also, as speeding up a single training can be limited by many aspects, the possibility of running several experiments parallel is also a preferable option. Based on the above, it is clear that a normal notebook is not the most suitable candidate for this task with trainings often taking over a day. We chose to try the cloud resources of Amazon (Amazon Web Services) and Microsoft (Microsoft Azure).

Name	Service	CPU	Cores	RAM	GPU	Storage	Seconds per epoch
Notebook	-	Intel Core i5-3210M @ 2.50 GHz	2/4	8 GiB	NVIDIA GeForce GT 630M	SSHD	16.2
Remote 1	AWS	Intel Xeon E5-2670	4	15 GiB	NVIDIA Grid (Kepler GK104)	SSD	-
Remote 2	Azure	Intel Xeon E5-2673 v3 @ 2.40 GHz	4	14 GiB	-	SSD	12.5
Remote 3	Azure	Intel Xeon E5-2690 v3 @ 2.60 GHz	6	56 GiB	NVIDIA Tesla K80	HDD	8.4

Table 4.3: Hardware comparison

Where speed (seconds per epoch) is computed as the average time to run one epoch of training (when the agent has not learnt anything yet), measured with the same experiment for all computers. The result for Remote 1 is missing because we had no longer access to it when this comparison was conducted.

Chapter 5

Program description and details

In this chapter we introduce some of the technical challenges and details of implementing the program accommodating our experiments. We start by listing and describing its most essential components.

5.1 Components

5.1.1 Python-side

Brain This component is the brain of the agent, a wrapper for the Q-network used to predict the next action based on the current state of the environment. It can build a neural network based on some given hyperparameters such as the number of layers and number of neurons and activation functions for each layer. It can also save the current model into a file or load any file containing a model into memory and use that as the current model. Furthermore it is capable of adding additional input and output nodes to an existing network. This is useful when we do curriculum learning and add complexity to a previously trained networks.

Agent The agent interacts with the environment through the Robot Interface module and relies on the Brain module to compute the next action. It keeps exploring the state space until it learns the right policy. To learn from experience the agent uses the Prioritised Experience Replay algorithm (Algorithm 2.5.3).

Robot Interface This component is a decorator for the environment on the Python side by using and extending the functionality of the environment on the C++side. It is responsible for translating and sending the current action (being taken by the agent based on the decision of the brain) to the environment. It also reads the state of the environment, extracts the relevant information which then the agent can process. Moreover it computes the reward $r(S, A, S')$ and decides whether a state is terminal or not (for instance whether goal is reached).

Configuration files / Setups As we conduct experiments with tens of different settings and configurations, it is logical to have separate files representing and describing these specifications. The configuration of each experiment is made up of two files, a Python and a C++, the former consisting of:

- Q-network parameters (layers, number of neurons, activation function, drop-out)

- RL parameters (discount factor, memory size, initial/mininal ϵ, etc...)
- Robot parameters (turn on/off touch sensors/joints, reward function)
- Paths to save/load models and stats to/from

Controller The responsibility of the controller is to put together the whole setup and to start the training of the agent. Furthermore it saves the current configuration and settings and results in files so that it can be later compared with other configurations and results.

Other components Further modules include

- An optimisation module to automatically find hyperparameters using grid search, random search or Bayesian optimisation.
- A memory module, making it possible to use other data structures for replay memory, such as a simple queue for traditional Experience Replay (Section 4)
- A module containing helper functions for logging, managing files, etc...
- A module for experimenting with other algorithms instead of Deep Q-learning.

5.1.2 C/C++ side

As mentioned in Chapter 3, our simulator, MuJoCo has a C++interface requiring to implement some parts of our application in C++.

Simulator API This is the main program that makes the simulation work using the MuJoCo API. It loads the given model into the program and runs the simulation loop. It also has a function to reset the simulation and we also set up a callback function which is called at every step of the simulation and can be used to control our robot hand.

Additional functions All the functionalities not elementary part of the simulation are included in this component. This includes

- Functions to read and generate an array of joint position and touch sensor values composing the state of the environment.
- Callback function that controls actuators on joint positions.
- Functions that communicate with Python side.
- Functions to run additional, dynamic initialisation of the loaded model.

Configuration files / Setups We already presented the Python configuration files and their functionalities, here we list their C++counterparts. The C++configurations are responsible for the following:

- Initial arm positions (especially height) (can be random as well)
- Initial joint positions
- Initial object position and orientation (can be also random)
- Object size and type

5.2 Communication between Python and C++

One great challenge of having written the simulator in C++and our agent in Python is that the two sides need to communicate with each other somehow. When programs or processes communicate with each other and manage shared data on the same platform it is called Inter-Process communication and can be handled by the following solutions:

Shared memory Shared memory is a part of the memory that is shared between different programs, that is, multiple programs can access it simultaneously and share data between each other. Its advantage is that it is the fastest solution for different processes to communicate, with the disadvantage being that userss need to implement locking for themselves.

Named pipes Another solution for inter-process communication is called pipes, an extension of the standard pipe concept on Unix like systems. A named pipe is like a standard pipe but has a name and does not die with the process. It is ideal for transferring data from one process to another without creating a temporary file. It is slower than shared memory and provides one-way FIFO communication only.

Domain sockets Domain sockets are similar to Named pipes, except that they can be used for communication both ways and are easier to use. They are also very similar to network sockets except that they communicate exclusively within the operating system's kernel.

We decided to go with the domain socket option since it is easy to implement, can be used for both way communication between the Python and C++ sides, and although it is slower than using shared memory, it would not create a bottleneck in our case, since the transferring time through sockets is dwarfed by the training time of the neural networks and the simulation. In order to make the implementation even easier we used the ZeroMQ (or ZMQ) library [78].

Chapter 6

Experimentation and Results

In this section we present the experiments we conducted by describing the configuration, the goal as well as presenting and evaluating our results. Before introducing the actual experiments, however, we will discuss a few general comments and settings.

6.1 Measuring progress

In order to be able to follow the progress of our trainings and measure the successfulness of a given configuration, we need a good measure to reflect them meaningfully. The most straightforward solution is to keep track of the total reward received for each episode. By definition, the larger the total reward for an episode, the better that episode is, so plotting the total reward represents the agent's learning curve. Although we know that more reward is better than less reward, the accumulated reward per episode can only show relative success (unless we know the maximum achievable cumulative reward), showing how fast our agent learns, but not whether it has completed the given task.

An even simpler solution is to keep track for each episode whether it was completed (task achieved) or not. Then we can compute the moving average of completed episodes for a given window size (e.g. 100). If it reaches 1 or 100% we know that our agent has perfectly learnt the given task. A disadvantage of this measure is that if for some reason the agent always fails to reach its goal, the percentage of completed episodes is 0 and we can not tell whether our agent learns at all.
We can also plot the average maximal Q-value per episode, which converges as the training is converging. As the maximal Q-value represents the value of action in the current state, usually the average maximal Q-value graph will have a similar curve as the average reward graph, that is (unless Q-values were initially rather overestimated), the average maximal q-values grow as the training procedure proceeds.

6.2 Hyperparameter searching techniques

Hyperparameters are parameters that describe the model but are not learnt by the model, such as the number of neural network layers, activation functions, discount factor, replay memory size, etc. One shortcoming of most machine learning algorithms of our time is that they depend on a huge number of hyperparameters, which, if not set carefully could make the model fail to learn the underlying function of the data. For instance a complex task cannot be learnt by an inflexible network, because the network simply does not have the representation capabilities to properly fit the unknown function. Or, if we do not discount our rewards in reinforcement learning, the agent would not be encouraged to find the big reward as fast as possible, since it is enough to find it at any point in the episode.

Generally we prefer models that are less dependent on their hyperparameters and are flexible enough to learn with a wider range of settings.

One disadvantage of Prioritised Experience Replay is that it has a large amount of hyperparameters. In this section we describe how these can be found and set.

Grid search For each hyperparameter the user provides an array of possible values, then the Cartesian product or combination of all these arrays is taken, i.e. in every iteration we choose one value from each array until all combinations are tested. During testing the main function which is subject to optimistion is executed and a score generated. The maximum score and the corresponding hyperparameter array is returned at the end.

l1\l2	10	15	20
10	10; 10	10; 15	10; 20
20	20; 10	20; 15	20; 20
30	30; 10	30; 15	30; 20

Table 6.1: Simple grid search example: rows determining the number of neurons in layer 1, and columns the number of neurons in layer 2, with initial candidate sets {10, 20, 30} and {10, 15, 20}

Random search Instead of systemically going through all the combinations, random search chooses a value uniformly for each hyperparameter from a pre-defined array or range. In practice random-search usually provides results similar to grid search and one of its advantaged is that it can be stopped anytime.

Bayesian optimisation Unlike Grid search and Random search Bayesian optimisation [79, 80, 81, 82] examines the parameter space in a more intelligent manner. It tries to fit a Gaussian Process [83] to predict the next hyperparameter vector to be tested. Bayesian optimisation is usually applied when executing the objective function is very expensive, and we can only afford it a limited number of times. If well-calibrated, Bayesian optimisation can find an efficient set of hyperparameters much faster then the alternatives, without the proper setting, however, it can prove completely useless. It usually works better for smaller hyperparameter space.

Trial and Error Another common technique is when the users experiment with different hyperparameters based on their intuitions, they try various settings keeping those that produce better results and dropping the ones that fail to do so. Trusting intuitions to choose the next set of hyperparameters can be both advantageous and disadvantageous, we can rely on our general knowledge and beliefs helping us to guess the ’real’ solution, it can, however, make us miss good possible settings if we have distortion or bad assumptions in our beliefs (which is usually the case).

Literature When applying the previous technique we can make our guesses more educated if we utilise relevant, commonly cited literature such as research papers and books.

One can look up resources dealing with similar topics and problems as their own project and alter the hyperparameters from these sources to fit their own problem.

6.3 Finding the right hyperparameters

In the previous section (Section 6.2) we have shown techniques to find hyperparameters in general. In this section we present how we have found and set the hyperparameters for our experiments.

First we list (most of) the hyperparameters we need to set. Some of them are more important and need to be chosen carefully, some can be easily set based on literature and common knowledge.

List of hyperparameters for the Q-network

- Number of layers
- Number of neurons for each layer
- Activation function for each layer
- Loss function
- Learning rate
- Drop-out rate, validation split

List of hyperparameters for experience replay

- Discount factor $\gamma \in [0,1]$
- $\epsilon_{max} \in (0,1]$ initial value of ϵ (used for the ϵ-greedy policy)
- $\epsilon_{min} \in [0,1)$ minimum value
- reduction function for ϵ
- Replay memory size
- Replay period
- Episode maximum length
- Experience replay batch size
- Target network update frequency
- $\alpha \in [0,1]$: how much prioritisation is used
- $\beta \in [0,1]$: how much importance sampling is used

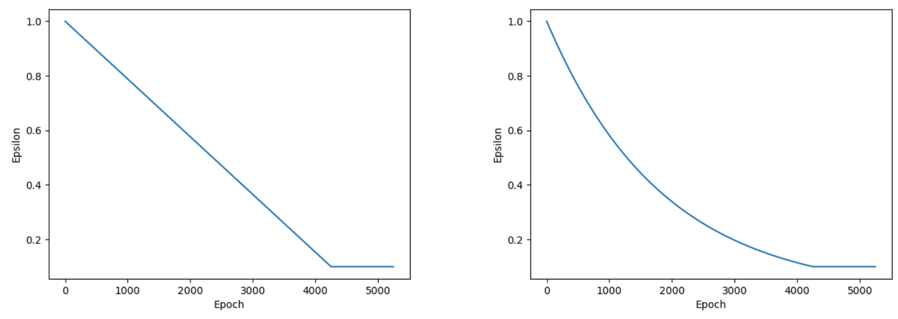
Now we describe how some of these values are chosen.

