Learning to Grasp Objects with Reinforcement Learning

Rik Timmers

March 14, 2018

Master's Thesis

Artificial Intelligence
University of Groningen, The Netherlands

First supervisor:
Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
Second supervisor:
Prof. Dr. Lambert Schomaker (Artificial Intelligence, University of Groningen)

university of groningen / faculty of mathematics and natural sciences

Abstract

In this project we will use reinforcement learning, the CACLA algorithm, to let an agent learn to control a robotic arm. Inspired by domestic service robots that have to perform multiple complex tasks, manipulation is only a small part of it. Using neural networks an agent should be able to learn to complete a manipulation task without having to calculate paths and grasping points. We will be using a 6 degree of freedom robotica arm, Mico, and make use of a simulator called V-REP to perform the experiments. We compare the results to a traditional simple inverse kinematic solver to see if there is a gain in speed, accuracy or robustness. Whilst most agents use one neural network to perform their task, we will experiment with different architectures, namely the amount of neural networks that each control a sub-set of the joints, to see if this can improve results. Whilst for reinforcement learning exploration is very important we test two different exploration methods; Gaussian exploration and Ornstein-Uhlenbeck process exploration, to see if there is any influence in the training. We experimented first with letting the end effector of the arm move to a certain position without grasping an object. It was shown that when using only 1 joint learning is very easy, but when controlling more joints the problem of simply going to a single location becomes more difficult to solve. While adding more training iterations can improve results, it also takes a lot longer to train the neural networks. By showing a pre training stage consisting of calculating the forward kinematics without relying on any physics simulation to create the input state of the agent, we can create more examples to learn from and improve results and decrease the learning time. However when trying to grasp objects the extra pre training stage does not help at all. By increasing the training iterations we can achieve some good results and the agent is able to learn to grasp an object. However when using multiple networks to control a sub-set of joints we can improve on the results, even reaching a 100% success rate for both exploration methods, not only showing that multiple networks can outperform a single network, also that exploration does not influence training all that much. The downside is that training takes a very long time. Whilst it does not outperform the inverse kinematic solver we do have to take into account that the setup was relatively easy, therefore making it very easy for the inverse kinematic solver.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Robots are becoming more common in our daily lives and the industry for creating robots is only growing more according to the International Federation of Robotics [19]. Since the late 70's the industrial robot has become quite important for faster production in factories, whilst robots that can be used outside of factories like the autonomous car [26], autonomous drones [11], security drones [10] and robots that can help in hospitals [29] are under development or even already being used. Whilst fully autonomous cars are still in development, an autonomous driving assistant [9] is already available and used in cars from Tesla [1]. A lot of research is also being done on domestic service robots, these robots should be able to help people with household tasks, or in hospitals and elderly care [5]. Domestic service robots should be able to perform lots of complex tasks: navigating, speech recognition, object recognition and manipulation. All this has to be done safely and fully autonomously. To keep track of the development of domestic service robots there is a competition, Robocup@home [47], that is used as a benchmark. The Robocup@home competition exists of different challenges that test different aspects of a domestic service robot. A reoccurring part in the challenges is manipulation: grasping an object and putting it down somewhere else, put objects in a cupboard with different height shelves, opening doors, prepare breakfast and turn off the television via the remote.

1.2 Project and Motivation

This project will focus on the manipulation part, mainly on picking up objects with a 6 Degree of Freedom (DoF) robotic arm. The objects that need to be picked up will have simple shapes because of the limited complexity of the robotic arm's end effector, a two finger underactuated gripper. Instead of using Cartesian control or planners that use inverse kinematics to control

the arm's motion, we will make use of neural networks to control the arm. Neural networks are capable of learning, the idea is that a neural network can learn to complete complex tasks without having to calculate paths and grasping points whilst also being able to handle unseen situations. Since manipulation tasks are getting more difficult, like using a remote to turn off a television, it would be better to be able to learn how to complete tasks instead of having to program an approach on how to solve the task.

In this project the arm, or rather an agent, will have to learn how to grasp an object by making use of Reinforcement Learning (RL) [42]. For this project we want to compare how a simple inverse kinematic planner performs versus an agent that has learned how to grasp objects. For the inverse kinematic solution we will make use of the MoveIt [40] package. MoveIt can plan collision free paths but it can not plan on how to grasp an object. The operator needs to tell the planner how to grasp an object by providing possible grasp angles. Giving multiple grasping angles is needed because not every grasp angle is reachable by the arm and the arm could collide with other objects in the scene on certain angles. Giving more possible grasp angles means more computation time for planning a path to grasp the object. By using RL we can let the agent learn how to perform the grasp and therefore when the training is done it is able to execute the grasping motion without the need for an operator to specify how to grasp it, nor does it need to calculate a path.

1.3 Research Questions

We will focus on neural networks that can control the robotic arm. We will make use of the Mico [21] robotic arm, with 6 DoF and a two finger gripper. We will perform the training and testing in a simulator called V-REP [35]. For RL we will make use of the CACLA [45] algorithm, it will try to learn velocity commands for each of the actuators so it can grasp an object. In order to test the performance we will compare the controller to a simple inverse kinematic planner with simple grasping suggestions using MoveIt. Since the arm itself does not contain any force sensors in its fingers and the fingers are underactuated, we cannot get any feedback during grasping and therefore this will make learning more difficult.

Another aspect is the architecture of the neural network, commonly only one neural network is used to learn a mapping from inputs to actions. In this project we will look at different architectures for controlling the arm, instead of one neural network we can have multiple neural networks each controlling a set of joints. This way we can change parameters for each of the networks separately to possibly improve training time or the performance.

Exploration is very important in reinforcement learning, without it learning can not be done. Since exploration influences how well an agent can learn we will explore two different methods for exploration to see if this influences the training.

Training will be done in simulation because training on a real arm would require constant supervision of a human operator that can reset the environment. It will also take a lot of time since the arm cannot operate faster than real time, furthermore only one arm is available therefore only one set of parameters can be tested at the time.

The main research question is:

- How does reinforcement learning compare to a much simpler inverse kinematic controller for grasping objects? Is there a gain in speed, accuracy or robustness?

The sub-questions:

- Which neural network architecture results in the best performance?
- How does the exploration method influence the training?

Chapter 2

State of the Art

In this chapter we will describe different methods that are used to solve manipulation tasks. We will discuss a trajectory planner, supervised and unsupervised machine learning methods, and reinforcement learning methods that are used for performing manipulation tasks with a robotic arm.

2.1 Trajectory Planner

In order for complex movements and obstacle avoidance, a trajectory should be created that the end effector should follow. In order to get information about the environment a sensor is needed, this allows the planner to plan a collision free path. We will make use of MoveIt [40], a framework for 3D perception and manipulation control. MoveIt makes use of the Open Motion Planning Library (OMPL) [41], an open-source motion planning library that primarily uses randomized motion planners.

2.1.1 URDF

The Unified Robot Description Format (URDF) [12] is used by MoveIt to represent a robot model. The URDF describes the robot's geometry and kinematics, which are used for self-collision checking and inverse kinematics calculations. It is important that the details in the URDF are precise and match the real world model. Since planning is based on the URDF model and not based on the real world model, executing a trajectory with wrong parameters in the URDF will result in wrong behaviour on the real robot.

2.1.2 RRT-Connect

The OMPL has lots of different trajectory planners, but we will only make use of one; the Rapidly-exploring Random Trees Connect (RRT-Connect) [23] path planner because in a simple benchmark it performs the best in both planning time and solving result. The planner needs to know about

its environment in order to plan collision free paths. The environment can either be in 2D or 3D. In our case MoveIt will provide the environment to OMPL which provides it to the planners. The RRT-Connect is an extension of the RRT [24] path planner. RRT tries to create a tree between the starting point, q_{init}, and the goal point, q_{goal}. In order to expand the tree a random point, q, is chosen from the search space. The random sampling of the point q prefers to explore large unsearched areas. When a point q is chosen, the nearest point in the tree, q_{near}, is found and a connection is attempted to be created between those two points by a vector pointing from q_{near} to q. The new connection however is limited to a certain distance, ϵ. A new point, q_{new}, is added to the tree with a distance ϵ from q_{near}, if q_{new} does not collide with an obstacle by checking if the new path does not intersect with an obstacle in the environment.

RRT-Connect makes use of two trees, one starting from q_{init}, and one starting from q_{goal}. For each tree a q is also chosen, but instead of one ϵ step, it will repeat the distance step until it has reached q or reaches an obstacle. At each step a tree will attempt to connect to the other tree by finding the q_{near} of the other tree, if a collision free path can be found it will be connected and a path is found.

2.1.3 MoveIt

MoveIt is the framework that is used for combining the different packages that are needed in order for making trajectories that a manipulator has to perform. MoveIt creates a planning scene which represents the world and the current state of the robot, it can make use of 3D sensors or manual input to create a model of the world. It makes use of the URDF model and actual robot joint positions to place the URDF model into the current state of the robot, the initial starting position. MoveIt also configures the OMPL planner so that it has a representation of the arm. The planner will use the end-effector for trajectory planning, an inverse kinematic solver will tell if a valid solution exists for a certain position and orientation of the end-effector. Collision detection is provided by the Flexible Collision Library (FCL) [32]. The collision detection makes use of an Allowed Collision Matrix (ACM) to reduce the amount of collision check operations. The ACM has a value for each pair of bodies in the world, such as parts of the robot itself, the table and objects that are on the table. If a pair of bodies are never able to collide, for example the body of an arm that is fixed and an object on a table, the ACM sets a value for the two bodies so that they are never checked, therefore saving computing time. The OMPL planners do not take into account maximum velocities and accelerations of the joints, therefore MoveIt will process the path and generate a trajectory that is properly time-parameterized accounting for the maximum velocity and acceleration limit

that each joint has. The final processing step will create smooth trajectories.

MoveIt does not have an object recognition package thus in order to know where the object is an external package is needed, or manual input is required. Since MoveIt does not know about objects, it will see everything as an obstacle. It is therefore needed to place a simple geometric shape that resembles the object into the planning scene where the actual object is. This simple geometric shape can then be set to allow some collision when attempting to grasp, but it can also be used, when actually grasped, to plan a trajectory that takes into account the end-effector holding the object so that it does not collide with other obstacles or the arm itself.