Neural network configuration Based on similar projects [11] and other deep Q-network implementations like [1], it was clear that the network should have two or three hidden layers with 100-5000 neurons each. We also know that larger networks take longer to train, but are more flexible and can represent more complex functions, however they are prone to overfitting, i.e. because of their flexibility they can fit the training data 'too well', by also learning the noise in it thereby loosing their generality. On the other hand, smaller networks are faster to train, they are less likely to overfit, however, they can lack the capability of representing the given function resulting in underfitting. In the early stages of the experimentation we used grid-search (Section 6.2) to determine the ideal network size, assuming two hidden layers, and then refined them by manual tuning. For more details see the concrete experiments.

Replay memory size The size of the replay memory determines how long old memories i.e. experiences of the form (S, A, R, T, S') should be remembered. Typical range of memory size for problems like ours is between $100,000$ and and $1,000,000$.

Initial and minimum ϵ As described in the background section (Section 2.3.8) ϵ is the probability of taking a random action instead of the action the Q-network suggests when we follow the ϵ-greedy algorithm. When the training starts, ϵ is set to an initial value (usually 1) and gradually reduced according to some function of the epochs until it reaches ϵ_{min}. With $\epsilon_{min} > 0$ there will remain a small chance for the agent to take random actions and keep exploring until the training finishes.

We can decrease ϵ linearly by always deducting the same small amount from the current value, or it can be reduced exponentially, by multiplying the initial value with the same small amount.



(a) Linear epsilon reduction **(b)** Exponential epsilon reduction

Figure 6.1: Epsilon reduction

Discount factor γ Choosing the discount factor is easier than finding most of the other hyperparameters, since essentially all relevant works set it to 0.99 for such experiments [1, 11]. Nonetheless it is worth testing values between 0.95 and 0.99.

6.4 Finding the right reward function

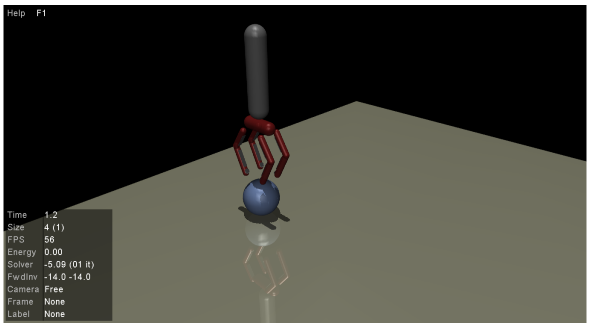
The reward function is one of the most fundamental part of a reinforcement learning task. It is the component that leads the agent to learn the desired task and find a policy that produces a maximum cumulative reward. An ill-calibrated reward will likely produce unsought effects.

In the original Atari paper [1] the authors suggest to normalise the reward function to a range between -1 and 1, though they also say that it is not essential. The reward function, based on the original definition (Section 2.3.3) has a form of $r(S, A, S')$, that is, it is a function of the current state, the action taken in this state and the new state the environment is transformed to.

When designing the function it makes sense to give a high positive reward when a goal state is reached, high negative reward when the agent ends up in an undesirable state and small, real-valued rewards otherwise. In this last case, the reward for non-terminal steps, can be somewhat problematic. If it is negative then it will motivate the agent to take as few steps as possible before reaching the goal, however, at the same time it also discourages exploration, and more complex operations will be less likely to be executed even if they give larger rewards in the end. This is addressed by the ϵ-greedy approach to some extent, although it requires careful calibration. On the other hand, if non-terminal rewards are small positive values, the agent might have difficulties with finding the goal state (especially if that has low chance to be found by accident), since the agent will not be discouraged from repeating the same, non-terminal sequence of steps and accumulating rewards that way.

Also, during the experiments we encountered some undesired behaviour when altering the reward function to penalise the robot hand with a large negative reward as it goes below a certain height, while we made such states terminal. In some cases the absolute value of the negative reward was not large enough to be less profitable than keeping exploring. I.e. the accumulated negative rewards gathered while exploring were more than the one time big negative value in this terminal state, even though if the agent had followed the right policy this would have not been the case. To solve this issue, we could either give even bigger negative rewards for such states, or stop by marking them as terminal.

6.5 Lifting a sphere with the gripper



```
Time     1.2
Size     4 (1)
FPS      56
Energy   0.00
Solver   -5.09 (01 it)
FwdInv   -14.0 -14.0
Camera   Free
Frame    None
Label    None
```

Figure 6.2: Four finger gripper

In order to keep things simple and add complexity gradually we started with the simpler, gripper model shown in Figure 6.2, instead of the MPL hand.

State dimensions	11
Number of actions	9
DOF	3

Table 6.2: Summary

Neuron id	Description
0	right claw increase angle
1	right claw keep unchanged
2	right claw decrease angle
3	left claw decrease angle
4	left claw keep unchanged
5	left claw increase angle
6	arm move upwards
7	arm keep unchanged
8	arm move downwards

Table 6.3: Action Descriptions

In this first experiment the goal was to lift a sphere by 0.03 unit sugar up to a height of 0.26 units. The arm was placed above the sphere at a height of 0.15 units. We let the arm move only along the vertical axis and therefore the setup had altogether three degrees of freedom: one-one to control the joints of the claws, and one for changing the arm's vertical position. The sate of the system was represented by the position of the three joints (two joints and the arm's height) and the eight touch sensor values (there are four fingers with two touch sensors on each of them). There were nine actions (three for each joints), see Table 6.3.

6.5.1 Right setup

This being our first experiment, it took significantly longer to make the setup work than in the case of following experiments. Finding the right hyperparameters, tuning the reward function and the simulation settings proved quite challenging. We decided to use grid search to find the ideal network size (assuming that it has two hidden layers) and the replay memory size. It turned out that initially we underestimated both with 128 neurons per layer and a memory capacity $100,000$.
After numerous attempts and weeks of experimentation we ended up with the following hyperparameters:

Memory size	300,000
Discount factor	0.99
Neurons	1024, 512
Epochs	3000
Number of steps per episode	250
Initial ϵ	1
Minimum ϵ	0.01
Replay period	every 4 episodes
Target network update frequency	every 1000 training steps
Prioritisation exponent α	0.6
Initial importance sampling exponent β_0	0.5

Table 6.4: Hyperparameters

We used a neural network with two hidden layers to map joint positions and touch sensor data to Q-values.

Table 6.5 provides a summary of the neural network architecture and details.

Layer	Input	Activation	Output
1	11	ReLU	1024
2	1024	ReLU	512
3	512	Linear	9

Table 6.5: DQN Network Architecture.

Finding the ideal reward function was even more crucial and difficult at first, than choosing the right hyperparameters. Our initial reward function was simply a linear function of the object height:

Algorithm 6 Reward function

```
if $height(object, t) > 0.26$ then
    return reward: $10$, terminal: True, goal reached: True
else
    return reward: $height(object)$, terminal: False, goal reached: False
end if
```

However this proved to be unable to properly learn our proposed task. We decided to apply different functions such as exponential, quadratic, square root, and others to the object height, but none of them could get the agent to perfectly learn to grip. This could be explained by our discussion in Section 6.4.

Finally we decided to try a different approach:

Algorithm 7 Reward function

```
if $height(object, t) > 0.26$ then
    return reward: $10$, terminal: True, goal reached: True
else if $height(object, t) > height(object, t-1)$ then
    return reward: $0.1$, terminal: False, goal reached: False
else
    return reward: $-0.11$, terminal: False, goal reached: False
end if
```

This reward function (Algorithm 7) gives a positive reward (0.1) when the manipulated object's height in the current time step is greater than in the previous time step. Otherwise it gives a negative reward at a slightly larger magnitude (-0.11). There are two major differences between this solution and the previous reward function (Algorithm 6):

- The second solution more explicitly rewards the lifting action, instead of computing the reward based on only the state the agent arrived at.

- Our second solution penalises every other actions and this way it discourages infinite exploration.

6.5.2 Results

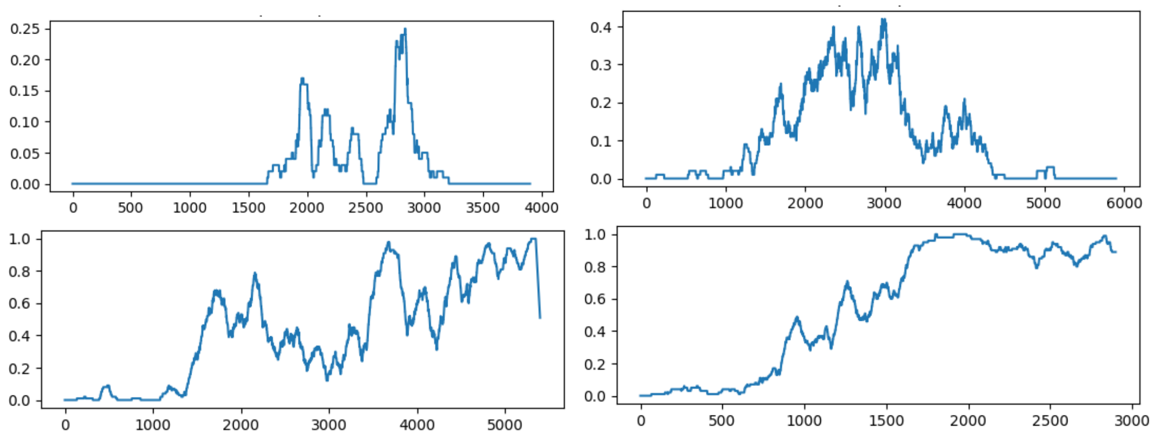
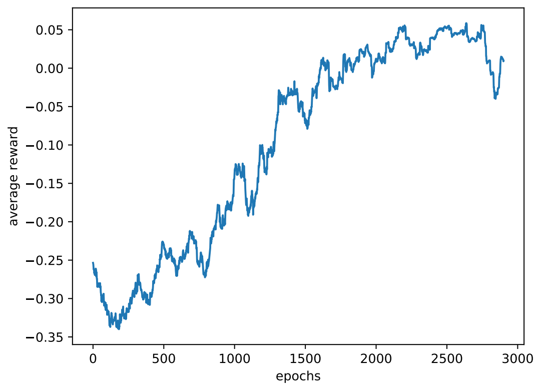
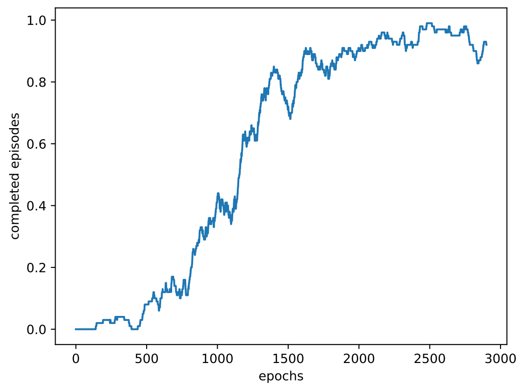


Figure 6.3: Proportion of completed episodes of trainings with different hyperparametes

Figure 6.3 presents four plots of four different trainings, with different settings, showing how our results improved over time.



(a) Average reward



(b) Percentage of completed episodes

Figure 6.4: Results

Figure 6.4 shows how the values of average reward and completed episodes change with the number of iterations in the case of optimal specification. Examining figure 6.4b we can see quite a steep learning curve, and by epoch 2500 the agent has learnt the task perfectly.

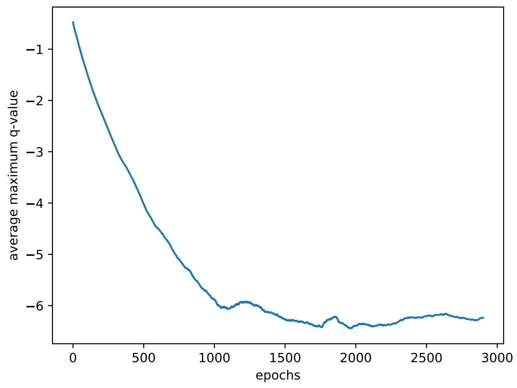


Figure 6.5: Average maximal q-value.

In Figure 6.5 it can be seen how the average maximum q-value converges with the number of epochs.

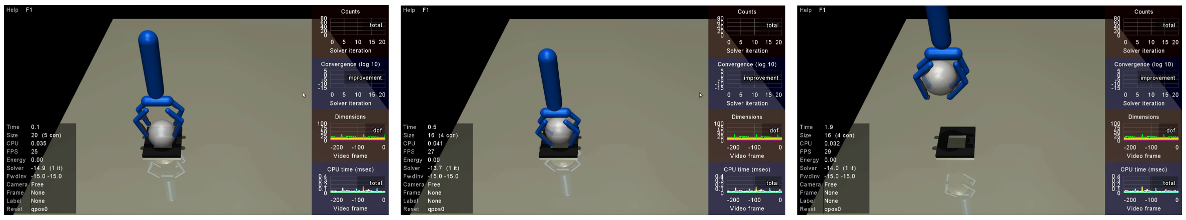


Figure 6.6: Working gripper

6.5.3 Conclusions

Based on this result we can conclude that it is possible to learn to lift objects relying solely on tactile and joint position data of the environment. We have also seen how the effectiveness and success of the training depends on the right reward function and the importance of the correct hyperparameter settings.

6.6 Claw with varying starting height

A somewhat more challenging task is to find and lift the same sphere if the initial position of the arm is not constant but randomly set for every episode. This requires a bit more intelligent behaviour on the part of the agent, because in the previous setup the agent could have learnt that it needs to close its claws after a certain number of steps, instead of making the decision to close the claws based on whether it can feel the sphere with its touch sensors or not. Of course we cannot know what the Q-network exactly learnt, but we can test the model from the previous experiment with this new setup to see whether it is general enough to lift a sphere when we vary the starting height of the arm.

In this setup the initial arm height was uniformly chosen from a range between 0.10 and 0.20.

6.6.1 Trying the model from the previous experiment

The model that was trained with constant initial height of the arm failed to lift the object in this case, the model not being general enough. Therefore, we had to try other options.

6.6.2 Retrain the model

An obvious solution is to retrain the model from the previous experiment from scratch in order to have it learn this new task. We were not focusing so much on this solution, however, the few experiments we have conducted with the same or similar hyperparameters as for the previous experiment have failed. We could have increased the number of training iterations, or the size of the Q-network to get better results, yet we decided to focus more on the next option.

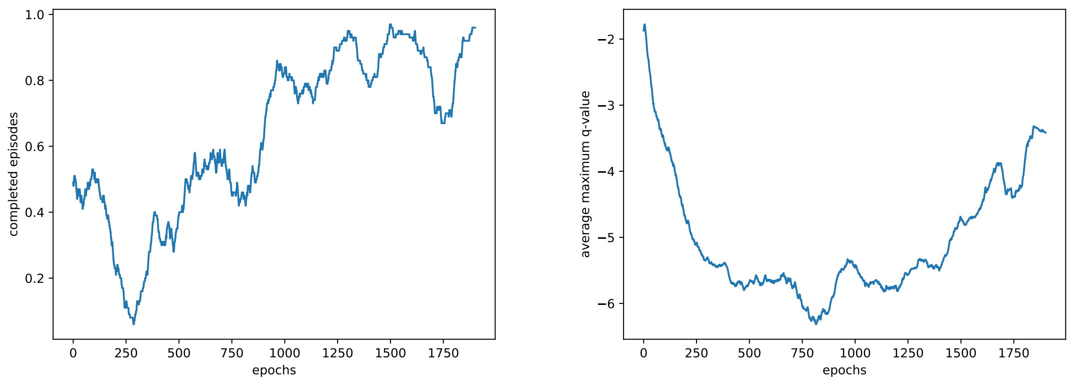
6.6.3 Curriculum learning

Finally, another option is to use the model from the previous experiment, and continue training it for a while (ideally for a shorter period than the original training time) so that it further generalises our already existing model. This solution worked surprisingly well. We could use the same neural network architecture and keep most of the settings. We reduced the number of epochs, the size of the replay memory and the initial value of ϵ.

6.6.4 Right setup

Memory size	100,000
Epochs	2000
Initial ϵ	0.2

Table 6.6: Hyperparameters



(a) Percentage of completed episodes **(b)** Average maximal Q-values

Figure 6.7: Random starting height with curriculum learning

On Figure 6.7 we see that by epoch 2000 the agent learns to lift the sphere with the initial arm height set randomly.

6.6.5 Conclusions

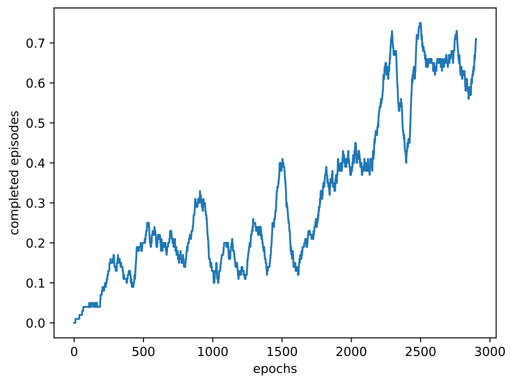
This was the first experiment in this project that proved that a simple version of Curriculum Learning can be used to add complexity to an already working model. When we tried to train the agent with setting taken from the previous experiment, or with the setting of the curriculum learning setup but without the pre-trained model, it failed both times.

6.7 Binary vs. real-valued touch sensors

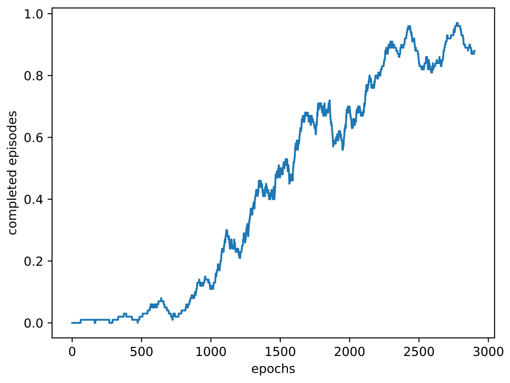
Touch sensors take real values, their range depends on the settings and physical properties of the objects in the simulations. Based on our observations they are usually between 0 and 120 for the gripper and between 0 and 25 in the case of the hand. However, in some circumstances during training (like pressing fingers too hard to the ground or to the object) these values can become much larger and neural networks are known to be sensitive to large values, therefore we decided to cut off touch sensor values above 20, which proved to be an ideal threshold.

We also ran experiments with touch sensors set to binary, i. e. 0 when not touching and 1 when touching. One could argue that, for instance in the case of the gripper, it is enough to know whether the claws touch something or not and binary values are known to be processed easier by neural networks. However, we found that we get better results with real values, and thus we came to the conclusion that the intensity of the sensors is important and utilised by the Q-network.

Below we can see the difference in the training progress when binary touch sensors and when touch sensors with cut-off at 20 are used. It is clear that – although both agents learn quickly – the original, real-valued solution is more effective.



(a) Percentage of completed episodes with binary touch sensors

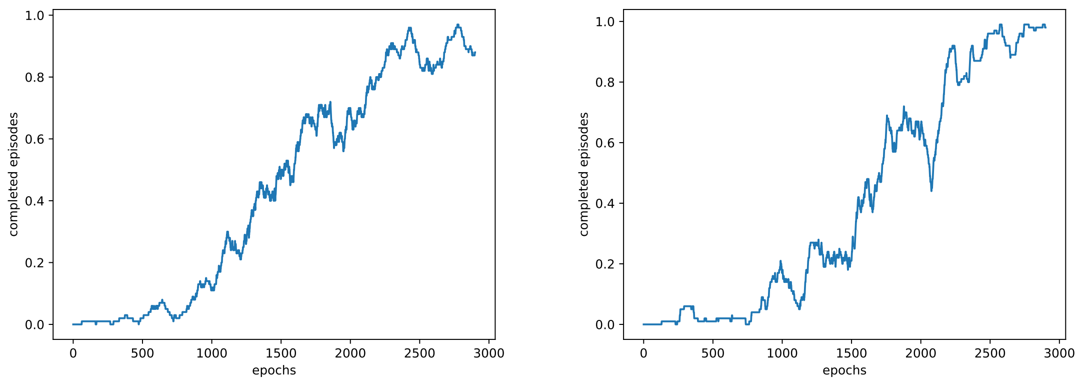


(b) Percentage of completed episodes with real-valued touch sensors

6.8 Simultaneous actions vs. single action

The original Q-learning algorithm (Algorithm 3), just like the versions featured in the Atari paper [1] and other works as well, choose the next action to take to be the one with the maximum Q-value, or a random action. This way only one action is executed at each step, which is fine when actions are mutually exclusive (i.e. going left or right in a grid world). In the case of a robotic hand, however, moving joints simultaneously would be desirable and more real-life like (although in our simulation actions are taken so quickly after one another that it appears to be moving simultaneously even in the sequential case).

We propose a version of Q-learning where instead of choosing the action with the single maximal Q-value, we select an action for each joint at every step by taking the one with maximal Q-value for every three neurons corresponding to a given joint. This way we have a vector of actions at every step that can be taken simultaneously.