MoveIt itself does not know how to grasp objects. The operator needs to tell (via programming) how an object should be picked up, for example from which angle does the end-effector approach the object, how close should the end effector approach the object and in which direction the end effector should move when it has grasped the object. All these parameters and more need to be set by the operator and depend on different scenarios as well. Each object needs to be picked up differently, and after grasping the motion depends on the environment. Picking up an object on a table will allow for an upwards motion after grasping, picking up an object from a closet with shelves will not always allow for an upwards motion after grasping. All these parameters need to be programmed, MoveIt will not do this itself.

2.2 Supervised Learning

Supervised Learning is a form of machine learning that makes use of labeled training data. Labeled data can come from human experts, but also from agents that can create the labeled data. Supervised Learning is often used for classification tasks, like object recognition [15]. For classification a training data set is usually created with annotated data, and after training the classifier should be able to classify new data correctly. This can also be used to classify images on where to grasp a certain object. Learning from Demonstration (LfD) is also a supervised method, where a human expert can show a robotic arm how to grasp an object. This motion can be recorded and used for training.

2.2.1 2D Images for Grasp Localization

In [37] the authors use 2D images of objects and label them on where a gripper should perform a grasp. Since creating a training data set manually would take a lot of time, they make use of synthetic data. They create 3D models of objects and use ray tracing [16], a standard image rendering method, to create a 2D image of that object. By changing the camera

position, adding different colors and textures they can create lots of different examples of the same object. For each object they only need to specify the grasping area once instead of labeling each image separately. With this data set of labeled data a logistic regression model is trained that can give the probability of a good grasping point at a 2D position in an image. After supervised training is done the classifier can only classify in 2D space where to grasp an object. The robot takes at least two images from different orientations and determines the 3D grasping location from it. Once the grasping location is determined the robot makes use of a planner to create a path and grasping motion. The authors trained the logistic regression model with only 5 synthetic objects, resulting in a total of 2,500 examples for training. The tests were performed with a real robotic arm and with 14 real objects, 5 objects that are similar to the objects used for training, and 9 objects that were different from the training objects. The similar objects have an average of 1.80cm in grasp position error and a success rate of 90% when grasping the object. The new objects have an average of 1.86cm in grasp position error and a success rate of 87.8% when grasping.

2.2.2 Depth Segmentation for Grasping Objects

The authors of [34] also make use of Supervised Learning to determine possible grasping places of objects by making use of depth information. Instead of only choosing one grasp point, they determine all possible grasp positions on an object. The robot makes use of a 3D camera and looks for specific segments and tries to determine with a classifier if these segments are good for grasping the object. The classifier makes use of both color and depth information. The classifier makes use of three different features. Geometric 2D features; for each segment they compute the width, height and area. Color image features; an RGB image is converted to a LAB image format which is less sensitive to lighting conditions. The variance in L, A and B channels are used as feature. Geometric 3D features; The variance in depth, height and range of height from segments. The authors' intuition is that ungraspable segments tend to be either very small or very large and tend to have lesser variation in color composition. With these features a Support Vector Machine [6] is trained to classify segments as graspable or not. From all possible graspable segments they select the segment to grasp that is closest to the robot but also farthest away from other graspable objects in the scene. From a graspable segment a mesh is created, on that mesh sampling is performed to find contact points that are feasible for grasping. The authors trained the classifier with 200 scenes of 6 objects. The testing was performed on a real robot where objects were placed on a table and the robot was placed at different positions with respect to the table. The robot tried to grasp 8 different objects, resulting in a total grasping success rate of 87.5%.

2.2.3 Learning from Demonstration

Learning from Demonstration (LfD) can be seen as a form of Supervised Learning. LfD makes use of example joint trajectories instead of labeled data. A teacher can show a robot how a job is performed by manually moving the end effector of the robotic arm. The data is recorded and a policy can be learned so that the robot knows how to execute the behaviour on its own. The idea behind LfD is that it does not require an expert in machine learning or robotics to make a robot learn specific tasks.

In [38] the authors use a system that makes use of both speech and gestures to learn from. With a vision system the robot can focus on a region that the instructor points at, and also perform object detection. With speech the instructor can give simple descriptions like 'the red cube' or 'in front of'. For grasping the robot arm makes use of a camera mounted on the arm, so it can move itself above an object. Grasping is done by slowly wrapping the fingers around the object and using force feedback to determine if the fingers are touching the object to be grasped. They also suggest a way to learn grasping by imitation. By transforming hand postures to joint angle space of the robot hand, they can learn how a robot hand should grasp objects by just letting the instructor grasp objects with his/her own hands. In [46] the authors make use of a glove to register hand motions, including grasping and the forces that are required to grasp an object. The authors performed a test with the peg-in-hole task where the goal is to put an object, the peg, inside a cylindrical object, the hole. The learned task was performed by a robot arm with a 4 finger gripper, which was able to perform the task 70% of the time.

2.3 Unsupervised Learning

Unsupervised Learning is a form of machine learning that makes use of unlabeled training data. A common Unsupervised Learning algorithm is k-means [27] for clustering data. It can cluster data without labels into k clusters. Unsupervised Learning can be used to let a robot learn about the environment but also to let it learn how to grasp objects.

2.3.1 Object Affordance

The word affordance was first used by perceptual psychologist J.J. Gibson [13][14] to refer to the actionable properties between the world and an actor. Affordances are relationships [31] that suggest how an object may be interacted with. In [44] the authors use Unsupervised Learning to let a robot learn about object affordances. The robot can perform different fixed behaviours, like move forward, rotate, and a lift behaviour. The robot learns

from interacting with the environment, the robot can collide for example with a box and it would learn that the box doesn't move, if the robot collides with a sphere, the sphere would roll away and clear a path. By using a robotic arm with a magnet the robot can try and lift objects to see if they are graspable. When interacting with the environment it creates relation instances; the effect, the initial feature vector that comes from a 3D range scanner representing the environment, and the executed behaviour. The effect is clustered using k-means and for each cluster an effect-id is assigned. The relationship between objects and the effects created by a given behaviour are learned with a Support Vector Machine [6] classifier. These learned affordance relations can be used during planning to see if an action is possible. The authors used a simulator to train the classifier and tested it on the real robot. One object was placed in front of the robot and the robot performed the behaviour to either move or lift the object. The robot was able to perform the correct action 90% of the time.

2.3.2 Grasping

Manually labeling data will take a lot of time, not only because objects can be grasped in multiple ways, it also needs different viewpoints of the object. In [33] a robot is used to create the data set. A human is only needed to start the process and place objects on the table in an arbitrary manner. The robot builds a data set by trial and error. It finds a region of interest, an object, on the table and moves the end effector 25cm above the object. The end effector has a camera which is now pointing at the object, the arm selects a random point to use as grasping point and an angle to orientate the gripper. The robot then attempts to grasp the object, the object is raised 20cm and annotated as a success or failure depending on the gripper's force sensor readings. All the images, trajectories and gripping outcomes are recorded. They collected data for 150 objects, resulting in a data set of 50,000 grasp experiences, consisting of over 700 hours of robot time. A Convolutional Neural Network [22] was trained with a region around the grasping point as input, as output they train an 18-dimensional likelihood vector. Each output represents a grasp angle from 0° to 170° with 10° steps. The output will indicate if the angle is a good grasping angle or not. To test the system the robot tried 150 times to grasp an unseen object, and 150 times a previously seen object. For the unseen objects the robot was able to grasp and raise the object 66% of the time, for the previously seen objects the robot was able to grasp and raise it 73% of the time.

2.4 Reinforcement Learning

Reinforcement Learning (RL)[42] is a form of machine learning that does not make use of a data set, but allows an agent to learn which actions to take

in an unknown environment so that it receives a maximum sum of rewards. RL can be used to let a robotic arm grasp an object, or open a door.

2.4.1 Grasping under Uncertainty

In [39] the authors make use of RL to learn how to grasp an object under uncertainty. Instead of learning to grasp an object from a specific location they take into account that the object might not be at the exact location because of inaccuracies of a sensor for example. They acquire movement primitives for their arm as Dynamic Movement Primitives [20], that are initialized with a trajectory using LfD and a grasp posture that is generated with a grasp planner, GraspIt! [28]. The Dynamic Movement Primitives are optimized for grasping with RL, where the reward function only rewards a successful grasp. To learn the Dynamic Movement Primitives they randomly sample the actual position of the object from a probability distribution that represents a model of the uncertainty in the object's pose. With the Policy Improvement with Path Integrals [43] algorithm the arm is able to learn robust grasp trajectories that allows for grasping where the object is not always in the same exact location. Experiments were done in simulation, the arm had to grasp a cylinder that could be placed with a deviation of 5cm from its original position. It was able to grasp the object 100% of the time. The arm also had to learn to grasp more complex objects. The objects were made of boxes so that the object would represent the letters 'T' and 'H'. For each experiment they initialized the grasping position to a different part of the object, resulting in five different grasps. For three grasps the success rate was 95%, while for two grasps the success rate was 60%.

2.4.2 Opening a Door

Whilst often robots are trained in simulation, in [17] the authors gather data directly from real physical robots. The authors want to train a robotic arm so that it is able to open a door. Training was done with the Normalized Advantage Function [18] algorithm. For training the authors make use of a replay buffer to perform asynchronous learning. The asynchronous learning algorithm was proposed by [30] to speed up simulated experiments. Since a real physical robot can not operate faster than real time, they make use of multiple robot arms to train the same network. A training thread trains on a replay buffer, while for each physical robot arm a worker thread is used to collect data. A worker thread gets the latest network for controlling the arm at the beginning of its run, and then it executes actions based on the network, with some exploration added to it. The worker thread records the state, reward, action and new state so that it can be sent to the replay buffer. One worker would require more than 4 hours to achieve 100% success rate, 2 workers can achieve a 100% success rate in 2.5 hours of training time.

Chapter 3

Theoretical Framework

In this chapter we will describe the theoretical framework used in this project. We discuss Forward and Inverse Kinematics, used to determine the position of a robotic arm end effector or joint positions. We also discuss Reinforcement Learning in general, and CACLA which is the method used in this project to let a robotica arm learn to grasp an object via Reinforcement Learning. Multilayer perceptrons are also discussed as they are used for letting the agent learn. We discuss two different exploration methods and the use of a replay buffer to help reduce training time.