(a) Percentage of completed episodes taking multiple actions simultaneously

(b) Percentage of completed episodes with single action per step

Figure 6.9

Interestingly, this proposition neither improves nor worsens the results, see Figure 6.9 i.e. we get similar results when settings otherwise are mostly identical (in case of the simultaneous actions experiment we decreased the number of steps in an episode from 250 to 200).

6.9 Two vs. three operations per actions

As described in Section 4.2.2, we have three actions for each joint position servo: decrease angle, keep angle unchanged, increase angle. So why is this setting better than having only two actions: increase angle and decrease angle? One could argue that this second option would be more efficient since there should always be, at every step, a joint that changes its angle (either in positive or negative directions), unless the task is completed, when the episode is marked as terminal anyway.
In practice, however, having three actions per joint proved more efficient despite the fact that it requires larger output space. We could think of two explanations:

1. With three actions per joint more sub-optimal action suggestions by the neural network are tolerated.

2. There are forces such as gravity (and only gravity in this case) that affect the target object even if it is not manipulated by the gripper. Having the option of not taking actions in a time step could provide more flexible interaction with the environment.

6.10 Relying on tactile information only

Another interesting setup is when the agent does not know about its joint position values getting the touch sensor reads only, i.e. when we exclude joint position values from the state fed to the Q-network. One could argue that this would be a more natural setting, although we humans do have an inner concept of how much our finger joints are bent, even if we do not know the exact angles.

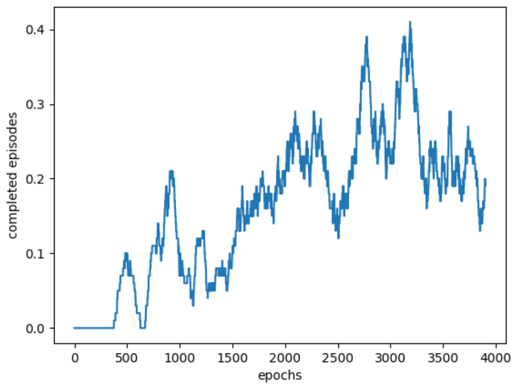
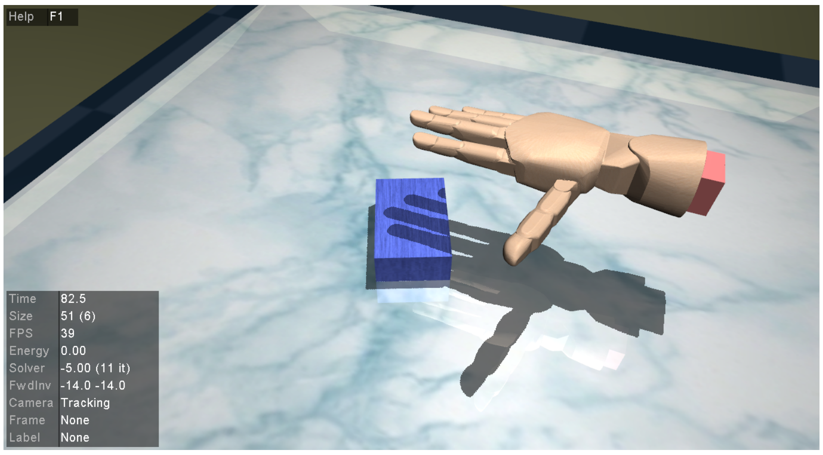


Figure 6.10: Completed episodes with input state excluding joint position data

Figure 6.10 shows that the agent could learn grasping up to 40% success rate without joint position data.

6.11 Robothand

After some experimentation with the gripper model, exploring different behaviours and options, we decided to switch to the hand model.



```
Time     82.5
Size     51 (6)
FPS      39
Energy   0.00
Solver   -5.00 (11 it)
FwdInv   -14.0 -14.0
Camera   Tracking
Frame    None
Label    None
```

Figure 6.11: Hand with 5 fingers and 22 joints

Joints	DOF	Touch sensors	Position sensors	Motors	Controlled DOF
22	25	19	22	13	16

Table 6.7: MPL hand summary

6.12 Lifting with two fingers

In our experiments with the gripper we had three degrees of freedom, so switching to the hand model with 16 DOF would have been a rather large jump in complexity. In order to keep things simple, we decided to start experimenting using only two of the fingers (thumb and index), and possibly adding more fingers depending on our results.

As we can see in Table 6.8, there are 22 joint in the MPL hand model, see Table 4.1 from Chapter 4 for more details. We can also move the arm in 3D, adding three more degrees of freedom. There are three touch sensors on each of the fingers and four on the other parts of the hand. For each joint there is a position sensor, and our position data also includes the 3D position of the arm. The fingers and the wrist are controlled by 13 position servos and again, there are three more controlled DOF for the arm.

The setup of this experiment was quite similar to the ones with the gripper. Our first goal was to lift a small cube above the height of 0.25 units. The hand was placed above the cube at a height of 0.17 units and was allowed to move along the vertical axis only.

Joints	DOF	Touch sensors	Position sensors	Motors	Controlled DOF
8	11	6	8	6	7

Table 6.8: Summary

As it is presented in table 6.8, in this setup we have 8 joints, four and four for the thumb and index finger each, with the vertical movement of the arm. That means seven controlled degrees of freedom, since the thumb has four position servos, the index finger has two and we also can also change the arm's height. Both fingers have three touch sensors, and each joint in the finger has a position sensor.

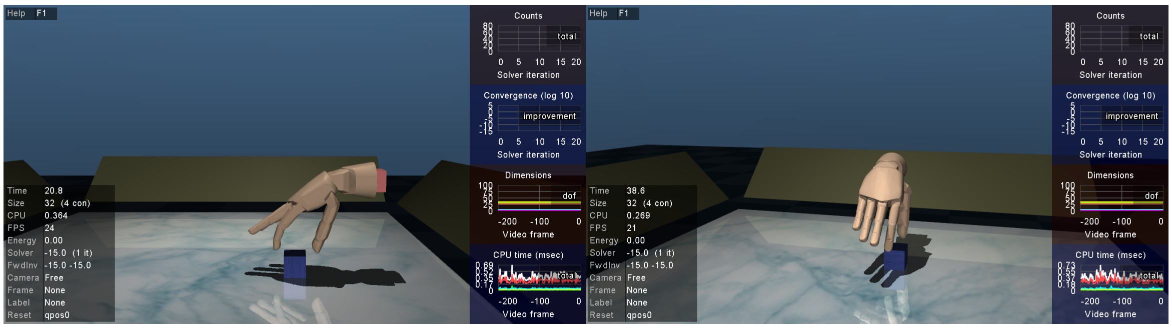
state space dimensions	15
output size	21

Table 6.9: State and action vector size

Based on the above, table 6.9 summarises the size of the state space (8 joints + arm height + 6 touch sensors), which will be the input for the neural network, and the size of the action vector, the output of the neural network (6 motors + arm height control, 3 actions for each).

6.12.1 Setting the initial position

We also set the initial position of the thumb and index fingers for every episode, as can be seen on Figure 6.12 in order to have a fingertip contact with the object. This is achieved by increasing the wrist flexion joint and bending the thumb and index fingers.



```
Time     20.8
Size     32  (4 con)
CPU      0.364
FPS      24
Energy   0.00
Solver   -15.0  (1 it)
FwdInv   -15.0 -15.0
Camera   Free
Frame    None
Label    None
Reset    qpos0
```

```
Time     38.6
Size     32  (4 con)
CPU      0.269
FPS      21
Energy   0.00
Solver   -15.0  (1 it)
FwdInv   -15.0 -15.0
Camera   Free
Frame    None
Label    None
Reset    qpos0
```

Figure 6.12: Initial position

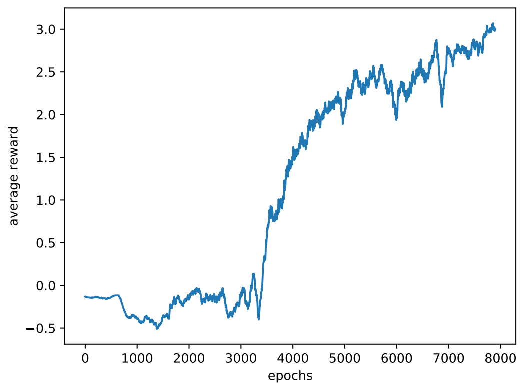
6.12.2 Right setup

Neurons	1024, 1024
Epochs	8000
Step size	500

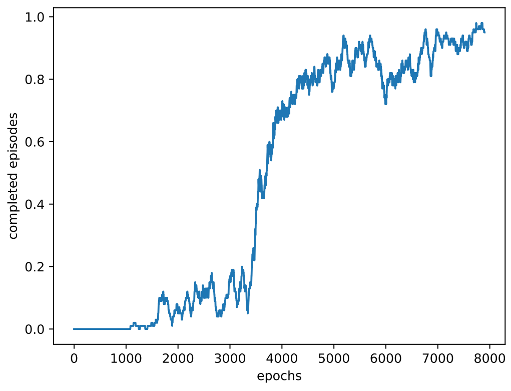
Table 6.10: Updated hyperparameters

6.12.3 Results

In this setup it took significantly longer for the agent to learn the right policy than in the case of the gripper (about twice as many epochs, each with 500 steps instead of 250), however, we can see a similar jump in the average received rewards as in Figure 6.13a between epochs 3500 and 4000. Figure 6.15 shows how the MPL hand lifts the cube once it has learnt the optimal policy.



(a) Average rewards



(b) Completed episodes

Figure 6.13: Results

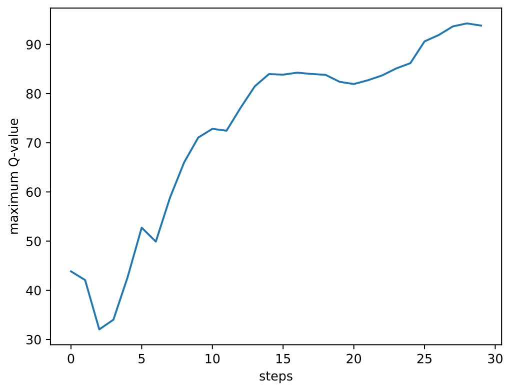


Figure 6.14: Q-values in a single episode once the agent has learnt the optimal policy

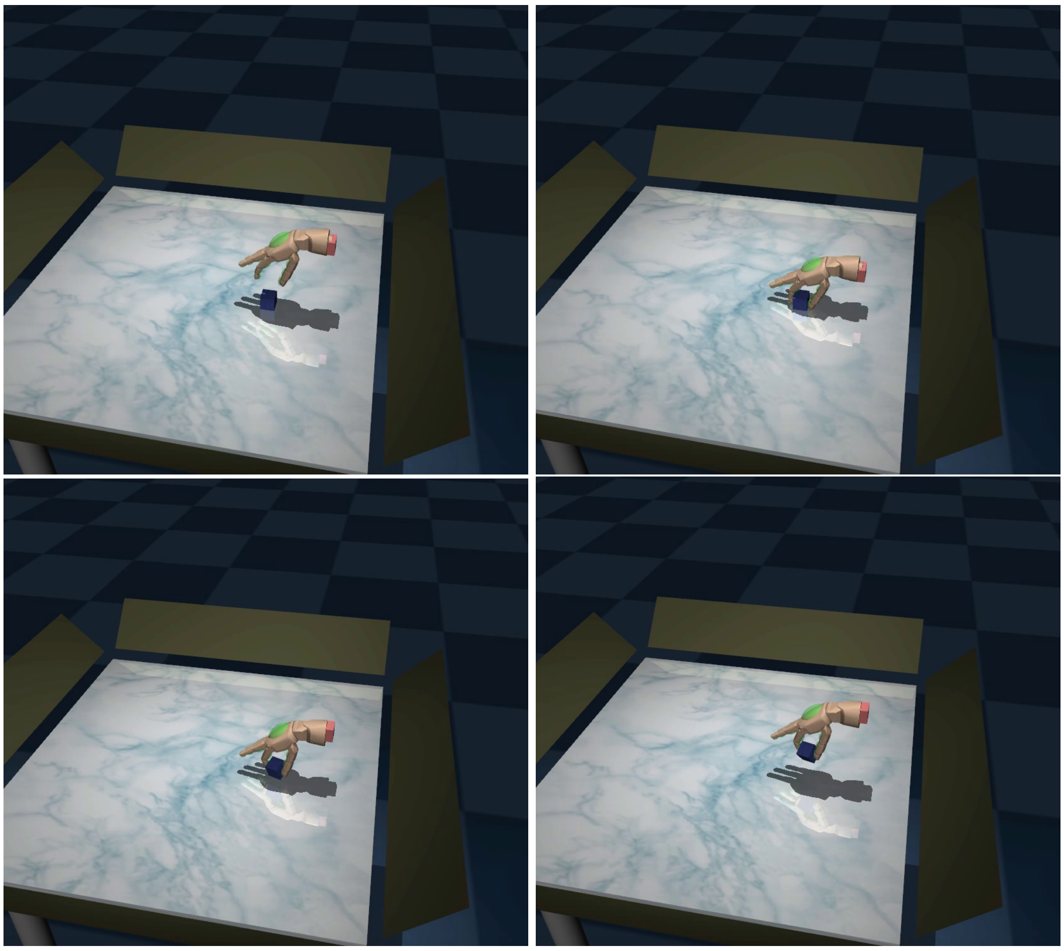


Figure 6.15: Precision grasping with two fingers

We can place this type of grasping in the category of precision grasps with thumb abducted using pad opposition type and two virtual fingers using Figure 2.2 from Chapter 2.

6.12.4 Conclusions

After training our 3 DOF gripper model we could also successfully train the MPL hand model with 7 DOF to use precision grasping for grasping and lifting a cube, even if it required twice as many and twice as long episodes.

6.13 Random initial height

Similarly to the case of the gripper, we altered this experiment with the starting height of the arm randomly selected from a given range ($[0.1, 0.3]$). Relying on the results of the experiment in Section 6.6 and the fact that training from scratch would take even longer than for the previous experiment, we decided to try curriculum learning again.

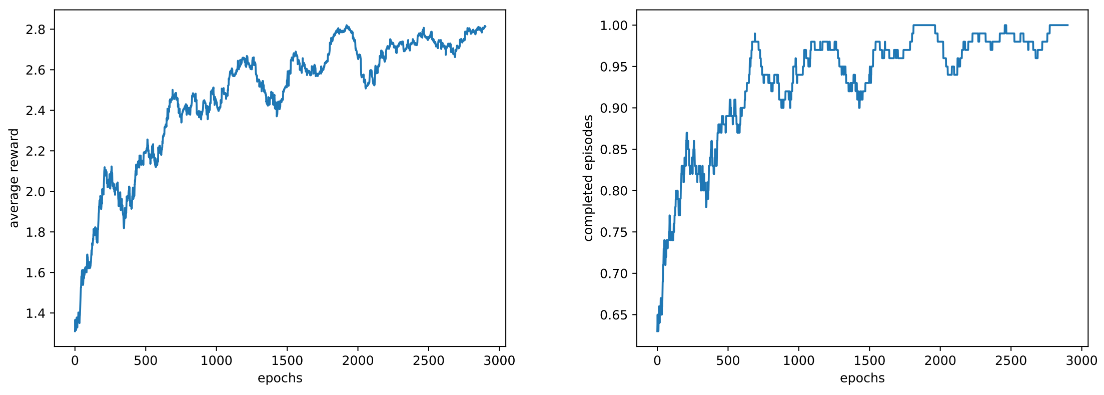
6.13.1 Right setup

The working configuration:

Memory size	100,000
Epochs	3000
Initial ϵ	0.2

Table 6.11: Hyperparameters for curriculum learning with the MPL model

6.13.2 Results



(a) Average rewards **(b)** Completed episodes

Figure 6.16: Learning curve with curriculum learning in case of random initial arm heights

We can see rather quick convergence on figure 6.16, the neural network from the previous experiment could be generalised easily to adapt to random initial heights.

6.13.3 Conclusions

With this experiment we saw again how we can teach a model that has been already trained, to do more complicated tasks quite efficiently with curriculum learning.

6.14 Varying the shape and size of the object

The agent could easily learn to pick up a cube, but what about a sphere? It is a somewhat more challenging task since a sphere could roll away, furthermore, it has a smaller area ideal for proper grasping. Can the same agent learn to grasp different objects?

Just like in the case of the previous experiment, we could easily teach the hand to lift objects with different shapes including spheres, ellipsoids, cubes and cuboids of different sizes. The agent had no trouble to learn these grasps from scratch with full training, however, with curriculum learning we achieved similar, fast convergence just as in the case of the random starting height experiment. In these experiments the shapes and sizes of the object were randomly chosen each time the model was reset (and started a new episode). Note that when we added different shapes to the cube-lifting model we only introduced one new shape per training instead of adding all of them at once.

6.14.1 Results

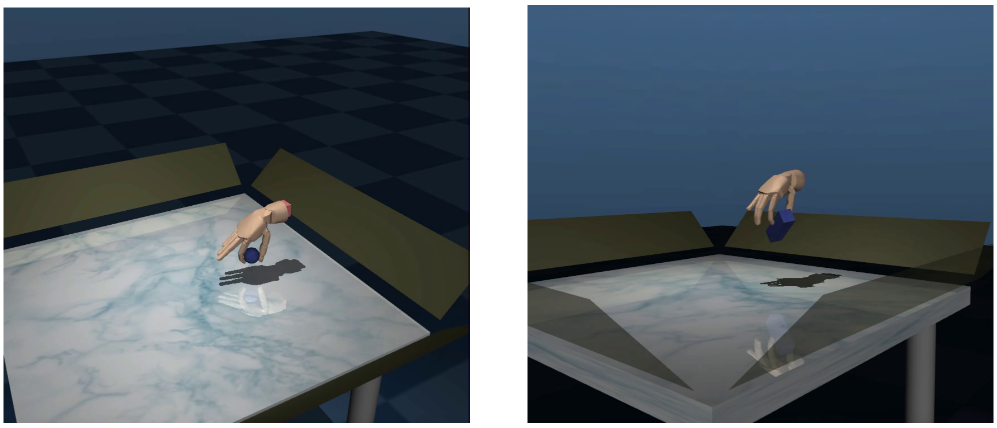


Figure 6.17: Lifting different objects with two fingers

6.15 Different ϵ values per actuator

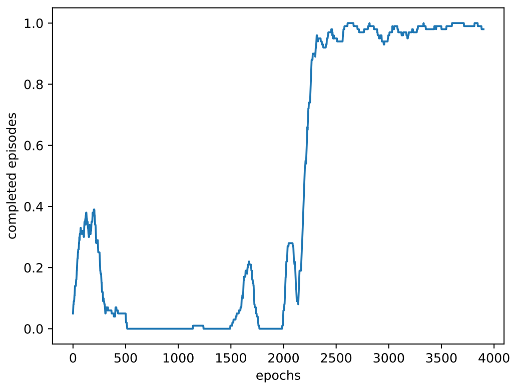
When we use the ϵ-greedy algorithm to decide on the next action, we choose a random action with ϵ probability at each step of every episode and rely on the Q-network's suggestion otherwise. As the training proceeds we gradually reduce ϵ until it reaches a predefined final value ϵ_{min}. In our experiments the task is to grasp a given object and then to lift it above a certain height. In this setup the movements different joints perform have different levels of complexity. For instance lowering the arm until the fingers grasp the object then lifting it up is a less challenging task than finding the right grasp with the fingers. This motivated us to investigate how introducing separate ϵ values per position servo could alter the convergence speed of the training. We defined an array of initial ϵ values, one ϵ for each actuator. When choosing the next action the neural network would propose one, which then would be altered with probability ϵ^i where i corresponds to the different position servos.

Algorithm 8 ϵ-greedy with different ϵ values per motor

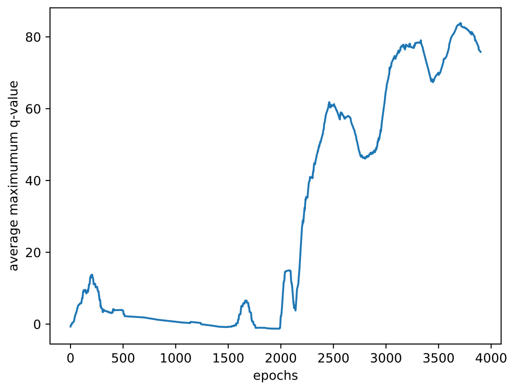
```
Input: array of $\epsilon$ values $epsilons$
1: $a \leftarrow \arg\max Q(s,a)$
2: if $random() < epsilons[a/3]$ then
3:     $a \leftarrow chooseRandomFrom(allActions)$
4: end if
Output: $a$
```

6.15.1 Results

For instance we can set all ϵ_0-s to 1 except for the one actuator that moves the arm up and down, which could be set to 0.2. We show the result for this setting below.



(a) Completed episodes



(b) Average maximal Q-values

Figure 6.18: Results

Looking at figure 6.18a it is clear, that this is a huge improvement compared to the original setting, getting almost 100% success rate by epoch 2500 instead of 6000-8000. It is also interesting how sudden the curve jumps a little after epoch 2000. We rerun the experiment and got a smoother curve for the second time.
Trying this variant with a sphere produced similarly good results:

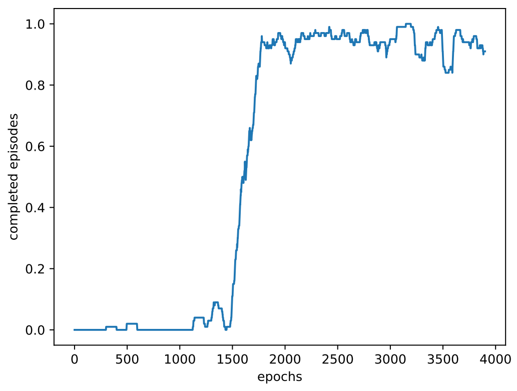


Figure 6.19: Learning to lift a sphere using different ϵ values per actuator

6.16 Searching the plane for the object

We conducted some experiments where the object was placed randomly on the plane within a certain radius of the arm. We wanted to see whether the agent could learn a policy where it had to find the object by applying some kind of space-searching technique discovering the space with its fingers. We rewarded the agent when it managed to move the object in any direction, signalling that the object had been successfully found. This proved to be an infeasible task, which is not very surprising since this would have required much more significant generalisation of the neural network from the training data. Finding the object with a constant position, on the other hand, was easily learn by our agent.