3.1 Forward Kinematics

Forward Kinematics (FK) [7] uses kinematic equations to compute the end effector position of a robotic arm given the joint states. A robotic arm can be seen as a set of joints connected with each other via links. These links can be described by a kinematic function so that the relation between two joints is fixed. To describe the link kinematically four values are needed, two describe the link itself, and two describe the link's connection to a neighboring link. This convention of describing the parameters is called the Denavit-Hartenberg notation [8] (DH). The four parameters are as follow (the following notations are from [7]), see also Figure 3.1:

a_i : the distance from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured along the $\hat{X}_i$

α_i : the angle from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured about $\hat{X}_i$

d_i : the distance from $\hat{X}_{i-1}$ to $\hat{X}_i$ measured along $\hat{Z}_i$

θ_i : the angle from $\hat{X}_{i-1}$ to $\hat{X}_i$ measured about $\hat{Z}_i$

With these parameters a transformation matrix (Equation 3.1) can describe the transformation from joint $i-1$ to joint i. To calculate the transfor-

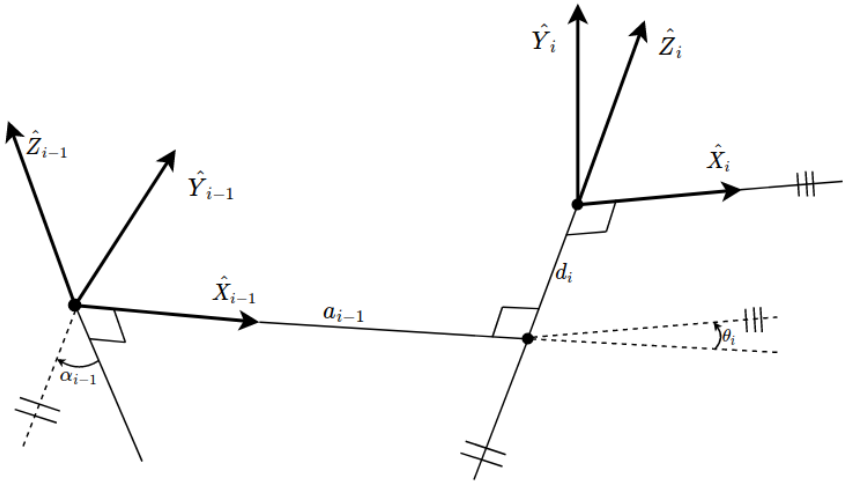


Figure 3.1: The transformations from frame $i-1$ to frame i.

mation from joint 2 with respect to the origin two transformation matrices, ${}^0_1T$ and ${}^1_2T$ need to be multiplied in order to create ${}^0_2T$.

$$ {}^{i-1}_{i}T = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & a_{i-1} \\ \sin(\theta_i)\cos(\alpha_{i-1}) & \cos(\theta_i)\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{i-1})d_i \\ \sin(\theta_i)\sin(\alpha_{i-1}) & \cos(\theta_i)\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & \cos(\alpha_{i-1})d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1} $$

To compute the end effector position we need to multiply the transformation matrices for each link that is part of the robotic arm. In Equation (3.2) the matrix ${}^0_eT$ contains the position of the end effector. The coordinates can be read from ${}^0_eT$ at positions [1,4] for x, [2,4] for y, and [3,4] for z, see Equation (3.3).

$$ {}^0_eT = {}^0_1T\,{}^1_2T\,{}^2_3T\,{}^3_4T\,{}^4_5T\,{}^5_6T \tag{3.2} $$

$$ {}^0_eT = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3} $$

3.2 Inverse Kinematics

Inverse Kinematics (IK) [7] is the reverse from FK, IK computes the joint positions of a robotic arm given the end effector position and orientation. This can be used for motion planning where the planner plans the trajectory of the end effector. The IK will provide, when possible, the solutions for the joint positions. IK solutions can only be found when the end effector is within the reachable workspace of the arm.

Two common approaches for solving IK problems are algebraic and geometric solutions. Algebraic approaches make use of the transformation matrix, like Equation (3.3), in order to find the joint angles. By providing the required position and orientation 6 constants are known which can be used to solve the equations. The main idea of the algebraic approach is manipulating these equations into a form for which a solution is known. In the geometric approach the spatial geometry of the arm is decomposed into several plane-geometry problems. On a plane the links of the arm create a triangle which can be used to find the angles of the joints. A problem in IK is that sometimes there are multiple solutions available, a valid solution for this would be to find for each current joint angle the closest solution, this would mean that the joint does not have to move as much.

3.3 Reinforcement Learning

In Reinforcement Learning (RL) [3] a decision maker tries to learn to complete a task in an environment. The decision maker, called an agent, can interact with the environment by performing actions. The agent perceives the environment as a state in which it can make an action that will modify its state. When the agent takes an action the environment provides a reward from which the agent can learn. RL makes use of a critic that tells the agent how well it has been performing.

The following notation is used from [3]. Time is discrete as $t = 0, 1, 2, ...$, and states are defined as $s_t \in \mathcal{S}$, where s_t is the state in which the agent is at time t, $\mathcal{S}$ is the set of all possible states. The actions are defined as $a_t \in \mathcal{A}(s_t)$, where a_t is the action that the agent takes at time t, $\mathcal{A}(s_t)$ is the set of possible actions when the agent is in state s_t. An agent gets a reward $r_{t+1} \in \mathcal{R}$ when going from s_t to s_{t+1}, where $\mathcal{R}$ is the set of all possible rewards. The RL problem is modeled using a Markov decision process, the reward and next state are sampled from their probability distributions, $p(r_{t+1}|s_t, a_t)$ and $p(s_{t+1}|s_t, a_t)$.

A policy, π, defines the agent's behavior and is a mapping from state

to action: $\pi : \mathcal{S} \rightarrow A$. The value of a policy, $\mathcal{V}^{\pi}(s_t)$, is the expected cumulative reward that will be received when the agent follows the policy, starting from state s_t. A discount factor, $0 < \gamma < 1$, is used to determine the influence of rewards that happen in the future. A small discount factor indicates that only rewards in the immediate future are being considered, a larger discount factor prioritizes rewards in the distant future. For each policy π there is a $\mathcal{V}^{\pi}(s_t)$, therefore also an optimal policy π^* exists such that $\mathcal{V}^*(s_t) = \underset{\pi}{max}\mathcal{V}^{\pi}(s_t), \forall s_t$. Instead of using the $\mathcal{V}(s_t)$ it is also possible to use the values of state-action pairs, $\mathcal{Q}(s_t, a_t)$, which tells how good it is to perform action a_t when the agent is in state s_t. The definition of the optimal $\mathcal{Q}(s_t, a_t)$ is known as the Bellman's equation [4] (3.4). With $\mathcal{Q}^*(s_t, a_t)$ a policy π can be defined as taking the action a_t^*, which has the highest value among all $\mathcal{Q}^*(s_t, a_t)$.

$$\mathcal{Q}^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} \mathcal{P}(s_{t+1}|s_t, a_t) \max_{a_{t+1}} \mathcal{Q}^*(s_{t+1}, a_{t+1}) \tag{3.4}$$

When the model is not known beforehand exploration is required to find values and rewards for new states, with this information the value for the current state can be updated. These algorithms are called temporal difference algorithms because they look at the difference between the current estimate value and the discounted value of the next state. When the model is deterministic there is no probability of moving to a certain state, therefore the Equation of (3.4) can be reduced to (3.5). A reward is received performing an action, which gets the agent in a new state.

$$\mathcal{Q}(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} \mathcal{Q}(s_{t+1}, a_{t+1}) \tag{3.5}$$

The actions are discrete, meaning that they represent a certain action that the agent can perform in the environment. The policy will select the action and the agent will then perform this action, which could represent moving the agent one grid upwards in a grid-based environment.

3.4 CACLA

The Continuous Actor Critic Learning Automaton (CACLA) [45] is able to handle continuous actions. A continuous action can directly be performed by the agent, with or without scaling the action. Whilst with discrete actions only one action is chosen, with continuous actions the agents performs all actions.

CACLA uses the Actor-Critic method. An Actor is the policy of the agent, it gives the actions that the agent should perform, the Critic is the

value of the current state and gives the expected cumulative reward given the state. For both the Actor and Critic a multilayer perceptron (MLP) can be used to train the corresponding action and value outputs. In Equation 3.6 the temporal difference error is given, $\mathcal{V}$ is the output of the Critic MLP.

$$\delta_t = r_{t+1} + \gamma \mathcal{V}(s_{t+1}) - \mathcal{V}(s_t) \tag{3.6}$$

The Critic is trained by given the target value of the current output plus the temporal difference error (Equation 3.7). The Actor is only trained when the temporal difference error is larger than zero, in this way the Actor only learns from actions that benefit the agent and not from actions that would make it perform worse. The Actor is updated by taking s_t as input and train on the a_t plus exploration noise (Equation 3.8), where g_t is the exploration noise. The CACLA algorithm also has the benefit of being invariant to scaling of the reward function.

$$\tau_c = \mathcal{V}(s_t) + \delta_t \tag{3.7}$$

$$\tau_a = a_t + g_t \quad ; \quad if \quad \delta_t > 0 \tag{3.8}$$

3.5 Multilayer Perceptron

The multilayer perceptron (MLP) [3] is an artificial neural network structure that can be used for classification and regression problems. Neural networks are built from perceptrons, a perceptron is a processing unit which has inputs and an output. A weight is connected to each input, and the output of a perceptron is the summation of the input value times the weight value plus a bias value (Equation 3.9), notation used from [3]. w_j refers to the weight value that is connected to the jth input value x_j, y is called the activation of the perceptron.