6.17 Extending the Q-network

Applying curriculum learning is not too difficult when we do not change the model settings and use the same number of fingers for both experiments. However, in the case of introducing new fingers (or other joints) we have to deal with different input and output space sizes. For instance turning on the middle finger additionally to the thumb and index fingers, we will have four more joints, three more touch sensors in the case of the input space, and one more actuator (since the MPL model has one position servo for the middle finger) in the case of output space.
We overcame this issue by simple introducing new input and output neurons initialised either with random weights or copying weight of other neurons with the most similar functionality. For instance for the middle finger MCP motor neuron we can use the weights of the neuron for the index finger MCP motor.

6.18 Three-finger lifting

After the success of the two finger precision grasping experiments we decided to introduce a third finger, the middle finger.
One of our goals was with three-finger experiments to find a setup where the agent would fail to lift an object with two fingers but manage to do so with an additional finger.

6.18.1 Setups and results

As before, we had two options to train our models: either from scratch or with the help of curriculum learning, extending two-finger models.
First we experimented with the former option: we tried to train a new model for three finger grasping.
For the initial experiments we kept the hyperparameters and reward function unchanged and saw a quick convergence in training. However, the results were surprising at first glance. Instead of utilising all three fingers, our agent still used only two fingers. In some cases the thumb and the index finger and in other cases the thumb and middle finger combinations.
To solve this issue we modified our reward function so that it would only give positive rewards when the distal sensors (the ones on the tips) on each of the three fingers are non-zero. This way we forced the agent the grasp with three fingers. However, with this setup we could not teach the agent to grasp. As it can be seen on Figure 6.22 the best success rate the agent could achieve (with increased number of epochs) was around 7% at the peak.

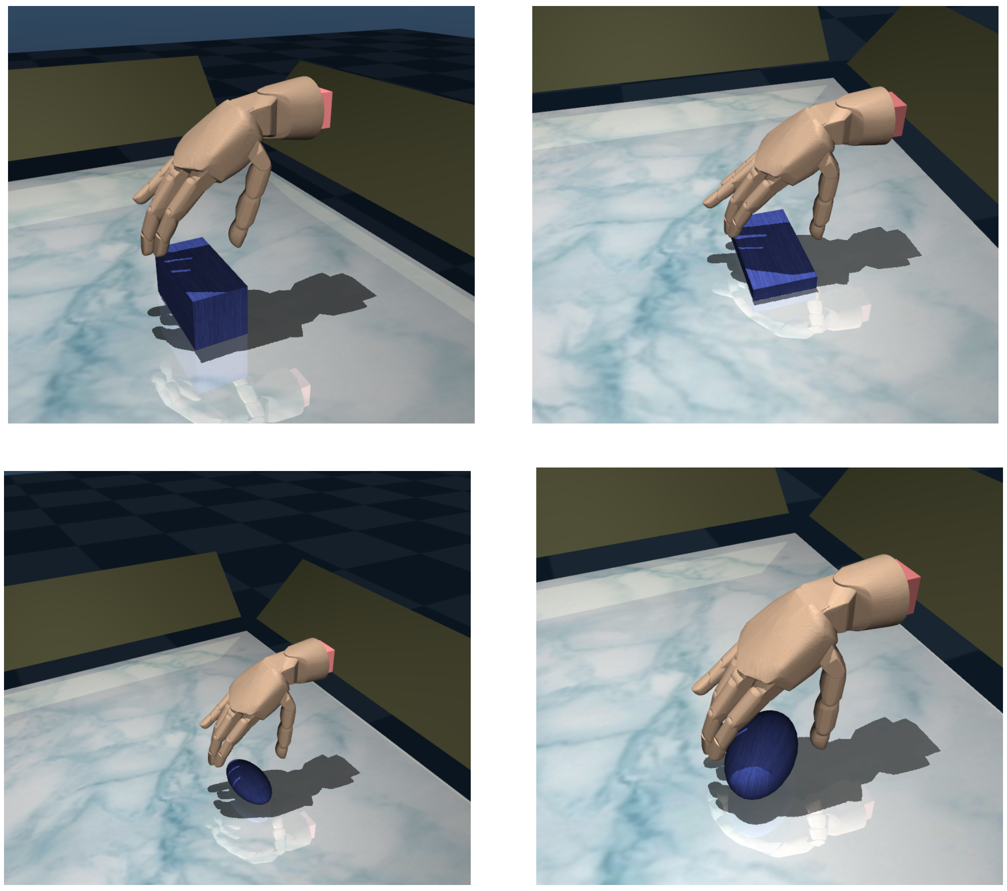


Figure 6.20: Objects for three-finger grasping

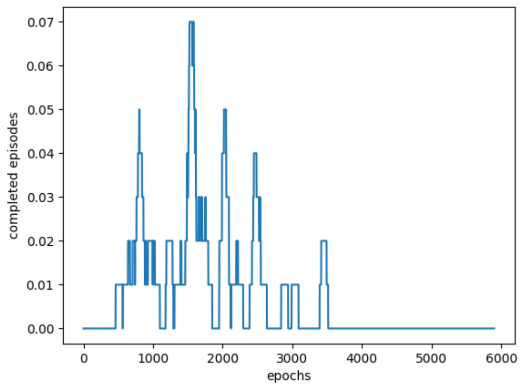


Figure 6.22: Success rate with three finger grasping

Also, examining Figure 6.22 we can see that the agent had some initial success which decreased with the number of epochs. One explanation could be that the ϵ values were reduced too quickly, not allowing sufficient exploration time for the agent. Longer training might would have produced better results.

With manual testing, i.e. moving the joints with preprogrammed actions we also discovered that the all three distal touch sensors are non-zero condition was rarely met even if all three

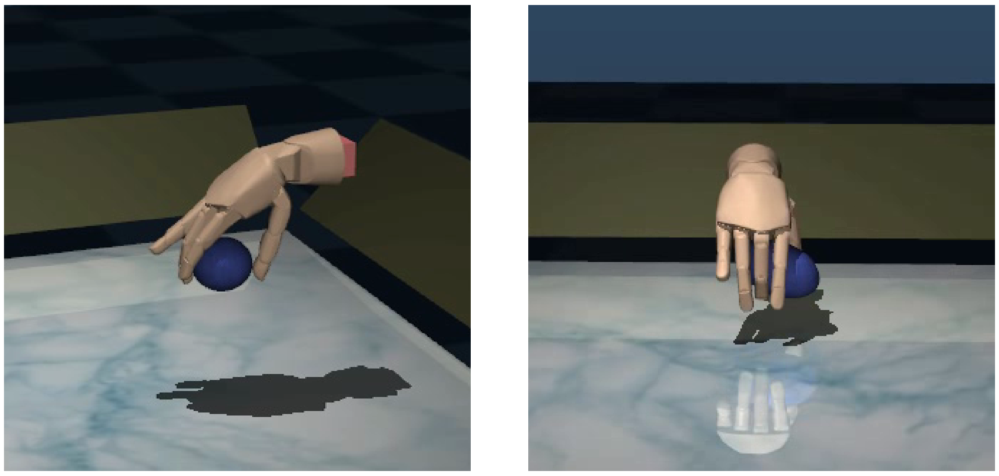


Figure 6.21: Objects intended to be grasped by three fingers were grasped by two fingers instead

fingers were touching the object with their distal phalanx. From this we can conclude that more or larger touch sensors could have improved the results.

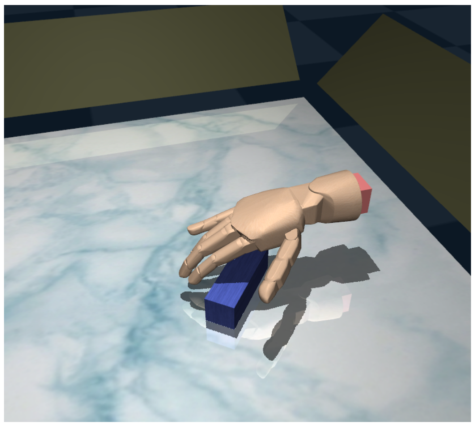


Figure 6.23: Three finger grasping setup with different initial position

On Figure 6.23 we see a modified initial position of the hand and object trying to resemble a more natural human grasping pose. However, we had no success with this setup, either.

Later we moved to curriculum learning hoping we would have more success with it. Although we have not managed to thoroughly experiment with different hyperparameter settings we have not managed to achieve better results.

6.18.2 Conclusions

Despite the success with two-finger lifting we had troubles when one more finger was added. The agent either learnt a two-finger grasping as before, or did not manage to learn to grasp at all. We believe that results could be possible improved by implementing one or more of the possible options:

- longer training time

- larger touch sensors
- more carefully constructed reward function
- more in-depth experimentation with curriculum learning
- increasing the weight or decreasing the friction properties of the object, so that it would more easily slip when grasped with two fingers

6.19 Resistance to shaking effect

Another variant of the previous experiment is where the arm is exposed to a shaking effect from the environment, and while we expect the arm to drop the manipulated object if it is held by two fingers, we wanted to see if it manages to keep holding the object when three fingers are used.
Shaking was implemented by introducing random noise in the robot arm's horizontal position once it had lifted the object above a certain height.
Applying this effect to models that had been already trained with two fingers achieved its purpose: the arm dropped the manipulated object. However when we retrained the agent it either managed to keep holding the object with two fingers or failed to learn grasping at all, similarly to the previous experiment.

Chapter 7

Evaluation

In this chapter we attempt to evaluate this project by discussing its strengths and shortcomings and comparing our results with related research. Moreover we discuss the novelty of our results and argue that they are useful contributions to the field.

7.1 Strengths

Deep Q-learning works with tactile feedback for grasping Our experimentations have shown that deep reinforcement learning, and the Deep Q-Learning algorithm in particular, is capable of learning simple grasping tasks relying on tactile feedback. We have successfully taught agents to grasp and lift objects of different sizes and shapes and to complete the task even when we vary the starting height of the arm or expose it to shaking during lifting.

Applied curriculum learning to gradually increase complexity We also successfully applied a simple variant of curriculum learning to teach the agent more complex tasks gradually that it could not have learnt or would have learnt with longer training times without splitting them up into simpler components. With curriculum learning our agent could learn to grasp different objects, or work with random starting height of the arm in often less than 2000 epochs.

Efficient implementation We used the most state-of-the-art variant of deep Q-learning available, with Double deep Q-network and prioritised experience replay [1, 50, 48]. This way our agent could learn to complete a given task twice as fast as with simple deep Q-learning with experience replay.