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \tag{3.9}$$

To make a perceptron learn, the weight values need to be updated. One way of training is online learning, where learning is performed on a single example with the target output at a time. Batch learning is making use of multiple examples, a batch, and calculates the average error over that batch to update the network. An error function can be defined from which the weight values can be updated. To train for example a single perceptron an error function can be defined (Equation 3.10), where r is the expected target

value and y the output of the activation function (Equation 3.9) based on the input x and weights w.

$$E(w|x,r) = \frac{1}{2}(r-y)^2 \tag{3.10}$$

In order to update the weights we can make use of stochastic gradient descent, as seen in Equation (3.11). The weights are updated by the error given the target output and actual output, η is the learning rate which dictates how large the update step is.

$$\Delta w_j = \eta(r-y)x_j \tag{3.11}$$

An MLP consists of multiple perceptrons, ordered in layers (Figure 3.2). The first layer is the input layer which gives the input values for the network, these input values represent data from which the neural network needs to produce correct output. The final layer is the output layer which represents the output of the neural network. In between are one or more hidden layers, which allow the neural network to learn nonlinear functions by making use of nonlinear activation functions such as the sigmoid function and hyperbolic tangent function.

Training an MLP is similar to a single perceptron, the weights for the connection between the hidden layer and the output can be updated with Equation (3.11), where the input is now defined as the output of the hidden layer. In order to update the weights from the input layer to the hidden layer we need to apply the chain rule (3.12), where w_{hj} are the weights from the input to the hidden layer, z_h the output of the hidden layer and y_i the output of the output layer. The process is called backpropagation [36], as it propagates the error from the output y back to the weights of the input.

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}} \tag{3.12}$$

3.6 Exploration

Exploration is needed for an agent to learn, since the agent does not know anything about the world and how to achieve its goal. Exploration is the process of choosing a random action, or adding noise to a continuous output. In this project we will use two different exploration methods, Gaussian noise and Ornstein-Uhlenbeck process (OUP) exploration.

Gaussian exploration makes use of the Gaussian function (Equation 3.13) to sample random noise, μ defines the expected value, or center value, and σ defines the standard deviation. To explore, the μ value is set to the current

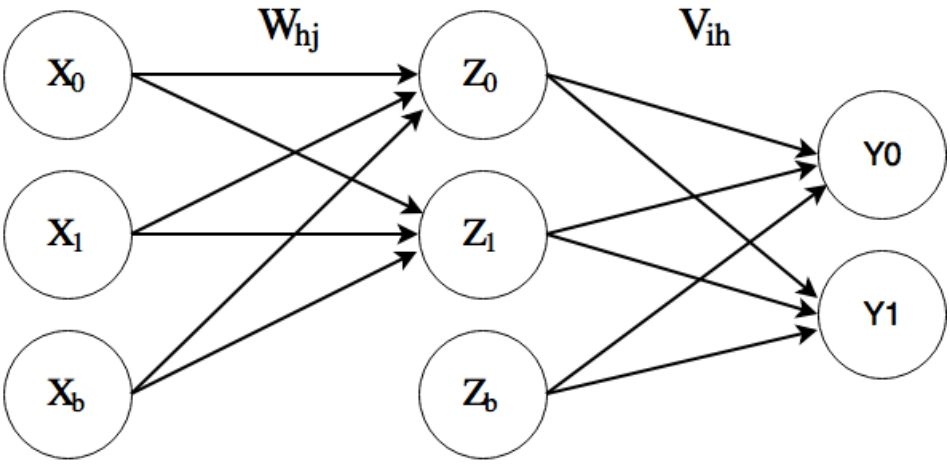


Figure 3.2: A multilayer perceptron with 2 inputs and one bias value, 2 hidden units and one bias value, and 2 outputs. W_{hj} are the weights from the input to the hidden layer, V_{ih} are the weights from the hidden layer to the output layer. X_i is the input value, Z_i is the output of the hidden unit, and Y_i is the output of the output unit.

value of the action, the Gaussian function then samples around this value with σ, the value that comes out of $g(x)$ is added to the current action value to create a new action value. With Gaussian exploration the agent explores around its current policy. By reducing the σ the agent will start to exploit its policy, instead of mainly exploring. To reduce the σ a decay rate can be set. If the σ would never decrease the agent would only be exploring and never perform its learned policy.

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{3.13}$$

OUP exploration takes into account the previous value of the action when exploring [25]. The idea behind this exploration method is that continuous actions are most likely to be correlated to the previous and next time steps. The OUP models the velocity of a Brownian particle with friction, which results in temporally correlated values centered around 0. Whilst not taking into account for the physics quantities it can be modeled as Equation (3.14). $\mathcal{K}$ is the decay factor, $\mathcal{K} \in [0, 1]$, and g_t refers to Equation (3.13) with $\mu = 0$. The previous exploration values are taken into account when exploring, making the actions rely more on previous explored actions.

$$o_t = \mathcal{K} o_{t-1} + g_t \tag{3.14}$$

3.7 Replay Buffer

When an agent is running in a physics simulator, or even in the real world, the rate at which an agent can learn is limited to how fast it can gather training examples. With online learning the rate is also depending on how fast the MLPs can perform the feed-forward calculation and how fast a training step is. Algorithm 1 shows the steps that are needed to do a single online learning step for updating the networks. Line 2, 7, and 8 are feed-forward passes of the Actor and Critic neural networks. $TrainActor$ and $TrainCritic$ are the update functions for the MLPs, where the first parameter is the input and the second parameter the target value.

Algorithm 1 Online learning single time step

1: $s_t = GetState()$
2: $a_t = \pi(s_t)$
3: $a_t = a_t + g_t$
4: $PerformAction(a_t)$
5: $s_{t+1} = GetState()$
6: $r_{t+1} = GetReward()$
7: $v_t = V(s_t)$
8: $v_{t+1} = V(s_{t+1})$
9: $\delta_t = r_{t+1} + \gamma v_{t+1} - v_t$
10: **if** $\delta_t > 0$ **then**
11: $\quad TrainActor(s_t, a_t)$
12: $TrainCritic(s_t, v_t + \delta_t)$

In order to not depend on the rate at which the agent can update the network, a training tuple $(s_t, a_t, s_{t+1}, r_{t+1})$ can be sent to a replay buffer. The replay buffer is a first-in-first-out buffer, meaning the first tuple that is added to the buffer is also the first tuple that is removed from the buffer. The buffer will only hold a limited amount of tuples, where new tuples will push-out the older tuples. With this replay buffer a separate thread can train the Actor and Critic MLPs without having to wait for the agent to perform its actions. The training thread can take a mini-batch, instead of just one training tuple, from the replay buffer to reduce training time. The mini-batch is sampled at random from the replay buffer, and it is therefore possible that the networks are trained on the same training tuple multiple times, but the output of the MLPs will be different at a later t because the MLPs have been trained with new examples. The agent can now use the examples faster because it does not have to perform multiple MLP operations, the only thing that the agent needs before it starts a trial is the newest version of the Actor network so it can perform the actions. During a trial

the agent will use the same Actor network that is not being updated, however when it starts a new trial the Actor network has been updated by the training thread many more times than it would have during online learning, this should provide the agent with a better Actor network for the next trial.

Even with the replay buffer the agent is sometimes still very slow in generating training tuples, therefore also limiting the exploration that it can do. By adding more agents the replay buffer can be filled with more diverse examples of different possible training tuples, allowing to find the optimal policy faster. Each agent receives the newest Actor MLP, adds its own exploration to the actions and pushes its training tuple to the same replay buffer.

Chapter 4

Experimental Setup

In this chapter we will discuss the experimental setup that is used for this project. We will describe the arm used for training, the simulation, different architectures we want to test and different exploration methods.

4.1 Kinova Mico Arm

In this project we use the Mico arm [21] from Kinova Robotics. We use the 6 Degree of Freedom (DoF) version of the arm, with a gripper that has 2 fingers. The fingers are underactuated, and do not contain any sensors. Each finger consists of two segments, see Figure 4.1(a), where the bottom segment is attached to a linear actuator, and the top segment is attached to the bottom segment and can move with a spring mechanism. When closing the fingers the actuators will pull the bottom segments toward each other, Figures 4.1(b)(c). When the bottom segment gets blocked by an object, the top segment will start to move closer because of the spring, Figures 4.1(e)(f). Because of the underactuated fingers it is difficult to determine the finger positions. Whether the fingers hold an object or not, when the fingers are fully closed the position of the actuators are very similar making it impossible to determine via position of the fingers whether it has grasped something. It is also difficult to grasp smaller objects with just the finger tips, the underactuated fingers are not able to put a lot of force on the finger tips when holding for example a pencil. The fingers work best when grasping larger objects where the fingers try to wrap around the object.

The real robotic arm has a link that determines the position for the end effector. The position of the end effector is determined by forward kinematics and depends on the joint positions. This end effector link is not available in the simulation model, it is therefore added into the model. In table 4.1 the DH parameters are given for the real robotic arm.

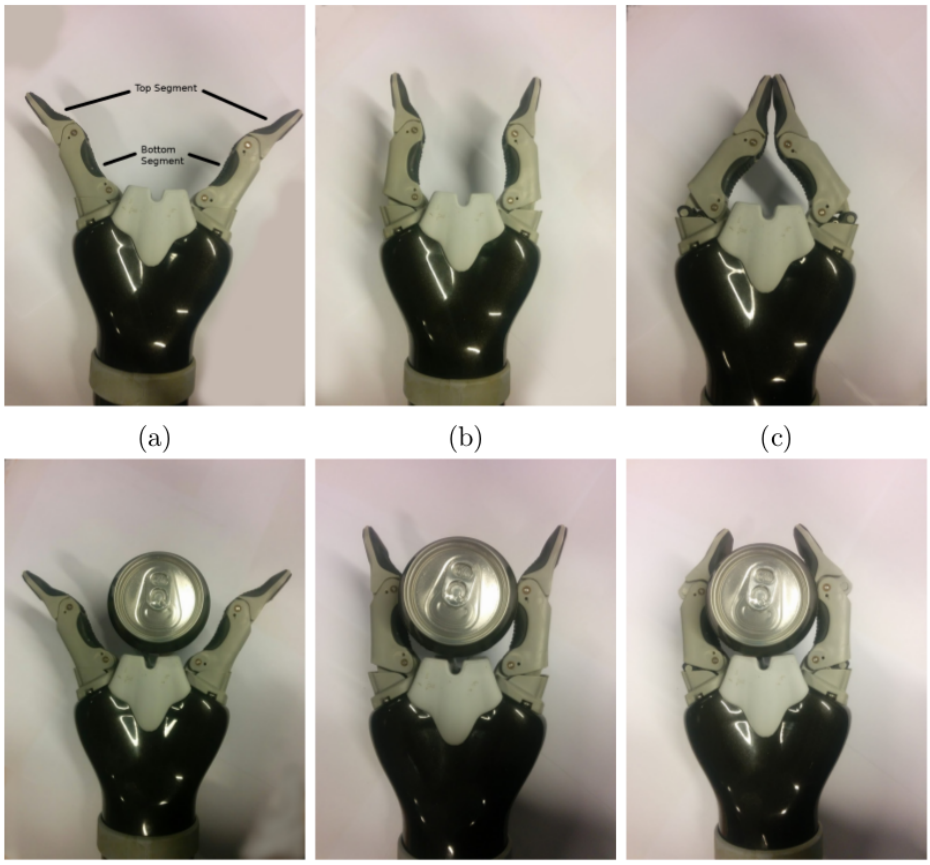


Figure 4.1: The Mico end effector with underactuated fingers. (a) shows the segments of the fingers, and the fingers are fully open, (b) shows the fingers half closed, and (c) shows the fingers fully closed. In (d) the fingers are fully open with object, (e) the fingers half closed with an object, and (f) fully closed and grasping an object.

i	**α(i - 1)**	**a(i - 1)**	**$\mathbf{d_i}$**	**theta**
1	0	0	0.2755	q1
2	$-\pi/2$	0	0	q2
3	0	0.2900	0.0070	q3
4	$-\pi/2$	0	0.1661	q4
5	1.0472	0	0.0856	q5
6	1.0472	0	0.2028	q6

Table 4.1: The Modified DH Parameters for the Mico arm. Where **i** is the joint number, α is the angle between the z-axes measured about the x-axis, a is the distance between the z-axes measured about the x-axis, d_i is the distance between x-axes measured about the z-axis and theta is the angle of the joint.

4.2 V-REP

In this project we use the Virtual Robot Experimentation Platform, V-REP [35], from Coppelia Robotics. V-REP is free to use for educational purposes, supports multiple programming languages, including Python, and is accessible via a remote API. The simulator has models of multiple robotic platforms, including the Mico arm. Whilst the simulation model is very accurate and represents the real arm very well, the control of the fingers is different. Since underactuated fingers are more difficult to implement in simulation, a finger has two actuators that each control a segment. Whilst the real arm uses a linear actuator, the two actuators in simulation are rotational actuators.

Since V-REP isn't made just for grasping but more for general simulations, attempting to grasp objects based on physics is quite difficult because V-REP doesn't model friction that accurately. However in simulation we can detect collision fairly easily, therefore when the fingers are colliding with an object whilst trying to grasp it we can instead of relying on physics, attach the object to the end effector simulating a grasp.

V-REP can operate in different modes; asynchronous and synchronous. The asynchronous mode is most commonly used, it runs the simulation as fast as possible without pausing it. For the synchronous mode however a trigger signal is needed in order to advance the simulation one time step. We need to make use of the synchronous mode because we need the current state of the arm, perform an action and get the new state of the arm whilst keeping the delta time (Δt) the same. This also allows for calculating the reward value, send data to the train buffer, and calculate the new action without the arm still executing its previous action making it end up in a

different state.

The big downside of using the synchronous mode is that when a camera is added to the simulator the update rate drops to about 4Hz-6Hz, versus normally running at around 40Hz-60Hz. This makes it impossible to use visual input for training because it would take to long to train.

For this project we will try to grasp a simple cylinder, with no textures on it since there is no visual input. The cylinder is 12cm in height and has a radius of 3cm, approximately the same size as a Coca-Cola can. V-REP can calculate and visualize the minimum distance between points, in our case we take the minimum distance between the end effector and the middle of the cylinder, which will be used for the reward function. V-REP does collision detection, but we don't want to register all collisions it can possibly make, we are mostly interested in the end effector. We therefore only check if the end effector is colliding with the floor or the object, not with the arm itself. For checking in simulation whether the fingers are grasping the object we perform collision checking on each of the segments whether they collide with the object.

4.3 Tensorflow

In this project we use Tensorflow [2] to program our neural networks. Tensorflow can be programmed in Python which makes it very easy to connect the simulation with the neural networks. Tensorflow is also able to use a GPU to train the neural networks, even though our network sizes are not large enough to fully profit from the parallelism that GPUs provide, it will unload a lot of processing power from the CPU which needs its processing power to run the simulations.

Tensorflow makes it easy to create complex networks, although in our project we only make use of fully connected neural networks. A fully connected neural network can easily be created by matrix multiplications. Tensorflow also has lots of different activation functions available. Tensorflow will automatically calculate the derivatives needed to update weights, this allows for creating deep networks without having to worry about calculating the derivative ourselves. We can also specify the error function and training algorithm. It is also easy to make use of batches for training, and we can easily get the output of a network based on its input. In order to use different parameters we read in a configuration file that allows us to build a correct Tensorflow model based on the specified number of inputs and outputs, number of hidden neurons and layers and learning rates.

4.4 Architectures

Besides using one neural network to control the arm we want to investigate if it is possible to optimize the architecture for controlling the arm. An architecture is the amount of networks controlling a set of joints. Where one neural network would control all 6 joints and fingers, all depending on the same reward function, it could be possible to have two neural networks where each controls a subset of the available actuators and have their own reward function. For example the lower joints have a great effect on where the end effector will end up, whilst the higher joints are used to fine tune the final position of the end effector. Therefore giving these networks different reward functions might improve training performance. To reduce the amount of different hyper parameters that can be fine tuned when using multiple networks, we keep the amount of hidden neurons the same for each network in the architecture. We will however experiment with different reward functions for the networks.

4.5 Exploration

Exploration is an important part of reinforcement learning, without it the agent can not learn. In this project we will try two different exploration methods; Gaussian exploration, and Ornstein-Uhlenbeck process exploration. Besides those two different methods we will also limit the amount of exploration by setting a probability that an actuator will explore. Because we have 7 actuators, 6 joints and 1 end effector, moving in 3d space trying to reach a goal and perform a grasping task, it will become very difficult for just exploration alone to reach the goal. By letting an actuator sometimes perform their learned motion the arm will move closer to its goal and should explore around its goal.

4.6 Agent

An agent will handle the connection with the simulation and neural networks. Before the agent starts with a run it receives the latest Actor network and it resets the simulation world. It will then perform a maximum of 250 steps, where a step is a single action performed in simulation with a Δt of 50ms, resulting in a maximum run time of 12.5 seconds. For each step the agent gets the state, uses the state as input for the Actor network, and uses the output as the actions that it should perform. An action is the velocity in radians that a joint should perform. During training the agent will also add exploration to the action before performing it in simulation. After taking a time step in simulation the new state is gathered and the reward is determined. The begin state, reward, the performed action and the new state are

saved into the replay buffer. The state is the position of all used joints in radians, the x, y, z position of the end effector link in meters with respect to the origin of the arm, and when used, the angle of the fingers in radians, and finally the goal location (x, y, z) in meters. The angle inputs of the joints are normalized to a value between 0 and 1 by dividing the angle with 2π (a full rotation of a joint). The finger joints are normalized by dividing by 1.0472 radians (60 degree) since this is the maximum range they can open. For the position values we don't have to normalize the inputs since the reach of the arm is below 1 meter. This information can be gathered from the simulation. Besides the agent a training thread is running which will take a mini-batch from the replay buffer and will train the Critic and Actor networks, the training thread runs at about 100Hz.

4.7 Pre-Training

Training, or better creating training samples, can be relatively slow using a physics simulator like V-REP. We can however simplify the simulator by only using Forward Kinematics. By keeping track of the current position of each joint we can calculate the end effector position by calculating the forward kinematics. By adding the action from the Actor network, plus exploration, times a delta time of 50ms we can simulate the arm moving. We can run many more trials since one trial takes about 170ms to run. However the downside of this method is that we cannot grasp objects, nor do we have any collision checking. Therefore we will only use this method as a pre-training stage to see if we can reduce the training time when using the physics simulator.

4.8 Experiments

We will perform different experiments to show how well CACLA performs when trying to make a robotic arm learn to grasp objects. The experiments will start with only using a small sub-set of the joints of the robotic arm, until we make use of all joints. We will experiment with making the end effector learn to go to a certain position, and finally we will make the arm attempt to grasp an object.

4.8.1 Network

For the Actor and Critic networks we use the same amount of inputs and same amount of neuron in the hidden layers. The Actor network will always have one output for each joint that is used, when the fingers are used it will use only one output to control both fingers. The Critic network always has one output. The weights for both networks are initialized with a value

between 5×-10^{-2} and 5×10^{-2}. For the Actor the hidden layers will use a rectified linear unit (ReLU) activation function (4.1), the Actor output will be the hyperbolic tangent (tanh) activation function (4.2) to limit the actions within -1 and 1. For the Critic the hidden layers will use the tanh activation function, and the output layer will be linear.

$$relu(x) = max(0, x) \tag{4.1}$$

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{4.2}$$

The networks will be trained with a mini-batch of 64 examples taking from the replay buffer. The replay buffer will hold a maximum of 10^4 examples. The networks are trained with the stochastic gradient descent optimization algorithm.

4.8.2 Position

We first start with letting the arm learn to move the end effector to go to a certain position, this will demonstrate that the algorithm is working correctly and also gives a better insight to parameter tuning. The experiments will start with only using one joint, this means that the end effector will only move on a plane, once we add more joints the end effector will move in 3D space making it more difficult for it to find its goal. We start with using the first joint (Figure 4.2), and add more joints to increase the complexity. In the early stages we will test different learning rates and hidden layers sizes, to reduce the amount of parameters to test we will use early results to determine the best learning rate and hidden layer sizes. We will also test how robust the algorithm is and see if it can handle learning multiple goals. We will experiment with the two different exploration algorithms. When using multiple joints we will also explore different architectures.