Useful comparisons In the case of our gripper model we compared how our agent learns when joint position data is included and when it is excluded from the state what is fed to the Q-network to determine the next action. Although without the position data our agent failed to learn the task with full success, we saw evidence that it can still learn and complete the task up-to 40% success rate.

We also tested Deep Q-learning with actions chosen for each joint simultaneously at every time-step, instead of using a single action at a time. With this modification the agents could learn equally well.

Efficient extensions to Q-learning In Section 6.15 we presented a novel variant of the Deep Q-learning algorithm which could speed up our training times by more than 65% in some cases. In our implementation we modified the ϵ-greedy policy in a way where different actions are replaced by a random action with different probabilities in order to keep actions that are less likely to be already correct changed more often.

Flexible environment We implemented a flexible deep reinforcement framework (Chapters 4, 5) which can accommodate all kinds of grasping experiments with a wide variety of settings such as varying the model used for the simulation, turning joints, controllers and touch sensors on and off, calibrating the reward function and the hyperparameters, configuring logging and option to save and organise results automatically.

7.2 Weaknesses

Limited success with more complex tasks Despite our success with simple settings we also saw that with additional complexity, the agent fails to learn in reasonable time. Despite trying a large variety of configurations we have not succeeded with three or more finger grasping properly. Possible solutions for this issue could be longer training times, better hyperparameter or reward function settings, more touch sensors placed on the fingers, usage of other sensory input like vision or other kinds of haptic feedback, or exploiting curriculum learning more (see next point).

A shortcoming of our experimentation framework is that because of the huge number of configurable settings it is often difficult to know exactly which settings should be changed in the first place.

Not enough research on curriculum learning Although we had success with curriculum learning, due to time and resource limitations we failed to explore it more in depth and conduct further experiments for example in the case of three-finger grasps. It could have been also interesting to implement a variant of automatised curriculum learning described in the recent paper [55] called automatic goal generation, which uses Goal Generative Adversarial Networks to dynamically generate the reward function as the complexity of the task increases (within a single training).

Limitations of the algorithms One possible reason for difficulties with more complex experiments could be linked to the numerous hyperparameters of Deep Q-learning, it would have been interesting to see how other, more recent algorithms such as A3C [10] – a policy gradient method which is known to have less hyperparameters and less complex implementation – could have been applied to this problem.

Also, large state and action spaces are difficult to handle by today's machine learning algorithms in general. To deal with huge dimensionalities, training such algorithms – depending on the problem – can take days or weeks.

Sensitivity to initialisation Related to the previous point, our setups and algorithm are quite sensitive to the initialisation of the fingers and position of the robot hand. Our agent could learn only when it was initially placed rather close to the target object with its fingers bent and failed to learn without this initialisation with otherwise matching configuration. This is understandable to a certain extent since our agent is 'blind' and has to rely on its touch sensors to navigate in the environment. With better exploration strategy, however, this problem could have been overcome. It would have been interesting to see a bit more 'creativity' on the part of the agent i.e. finding solutions less suggested by the initialisations.

Simulation only experiments Although we initially considered to try transferring our working models to real robotic hands at some point, our research went in other directions such as exploring curriculum learning. Furthermore it would be useful to learn a wider range of object manipulation tasks and three or more finger grasping as well before applying the models to real-life tasks.

7.3 Comparison with related work

Quantitatively evaluating our project and experiments is a difficult task since exact training times are rarely available for related work. Moreover, the differences in settings, the tools available, algorithms used and goal specifications make it impractical to do such comparisons and draw quick conclusions.
Nonetheless, we present a short summary of the details here about one of our training setups. When correctly calibrated and applying our ϵ per actuator extension (Section 6.15) to deep Q-learning our agent could learn to grasp simple objects with two fingers within less than four hours on a modern computer. We summarise our two-finger cube lifting experiment in numbers below:

Episodes	4000
Training steps per episode	500
Training steps in total	2 million
Total training time	3-4 hrs
Average episode length	2.7 - 3.6 seconds

Table 7.1: Cube lifting experiment summary

Concerning related works, in [11] we see a rather similar solution both in terms of the problem and in the techniques involved. This paper summarises an in-hand object manipulation experiment using vision-based Deep Q-Learning with Convolutional Neural Networks. They use the same MPL hand model as we do, and they use four degrees of freedom. Their training consists of a total of 11 million training steps to teach their agent a simple in-hand manipulation task.

Although our experiments focus on grasping and lifting different objects instead of in-hand manipulation, and we rely on tactile feedback, not vision, requiring a different, simpler feed-forward NN architecture, we believe that our work would be a useful complementation of their results since the authors emphasise how they would leverage tactile sensory data as an extension to their experiments, while involving vision would be a sensible extension to our project.

We could also compare our experiments with the results of [22]. The authors use a Q-learning based asynchronous method to train real robots to different object manipulations tasks involving door opening and closing tasks. Their experiments involves seven DOF arms trained with a novel variant of the Normalised Advantage Function and Deep Deterministic

Policy Gradient algorithms [1] to train several robots in parallel speeding up the training times significantly. They use episodes of 300 time steps and can teach a robot to pull and push a door and execute simple object manipulation tasks with 100% success rate in 3000-4000 episodes.

[1]for more details see the paper

Chapter 8

Conclusions and Future work

In this project we explored how a recent deep reinforcement learning algorithm, deep Q-learning with prioritised experience replay is applicable to the problem of multi-finger grasping with tactile sensory input.
We introduced the background of the deep Q-learning algorithm and reinforcement learning in general, briefly discussed other grasping techniques and presented our reinforcement learning framework and experiments.
We successfully trained agents to grasp simple objects with two fingers utilising seven degrees of freedom. We also experimented and successfully applied curriculum learning to learn more tasks with models already trained for simpler tasks.
Furthermore we introduced a minor modification to Q-learning speeding up training times by more than 65% in some cases.
We faced difficulties with adding more degrees of freedom and failed to teach the agent to properly grasp with three and more fingers.

There are numerous options to continue this research such as exploring curriculum learning more in depth, applying other, more recent algorithms, use automatic hyperparameter searching techniques to find better settings and many others. In the following section we discuss these options which could improve our results and show opportunity to continue the research.

8.1 Future work

There is a great number of options to continue improving this project, it is definitely a rather open-ended area of research, with plenty of room for new solutions, experiments and discoveries. Below we list some of the options for extensions.

Adding more fingers and try other types of grasping We conducted experiments with two fingers, and three fingers for some extent, however it would be interesting to continue the research utilising three, four and all five of the fingers. Furthermore we managed to cover only a small subset of grasp types, most of the grasping table on Figure 2.2 is left to discover.

Larger variety of objects Experiments could be continued with wider range of objects, test how much our existing models could be generalised and learn new tasks, and what kinds of new models could be trained. We experimented with simple, convex shapes, learning to grasp more complex objects such as toys and everyday tools could be a next step.

Bayesian optimisation Finding the right hyperparameters is one of the major challenges and drawbacks of modern machine learning algorithms. We spent a significant amount of our time with just getting the settings right. An alternative to manual tuning is using machine learning to optimise the hyperparameters. Bayesian optimisation is a method which uses Gaussian processes to find an ideal set of hyperparameters especially useful when evaluating the objective function representing the problem is rather expensive in terms of time and computation. Although we used this technique to explore our initial hyperparameter space, we missed to apply it later for more thorough search.

Experimenting with other DRL algorithms such as A3C In the fast moving world of DRL algorithms, one or two year-old algorithms are often replaced by newer, even more efficient solutions. DeepMind's [47] most recent Asynchronous Actor-Critic (A3C) algorithm [10] is considered the best performing and most efficient RL algorithm so far for similar problems. Although it was not yet widely tested with robotic grasping tasks, it would be interesting to conduct some research with A3C.

More and more sensitive sensors In order to have more informative state space and thus more accurate results we could add more, and more accurate tactile sensors. The MPL model with its 19 touch sensors is far from the sensing capabilities of a human hand. It would be also interesting to see what results can we get if we exclude joint position data from states once more tactile sensors are used.

Other sensors Human finger tissues have a wide range of sensors, we can feel the texture, humidity, temperature, sloppiness, and other properties of the manipulated object and not just the strength of touching. The next step in mimicking human grasping would definitely require adding other types of haptic feedback to our robot hand model.

Vision for localisation and hand-eye coordination Most robotic grasping research use vision and Convolutional Neural Networks for object manipulation, while some of them, including this project relies on tactile (and joint position) data. The next step could be combining the two in order to come closer to the real human grasping process.

Different state representation Mnih et al. [1] in the Atari 2600 DQN paper used states that contained the last four frames of the game. We could alter our state representation too, so that a single state would contain the touch sensor and joint position data reads from the last few time steps instead of the last one only.

Also, completing joint position data we could utilise joint velocities and torques as well in our state representation.

Explore curriculum learning more We saw evidence that curriculum learning [53] can work with our reinforcement learning framework. [55] introduces an automatised variant of curriculum learning with dynamic goal generation, altering the reward function during training. It is definitely an option that could be a useful extension to this project.

Transfer it to real-life robot Transfer-learning is an area of research focusing on transferring models trained with simulation to real life robots or other application. Or in more general: transferring knowledge between different problems or between similar problems with different settings. It would be interesting to experiment with how trained grasping agents could be used with real-life robot hands.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," **arXiv preprint arXiv:1312.5602**, 2013. pages i, 3, 8, 26, 27, 28, 29, 31, 32, 44, 56, 57, 63, 76, 81

[2] F. Jiang, J. Yu, and Z. W. Liang, "New blood vessel robot design and outside flow field characteristic," in **Applied Mechanics and Materials**, vol. 29. Trans Tech Publ, 2010, pp. 2490–2495. pages 1

[3] J. Markoff, "Google cars drive themselves, in traffic," **The New York Times**, vol. 10, no. A1, p. 9, 2010. pages 1

[4] B. Dynamics. Robots. [Online]. Available: https://www.bostondynamics.com/robots pages 1

[5] ——. Bigdog. [Online]. Available: https://www.bostondynamics.com/bigdog pages 1

[6] Waymo. Self-driving car. [Online]. Available: https://waymo.com/ pages 1

[7] R. S. Johansson, G. Westling, A. Bäckström, and J. R. Flanagan, "Eye–hand coordination in object manipulation," **Journal of Neuroscience**, vol. 21, no. 17, pp. 6917–6932, 2001. pages 2

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in **Advances in neural information processing systems**, 2012, pp. 1097–1105. pages 3, 8, 9

[9] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," **arXiv preprint arXiv:1611.07004**, 2016. pages 3

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in **International Conference on Machine Learning**, 2016, pp. 1928–1937. pages 3, 37, 77, 81

[11] K. D. Katyal, E. W. Staley, M. S. Johannes, I.-J. Wang, A. Reiter, and P. Burlina, "In-hand robotic manipulation via deep reinforcement learning," 2016. pages 3, 32, 48, 56, 78

[12] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," **arXiv preprint arXiv:1610.00633**, 2016. pages 3, 32

[13] H. Asada, "Studies on prehension and handling by robot hands with elastic fingers," **University of Kyoto**, 1979. pages 6