The agent has succeeded when it is able to place the end effector within 4cm of the goal, keep it there for at least one time step and the length of the action vector is smaller than 0.001. This would allow the arm to move a bit when it has reached the goal, but making the arm learn to stop at the exact location requires a lot more training time. In this way the arm still has to slow down when it is close to the goal location but doesn't have to spend a lot of time around the goal location in order to come to a complete stop. For the reward function we take the negative distance times a constant, $c1$, and the negative dot product of the actions times a constant, $c2$, see Equation (4.3) where $dist(EEF, Goal)$ is the Euclidean distance between the end effector and the goal position. Early experiments showed that the value 1 for $c1$ and 0.2 for $c2$ give good results, meaning that the factor of distance is more important over the action output. With this reward

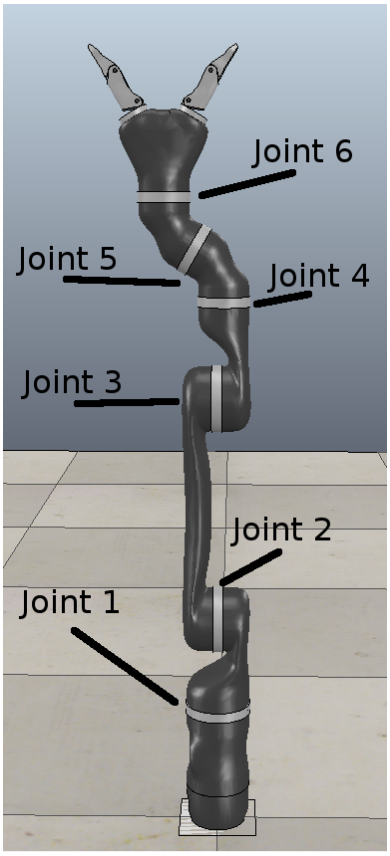


Figure 4.2: The Mico arm in simulation with joint number indication.

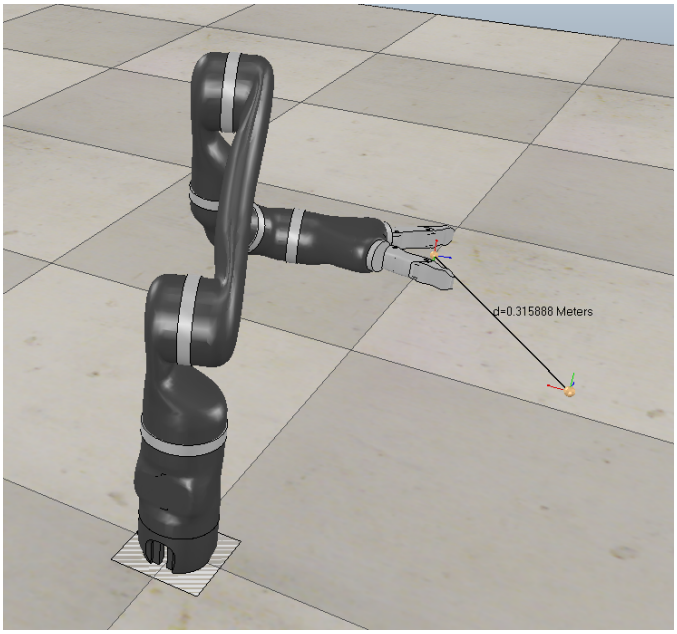


Figure 4.3: The simulation experimental setup for moving to a position. The starting position of the arm with a distance measurement to its goal position.

function the highest reward can be achieved by moving to the goal position and have a zero output value for each action, moving away from the goal location would give a lower reward. When the arm collides with the floor the run fails and is ended, an extra negative reward of -1 is added to the already existing reward that was calculated when taking its final action. If the arm is colliding with itself the run keeps continuing. The reason for not failing on self collision is because it requires extra calculations slowing the simulation down, but also because during early experiments it was shown that it never causes a problem because it barely happens and it will only receive larger negative reward since it does not progress towards the goal. In Figure 4.3 the experimental setup is shown. The line between the two indicators shows the distance between the end effector and its goal position.

$$r = -dist(EEF, Goal) \times c1 - c2 \times \sum_{i} a_i a_i \tag{4.3}$$

4.8.3 Grasping

For grasping we will only attempt to grasp a single object, a cylinder representing a Coca-Cola can. With a grasp we mean that the fingers are grasping the object, but it doesn't have to lift the object. Grasping and

lifting an object using the physics of the simulation can be a very difficult thing to do correctly and would require a lot of fine-tuning of settings in the simulator. Also not receiving any feedback from the end effector makes it very difficult to determine good grasps. We use all the joints in these experiments and the fingers are always added in these experiments. The fingers are controlled with one action, even though the fingers could be controlled separately we only control both of them at the same time. This will also create a better gripping motion. We will test the two different exploration methods, and also different architectures. We also test the same setup with a simple IK planner. We will use MoveIt for these experiments. The downside of comparing with this method is that it requires an extra step to tell the IK planner how to grasp the objects. We will use a very simple method of determining how to grasp the object, we will give it multiple angles for approaching the middle point position of the object, the planner will determine the best from that collection.

The agent has succeeded when all the finger segments are touching the object. This would mean that fingers are grasping the object, however it could also be that the grasp is not very accurate. The end effector could grasp the object with an angle causing the fingers to not fully grasp the object but only with a small surface. In real life the object would then move and most likely fit better into the grasping fingers, however in simulation this can cause the object to create weird collision behavior. It was therefore chosen that the object is heavy and could not be moved by the arm, in this way we don't have to worry about the object acting weirdly but do take into consideration that a grasp is not always an optimal grasp. For the reward function we take the negative distance times a constant, $c1$, when more than 5cm away from the object a negative angle of the fingers times a constant, $c2$, when within 5cm of the object a positive angle of the fingers times $c2$, and the negative dot product of the actions times a constant, $c3$, see Equation (4.4). Early experiments show that the value 1 for $c1$, 0.5 for $c2$, and 0.2 for $c3$ give good results. Similarly to the previous experiments the run stops when the end effector touches the ground, with -1 added to the reward, but not on self collision. Figure 4.4 shows the experimental setup for grasping the object. The distance measurement is the distance between the end effector point and the center of the object.

$$r = -dist(EEF, Goal) \times c1 + sgn(dist(EEF, Goal)) \times c2 \times \theta_{fingers} - c3 \times \sum_i a_i a_i \quad ; \quad sgn(dist(EEF, Goal)) = \begin{cases} -1 & if \quad dist > 5cm \\ +1 & if \quad dist <= 5cm \end{cases} \tag{4.4}$$

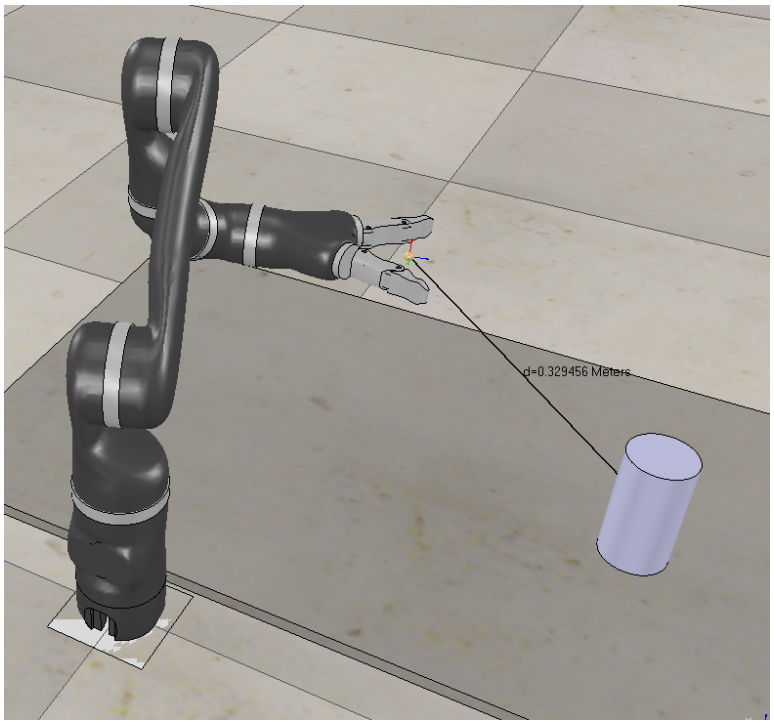


Figure 4.4: The simulation experimental setup for grasping an object. The starting position of the arm with a distance measurement to its goal position, the center position of the object. The object is a cylinder approximately the size of a Coca-Cola can.

Chapter 5

Results

In this chapter we present the results for the experiments that were performed. In the first part we present the results of the positioning experiments, in the second part we present the results for the grasping experiments.

5.1 Position Experiments

During each experiment we performed a test run after each 10 trials, a test run was without adding exploration and without training the networks at each step. The Actor and Critic networks have 2 hidden layers of each 150 neurons, and they were trained using the online method. The network configuration did not seem to have a large impact on the results in early experiments and we therefore chose this configuration to also keep it robust for later more complex experiments. The starting position was kept the same for all experiments, the experiment with one joint had a different goal position from when using multiple joints because it can only reach a position on a certain radius, when using multiple joints the position was always kept the same. Each experiment ran for a total of 5,000 trials and was conducted 10 times. We also experimented with the two different exploration methods. For the Gaussian exploration we started with a sigma of 30 degrees, for the OUP exploration we used a K value of 0.15. We used a learning rate of 0.01 for both the Actor and Critic as these seemed to give good results in early experiments. Table 5.1 shows the success rate and the mean distance to the goal location of 10 trials for different number of used joints with the Gaussian exploration method. Table 5.2 shows the result when using the OUP exploration method.

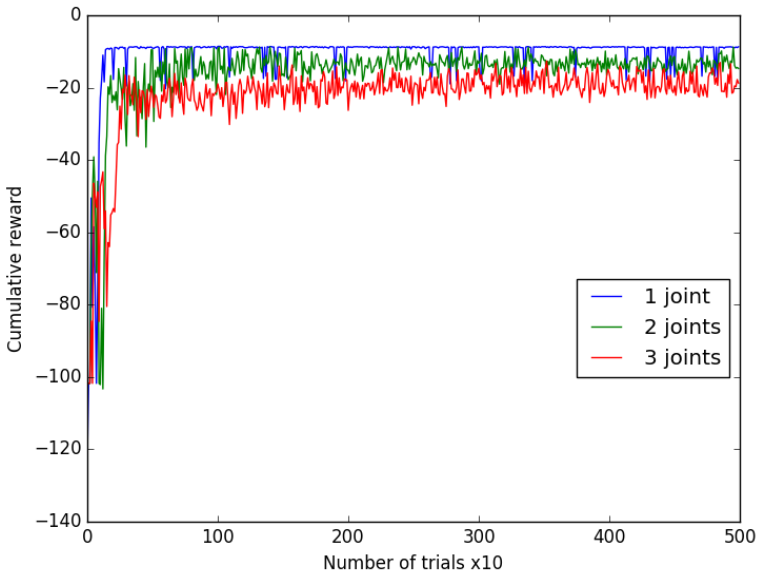
	1 Joint	2 Joints	3 Joints	4 Joints	5 Joints	6 Joints
Success rate	100%	50%	40%	20%	0%	10%
Mean distance	0.84cm	4.42cm	6.73cm	14.08cm	21.39cm	13.5cm
Standard deviation	0.21cm	2.10cm	4.21cm	11.19cm	13.62cm	11.89cm

Table 5.1: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, with the Gaussian exploration method.

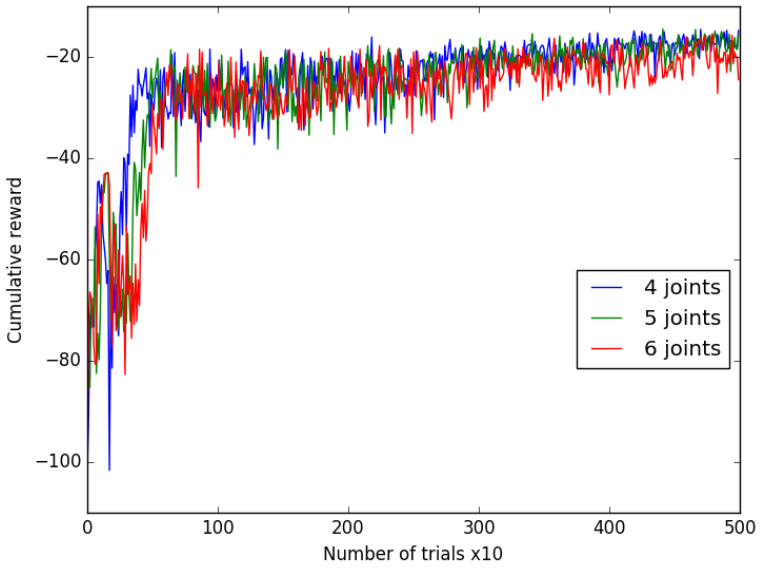
	1 Joint	2 Joints	3 Joints	4 Joints	5 Joints	6 Joints
Success rate	100%	80%	30%	0%	10%	20%
Mean distance	0.77cm	3.67cm	9.91cm	15.78cm	12.10cm	15.51cm
Standard deviation	0.21cm	1.60cm	8.68cm	11.73cm	6.56cm	13.29cm

Table 5.2: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, with the OUP exploration method.

The results show that with 1 joint the problem can easily be solved with a 100% success rate and the final position is very close to its goal position for both exploration methods, the problem is also much easier to solve since the end effector can only move with a fixed radius on a plane level. When adding more joints the end effector can move in 3 dimensions making it a much more difficult problem to solve. When adding the second joint the OUP exploration seems to perform slightly better, 80%, having a 30% higher success rate than Gaussian, 50%. The mean distance of both explorations methods are very close to the distance threshold for counting a success, 4.42cm for Gaussian and 3.67cm for OUP. Adding the third and fourth joint gives the Gaussian exploration a bit higher success rate, 40% and 20%, but only 10% and 20% higher compared to OUP, 30% and 0%. The mean distances are fairly close to each other. When adding the fifth and sixth joint however the OUP performs slightly better, 10% and 20%, both 10% higher success rate than Gaussian, 0% and 10%. For the five joints the Gaussian mean distance, 21.39cm, is almost double as that of the OUP exploration, 12.10cm, but for the 6 joints it is a bit smaller, 13.5cm for Gaussian and 15.51cm for OUP, even though it was not more successful. In Figure 5.1 the cumulative reward for the Gaussian experiments are shown for all the different joints used, it shows the test results during training. With one joint, Figure 5.1a the reward goes up very quickly and stays very constant. When adding more joints there are more spikes downwards, specially during the first 1,000 trials. This mostly happens because the Actor starts with very low velocity outputs and slowly learns to move towards the goal. But whilst it has sometimes learned to go to the goal it has not yet learned to stop, therefore overshooting the goal and creating extra negative rewards.



(a) The cumulative rewards for using 1, 2 and 3 joints.



(b) The cumulative rewards for using 4, 5 and 6 joints.

Figure 5.1: The cumulative rewards for a single run for the online 5,000 trials experiments with Gaussian exploration. A test run was done every 10 trials. In (a) the result for using 1, 2 and 3 joints, in (b) the results for using 4, 5 and 6 joints.

Online learning does not seem to work very well when using a higher amount of joints. Instead of online learning we can make use of batch learning with the use of a large buffer and using two agents to create examples. By using two agents to create examples we create more diverse samples in the buffer, but also generate examples faster since the agents work in a separate thread independently from each other.

	5,000 trials, Gaussian	10,000 trials, OUP	10,000 trials, Gaussian
Success rate	60%	80%	90%
Mean distance	9.20cm	5.49cm	2.63cm
Standard deviation	10.20cm	6.31cm	1.00cm

Table 5.3: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, by making use of a buffer for batch learning and two agents. All six joints are used.

In Table 5.3 we can see the results from batch learning using two agents. Using the same amount of trials for the agents, 5,000, the Actor and Critic networks have been updated with a batch of 64 samples approximately five million times. We can see that this gives better results than online learning, however 60% is not very efficient for a relatively easy task. When we let the agents learn for 10,000 trials the results become much better, both increasing in success rate, 80% for OUP and 90% for Gaussian, and in a lower mean distance, 5.49cm for OUP and 2.63cm for Gaussian. The OUP exploration seems to perform well, however the trials that failed had a high distance from their goal. The Gaussian exploration seems a lot better with only one trial failing, and having a very low mean distance and standard deviation. The big downside of letting the agents run for 10,000 trials is that it takes over 10 hours to train. If we add a pre-training stage of 6,000 trials, and a normal training stage of 3,000 trials the results are much better. The results are shown in Table 5.4. The Gaussian exploration has a 90% success rate, where the OUP exploration has a 100% success rate and a very low standard deviation, 0.35cm. The total number of trials is lower, whilst also having a better accuracy in both cases compared with the 10,000 trials without pre-training. Another aspect is that the total training time is reduced to approximately 4 hours.

In order to show the robustness of the algorithm we let it train to go to a point that is randomly chosen within a 20×20cm workspace, keeping the height always the same, during each trial. When testing we pick 100 random point within this area and see how many times it can find that point. Table 5.5 shows the results for each trial. For all trials except trial 8 the neural network is able to position the end effector in the correct position 100 times when choosing a random goal whilst also having a very low mean distance

	Gaussian	OUP
Success rate	90%	100%
Mean distance	3.01cm	2.25cm
Standard deviation	2.19cm	0.35cm

Table 5.4: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, using pre-training of 6,000 trials, and training using the physics simulator for 3,000 trials, using a buffer for batch learning and two agents. All six joints are used.

Trial	Success rate	Mean distance	Standard deviation
1	100%	1.80cm	0.41cm
2	100%	2.51cm	0.59cm
3	100%	2.5cm	0.47cm
4	100%	1.8cm	0.46cm
5	100%	1.47cm	0.78cm
6	100%	2.79cm	1.09cm
7	100%	1.63cm	0.28cm
8	85%	1.40cm	0.45cm
9	100%	1.69cm	0.32cm
10	100%	2.39cm	0.33cm

Table 5.5: The average success rate, mean distance with standard deviation of the final position relative to the goal position. The goal position was in a 20×20cm workspace. Per trial, 100 random points were chosen. Pre-training was used for 6,000 trials, training in the physics simulator was done for 3,000 trials, using a buffer for batch learning and two agents. All six joints are used.

and standard deviation. Only trial 8 has a lower success rate, 85%, where its mean distance and standard deviation are slightly larger than the other trials. This shows that the algorithm is able to learn multiple goals.

We also tested one architecture with two networks each controlling a set of joints, the first network controlled joints 1, 2 and 3, the second network controlled joints 4, 5 and 6. We only tested Gaussian exploration. Table 5.6 shows the results. It can be seen that in 70% of the time the multi neural network architecture is able to create a success. Whilst this is still lower than using the one neural network, it has shown that it is able to learn using multiple neural networks. Whilst the mean distance is relatively close to the results from one neural network, the standard deviation is quite large, meaning that the networks that have failed didn't even come relatively close

	2 Neural Network Architecture
Success rate	70%
Mean distance	3.64cm
Standard deviation	3.40cm

Table 5.6: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, using pre-training of 6,000 trials, and training using the physics simulator for 3,000 trials, using a buffer for batch learning and two agents. Two networks were used, one to control joints 1, 2 and 3, the other network to control joints 4, 5 and 6.

to the goal. This could mean that one of the two neural networks never learned to go to the goal properly or that both networks where not capable of learning correctly.

5.2 Grasping Experiments

During each experiment we performed a test run after each 10 trials, a test run was without adding exploration and without training the networks at each step. The Actor and Critic networks had 2 hidden layers of each 150 neurons, and were trained using the buffer method. We always used the same objects to attempt to grasp, and the object was always in the same position. Table 5.7 shows the results of using 8,000 trials of pre-training and 4,000 trials of learning in the physics simulator. As the results show the network has not learned to grasp the object. Whilst the mean distance does show it was coming close to the object it was not able to actually grasp it. Whilst during the test run it came very close to grasping in some trials, it mostly had a wrong orientation trying to grasp the object vertically instead of horizontally. This is most likely the case because it was over trained during the pre-training stage. Since this stage has no physics, and therefore can not even attempt to grasp, it has only learned to go to a position without knowing there is an object in its way that it should grasp.

Since grasping is a lot harder than just going to a point, we increase the total of runs to 10,000 and do not train with the pre-training stage. Table 5.8 shows the results. With a 70% success rate it is shown that the algorithm is able to learn to grasp an object. The downside is that it takes a very long time to train, over 10 hours, without using the pre-training method. The mean distance of 3.94cm shows that the networks were able to get the end effector close to its final position, but sometimes the orientation was not correct. Sometimes the end effector and fingers were around the object but the trial was not successful. This most likely means that not all segments

	Gaussian
Success rate	0%
Mean distance	6.11cm
Standard deviation	4.12cm

Table 5.7: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, using pre-training of 8,000 trials, and training using the physics simulator for 4,000 trials, using a buffer for batch learning and two agents. All 6 joints and two fingers were used.