[14] K. B. Shimoga, “Robot grasp synthesis algorithms: A survey,” **The International Journal of Robotics Research**, vol. 15, no. 3, pp. 230–266, 1996. pages 6

[15] J. Bohg, A. Morales, T. Asfour, and D. Kragic, “Data-driven grasp synthesisa survey,” **IEEE Transactions on Robotics**, vol. 30, no. 2, pp. 289–309, 2014. pages 6, 7

[16] R. M. Murray, Z. Li, S. S. Sastry, and S. S. Sastry, **A mathematical introduction to robotic manipulation**. CRC press, 1994. pages 6

[17] A. Bicchi and V. Kumar, “Robotic grasping and contact: A review,” in **Robotics and Automation, 2000. Proceedings. ICRA’00. IEEE International Conference on**, vol. 1. IEEE, 2000, pp. 348–353. pages 7

[18] C. Ferrari and J. Canny, “Planning optimal grasps,” in **Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on**. IEEE, 1992, pp. 2290–2295. pages 7

[19] A. Sahbani, S. El-Khoury, and P. Bidaud, “An overview of 3d object grasp synthesis algorithms,” **Robotics and Autonomous Systems**, vol. 60, no. 3, pp. 326–336, 2012. pages 7

[20] V.-D. Nguyen, “Constructing force-closure grasps,” **The International Journal of Robotics Research**, vol. 7, no. 3, pp. 3–16, 1988. pages 7

[21] Z. Li and S. S. Sastry, “Task-oriented optimal grasping by multifingered robot hands,” **IEEE Journal on Robotics and Automation**, vol. 4, no. 1, pp. 32–44, 1988. pages 7

[22] S. El-Khoury, R. De Souza, and A. Billard, “On computing task-oriented grasps,” **Robotics and Autonomous Systems**, vol. 66, pp. 145–158, 2015. pages 7, 78

[23] X. Wang, H. Rao, Y. Xiao, and Y. Zhao, “Fast force optimization of multi-fingered robotic hand grasps based on lagrange multiplier method,” in **Control Conference (CCC), 2016 35th Chinese**. IEEE, 2016, pp. 6317–6323. pages 7

[24] M. V. Liarokapis and A. M. Dollar, “Learning task-specific models for dexterous, in-hand manipulation with simple, adaptive robot hands,” in **Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on**. IEEE, 2016, pp. 2534–2541. pages 7

[25] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,” **arXiv preprint arXiv:1603.02199**, 2016. pages 7, 8

[26] I. Lenz, H. Lee, and A. Saxena, “Deep learning for detecting robotic grasps,” **The International Journal of Robotics Research**, vol. 34, no. 4-5, pp. 705–724, 2015. pages 7

[27] J. Yu, K. Weng, G. Liang, and G. Xie, “A vision-based robotic grasping system using deep learning for 3d object recognition and pose estimation,” in **Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on**. IEEE, 2013, pp. 1175–1180. pages 7

[28] A. Morales, T. Asfour, P. Azad, S. Knoop, and R. Dillmann, "Integrated grasp planning and visual object localization for a humanoid robot with five-fingered hands," in **Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on**. IEEE, 2006, pp. 5663–5668. pages 8

[29] H. Van Hoof, T. Hermans, G. Neumann, and J. Peters, "Learning robot in-hand manipulation with tactile features," in **Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on**. IEEE, 2015, pp. 121–127. pages 8, 33

[30] Y. Chebotar, O. Kroemer, and J. Peters, "Learning robot tactile sensing for object manipulation," in **Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on**. IEEE, 2014, pp. 3368–3375. pages 8

[31] L. Y. Ku, E. Learned-Miller, and R. Grupen, "Associating grasping with convolutional neural network features," **arXiv preprint arXiv:1609.03947**, 2016. pages 8

[32] A. Gupta, C. Eppner, S. Levine, and P. Abbeel, "Learning dexterous manipulation for a soft robotic hand from human demonstrations," in **Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on**. IEEE, 2016, pp. 3786–3793. pages 8

[33] T. Feix, J. Romero, H.-B. Schmiedmayer, A. M. Dollar, and D. Kragic, "The grasp taxonomy of human grasp types," **IEEE Transactions on Human-Machine Systems**, vol. 46, no. 1, pp. 66–77, 2016. pages 8, 9

[34] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," **Nature**, vol. 521, no. 7553, pp. 436–444, 2015. pages 9, 13, 15

[35] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," **arXiv preprint arXiv:1511.06434**, 2015. pages 9

[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," **Neural computation**, vol. 9, no. 8, pp. 1735–1780, 1997. pages 9

[37] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative adversarial text to image synthesis," in **Proceedings of The 33rd International Conference on Machine Learning**, vol. 3, 2016. pages 9

[38] G. Buzsaki, **Rhythms of the Brain**. Oxford University Press, 2006. pages 10

[39] BruceBlaus. Multipolar neuron. [Online]. Available: https://en.wikipedia.org/wiki/Neuron pages 10

[40] L. JACOBSON. Introduction to artificial neural networks - part 1. [Online]. Available: http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7 pages 11

[41] M. Minsky and S. Papert, "Perceptrons." 1969. pages 11

[42] chrisb. Artificial neural network. [Online]. Available: https://commons.wikimedia.org/wiki/Artificial_neural_network pages 12

[43] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” **IEEE Transactions on Neural Networks**, vol. 9, no. 5, pp. 1054–1054, 1998, iD: 1. pages 15, 21, 22, 24, 25

[44] Ucl course on rl. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html pages 15, 27

[45] C. J. C. H. Watkins, **Learning from delayed rewards**, 1989. pages 24

[46] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” **Machine learning**, vol. 8, no. 3-4, pp. 293–321, 1992. pages 26, 31

[47] Deepmind technologies. [Online]. Available: https://deepmind.com/ pages 26, 31, 35, 81

[48] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” **arXiv preprint arXiv:1511.05952**, 2015. pages 27, 29, 30, 31, 76

[49] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski **et al.**, “Human-level control through deep reinforcement learning,” **Nature**, vol. 518, no. 7540, pp. 529–533, 2015. pages 28

[50] H. V. Hasselt, “Double q-learning,” in **Advances in Neural Information Processing Systems**, 2010, pp. 2613–2621. pages 28, 31, 76

[51] csegeek.com. Sumtree. [Online]. Available: http://www.csegeek.com/csegeek/view/tutorials/algorithms/trees/tree_part8.php pages 30

[52] J. L. Elman, “Learning and development in neural networks: The importance of starting small,” **Cognition**, vol. 48, no. 1, pp. 71–99, 1993. pages 31

[53] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in **Proceedings of the 26th annual international conference on machine learning**. ACM, 2009, pp. 41–48. pages 31, 81

[54] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” **science**, vol. 313, no. 5786, pp. 504–507, 2006. pages 31

[55] D. Held, X. Geng, C. Florensa, and P. Abbeel, “Automatic goal generation for reinforcement learning agents,” **arXiv preprint arXiv:1705.06366**, 2017. pages 31, 32, 77, 81

[56] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in **Advances in neural information processing systems**, 2014, pp. 2672–2680. pages 31

[57] V. Kumar, A. Gupta, E. Todorov, and S. Levine, “Learning dexterous manipulation policies from experience and imitation,” **arXiv preprint arXiv:1611.05095**, 2016. pages 32, 40

[58] L. Y. Ku, E. G. Learned-Miller, and R. A. Grupen, “Associating grasping with convolutional neural network features,” **CoRR**, vol. abs/1609.03947, 2016. [Online]. Available: http://arxiv.org/abs/1609.03947 pages 32

[59] T. Baier-Lowenstein and J. Zhang, "Learning to grasp everyday objects using reinforcement-learning with automatic value cut-off," in **2007 IEEE/RSJ International Conference on Intelligent Robots and Systems**, 2007, pp. 1551–1556, iD: 1. pages 32

[60] O. Kroemer, C. Daniel, G. Neumann, H. van Hoof, and J. Peters, "Towards learning hierarchical skills for multi-phase manipulation tasks," in **2015 IEEE International Conference on Robotics and Automation (ICRA)**, 2015, pp. 1503–1510, iD: 1. pages 32

[61] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in **Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on**. IEEE, 2012, pp. 5026–5033. pages 35

[62] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx," in **Robotics and Automation (ICRA), 2015 IEEE International Conference on**. IEEE, 2015, pp. 4397–4404. pages 35

[63] N. Corporation. (2004) Physx physics engine. [Online]. Available: https://developer.nvidia.com/gameworks-physx-overview pages 35

[64] R. Smith. Open dynamics engine. [Online]. Available: http://www.ode.org/ pages 35

[65] G. Tech and C. M. University. Dynamic animation and robotics toolkit. [Online]. Available: https://dartsim.github.io/ pages 35

[66] R. time Physics Simulation. Bullet physics library. [Online]. Available: http://bulletphysics.org/wordpress/ pages 35

[67] H. (Microsoft). Havok. [Online]. Available: https://www.havok.com/ pages 35

[68] E. Todorov. Mujoco: Modeling, simulation and visualization of multi-joint dynamics with contact. [Online]. Available: http://www.mujoco.org/book/index.html pages 35, 40, 41, 42

[69] O. S. R. Foundation(OSRF). [Online]. Available: http://gazebosim.org/ pages 36

[70] J. F. PUGET. The most popular language for machine learning is ... [Online]. Available: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Language_Is_Best_For_Machine_Learning_And_Data_Science?lang=en pages 36

[71] G. Van Rossum and F. L. Drake Jr, **Python reference manual**. Centrum voor Wiskunde en Informatica Amsterdam, 1995. pages 36

[72] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," **Computing in Science & Engineering**, vol. 13, no. 2, pp. 22–30, 2011. pages 36

[73] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine

learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/ pages 36

[74] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang, "An introduction to computational networks and the computational network toolkit," Tech. Rep., October 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/ pages 36

[75] F. Chollet **et al.**, "Keras," https://github.com/fchollet/keras, 2015. pages 37

[76] J. H. APL. Modular prosthetic limb. [Online]. Available: http://www.jhuapl.edu/prosthetics/scientists/mpl.asp pages 40

[77] A. Foundation. Hand and wrist anatomy. [Online]. Available: http://www.arthritis.org/about-arthritis/where-it-hurts/wrist-hand-and-finger-pain/hand-wrist-anatomy.php pages 41

[78] ——. Zeromq: Distributed messaging. [Online]. Available: http://zeromq.org/ pages 52

[79] J. Mockus, "The bayesian approach to global optimization," **System Modeling and Optimization**, pp. 473–481, 1982. pages 54

[80] L. C. W. Dixon and G. P. Szegö, **Towards global optimisation**. North-Holland Amsterdam, 1978. pages 54

[81] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "Boa: The bayesian optimization algorithm," in **Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1**. Morgan Kaufmann Publishers Inc., 1999, pp. 525–532. pages 54

[82] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in **Advances in neural information processing systems**, 2012, pp. 2951–2959. pages 54

[83] C. E. Rasmussen, "Gaussian processes for machine learning," 2006. pages 54