	OUP
Success rate	70%
Mean distance	3.94cm
Standard deviation	1.93cm

Table 5.8: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, training using the physics simulator for 10,000 trials, using a buffer for batch learning and two agents. All 6 joints and two fingers were used.

of the fingers were correctly colliding with the object.

When using 2 neural networks, one to control joints 1, 2, and 3, and the other to control joints 4, 5, 6, and the fingers the results become a bit better. For the first neural network only the distance and action were taken into account for the reward function, for the second neural network the fingers were also taken into account. Table 5.9 shows the results of using the two neural network architecture. Whilst the success rate hasn't improved, 70%, the mean distance and standard deviation, 1.36cm and 0.37cm, have decreased quite a lot. Again not every failure was a complete failure. This shows that using multiple neural networks to control the robot arm can improve results a bit.

When using 3 neural networks the results can be improved even more. The first network controls joint 1 and 2, the second network joint 3 and 4, and the third network controls joint 5, 6 and the fingers. The first two networks only used the distance and action as rewards, the third network also takes into account the fingers. Table 5.10 shows the result from the 3 neural network architectures for each of the exploration methods. Both exploration methods perform almost similar, both achieving a success rate of 100% and a mean distance of 2.87cm and 2.51cm, with both a standard deviation of 0.9cm. This shows that using multiple networks to control a

	OUP
Success rate	70%
Mean distance	1.36cm
Standard deviation	0.37cm

Table 5.9: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, training using the physics simulator for 10,000 trials, using a buffer for batch learning and two agents. Two networks were used, one to control joints 1, 2 and 3, the other network to control joints 4, 5, 6 and the fingers.

	OUP	Gaussian
Success rate	100%	100%
Mean distance	2.87cm	2.51cm
Standard deviation	0.9cm	0.9cm

Table 5.10: The average success rate and mean distance with standard deviation of the final position relative to the goal position. Average of 10 trials, training using the physics simulator for 10,000 trials, using batch learning and a buffer. Three networks were used, one to control joints 1 and 2, the second network to control joints 3 and 4, the third network to control joint 5, 6 and the fingers.

robotic arm can actually perform much better than using only one neural network. The downside of using multiple networks is that the training time is even longer since it takes more time to iterate through all the networks. When using 3 neural networks (6 when including the Critic networks) the training time can approximately be 1.5 times longer than using only one neural network.

5.3 MoveIt

A very short experiment was performed using the MoveIt framework in order to compare the results from a neural network controller to a simple IK planner. The object was the same as the grasping experiments, and was positioned at the same position, the starting position of the arm was also the same. The planning algorithm used was the RTT-Connect algorithm using standard parameters. The grasping location was the center of the cylinder, the same position as used in the grasping experiments, and a suggested grasp angle was given to the planner for approaching the object from straight forward, +/- 90 degrees from the object with steps of 1 degree whilst keeping the end effector level with ground, and also angling the end effector downwards with steps of 10 degree up to 50 degrees, giving it 9000 possible

approach angles to perform the grasp. MoveIt will select the grasp approach from which it finds a valid solution first. Even when performing the runs multiple times the average planning time took about 0.5 seconds to find a valid solution, and it was 100% accurate with 10 trials. The experiments were however performed with the real robotic arm because of some stability problems with the simulation. The results show that the simple IK planner can easily solve a simple grasping problem.

Chapter 6

Conclusion

In this chapter we will conclude our research, answer the research questions and discuss future work that can be done to improve the results from this project.

6.1 Position

In the go to a position set of experiments we have shown that CACLA is able to learn and control a robotic arm. Whilst online learning is possible using one to three joints, the performance decreases when using more than three joints. Whilst one solution would be to let it run for longer than 5,000 trials, it would also take a lot more time to learn this relatively simple task. By making use of a buffer from which a training thread can train the networks with mini-batches, we have shown that when using the same amount of trials the results are much better, but not perfect yet. When doubling the amount of trials the success rate becomes acceptable with 80% and 90% success rates for each of the exploration methods, however the training time is also doubled. We introduced a simplified version of the representation of the arm using Forward Kinematics to increase the amount of iterations an agent can make to create training examples. When using a relatively large amount of pre-training trials and a smaller amount of training using the physics simulator we have increased the performance whilst also decreasing the total time it takes to train the networks. With a 90% and 100% success rate over 10 runs, the algorithm seems very robust in order to let the arm learn to move the end effector to a certain position. It is even more robust by letting it learn to go to a random goal that lies within a small workspace. With a 100% success rate in almost all the trials, it is shown that the network can learn multiple goals. The difference between the two exploration methods does not seem to effect the final results very much. In some cases the Gaussian exploration method seemed to perform better while in other cases the OUP exploration method seemed to perform better, however the

difference in success rate, mean distance and standard deviation was relatively small. When using two neural networks to control the arm the results do not improve, although we showed that using multiple neural networks still works with a 70% success rate.

6.2 Grasping

In the grasping experiments we have shown that CACLA is able to learn to let a robotic arm grasp an object. Whilst pre-learning worked really well when learning to go to a position only, the grasping part requires the actual physics simulator to learn, giving a 0% success rate when using the pre-training phase. When we use only the physics simulator and increase the total amount of runs the agent does to 10,000, the results show that it is able to learn to grasp an object with a 70% success rate. The large downside is that training takes quite a lot of time because we cannot speed it up with pre-training. When using multiple networks to control the arm the results slightly increase. Whilst only testing with the OUP exploration method, the success rate stays at 70% compared to the single network, but the mean distance is decreased from 3.49cm to 1.36cm, meaning it has gotten closer to its target. But when using 3 neural networks the results actually become a lot better, for both exploration methods the success rate is 100%, and a mean distance that is very close to each other, 2.87cm and 2.51cm. The main success can most likely be explained by the difference in reward function. Only the last network takes into account the finger position and therefore whether it is going to grasp or not, the networks with the lower joints are mainly for bringing the end effector close to its target. By only using distance and not the finger position it is most likely to learn a lot faster to put the end effector close to its goal position since it can learn separately from the grasping part.

6.3 Answers to Research Questions

How does reinforcement learning compare to a much simpler inverse kinematic controller for grasping objects? Is there a gain in speed, accuracy or robustness?

Comparing a reinforcement learning algorithm with an inverse kinematic controller is quite difficult. Whilst in short we can say; no there is no gain in speed, it is not more accurate, and it is not (always) more robust, it is not completely fair. Kinematic controllers are being used and developed for a much longer time and are specifically aimed at robotics, whilst reinforcement learning can be applied to much more problems than just robotic grasping and therefore not as much used to control a robotic arm for ex-

ample. Because of the simple setup the inverse kinematic planner has no problem solving it, even when given 9000 possible approach angles, and has no problem grasping the object and is always able to complete its task. But if we extrapolate the problem and assume that a reinforcement learning algorithm can learn to solve much more complex problems, then the answer for speed can be answered in two different ways. One way of speed is taking into account the amount of time it takes to calculate a solution and perform the trajectory versus the amount of time it takes to train a neural network and it takes to execute its trajectory. If the planner would take more than 0.5 seconds, let's say 10 seconds to solve a complex problem then compared to how much time it takes to train a neural network it is still faster. However if the complex problem needs to be solved many times, e.g. in an industrial setting, the amount of training time, that could perhaps take 20 hours, might be negated since a trained neural network can execute its trajectory immediately. So if a trained neural network is used more than the total time it takes for the kinematic controller to plan its trajectory, the neural network controller is faster. For the other speed measurement, the actual speed of the execution, it is a bit more complex. A trained neural network will always output the same velocity if it needs to repeat a run with the same goal position. Whilst with a kinematic planner the velocity can be scaled to the operators needs, this is however also possible with the neural network controller by simply multiplying the output with a scaling factor. However the total trajectory execution time cannot be determined beforehand with a neural network controller and therefore there is no gain in speed compared to a kinematic controller.

Accuracy can be measured as how precise the grasping is compared to where it was supposed to grasp. In this case the kinematic controller is much more accurate than the neural network controller. Since the kinematic controller follows a specific path and plans to a specific goal location given by the operator it is not going to deviate from it. The neural network controller has to learn itself and it is therefore very difficult to have millimeter precision. We define a success when it is within a certain threshold, measured in centimeters, otherwise it would never be able to succeed. The inverse kinematic planner's success can be measured in millimeters, although its result depends more on the controller than on the planner.

The neural network controller has shown that it can be very robust, in most cases at least 1 trial was able to complete its task. The 3 neural network architecture controller shows that it is very robust and can learn with a 100% success rate to complete its task. The kinematic planner however is also very robust since it is also always able to complete its task. In more complex scenes the kinematic planner has the advantage of failing before executing, if the planner can not find a valid plan it will not execute movement of the robotic arm. The neural network controller will always output something because it cannot determine beforehand if it can solve the prob-

lem or not, as seen in the failed trials where it did execute its trajectory but was not able to complete it. So for robustness it can in some cases compete with a kinematic controller, except when it is going to fail since a kinematic controller would not execute where a neural network controller would try to solve the problem.

Which neural network architecture results in the best performance?

Whilst we were not able to test all different architectures because of time limitations, we can conclude that the 3 neural network architecture performs best, even better than a single neural network when trying to grasp an object. However we do have to take into account it was only tested within a very simple experimental setup, it would require more testing to see whether a 3 neural network architecture can outperform a one neural network. We were also not able to test an architecture where each joint got its own neural network.

How does the exploration method influence the training?

When looking at our final experiment, it doesn't seem to be of any influence at all. The amount of success rate is the same, 100%, and the mean distance has only a difference of 0.37cm, showing that the exploration doesn't really change the results. But if we look at earlier results there is still no clear indication which exploration method performs better. In some cases the OUP exploration was slightly better, whilst in other cases the Gaussian exploration was better. Therefore the exploration methods used in this project do not influence the training.

6.4 Future Work

There are a lot of things that can and need to be done to improve the results. First we could add a camera to the setup so that the network can learn from perception. Whilst this was the initial idea to use, we hit a limit in the V-REP simulation with respect to update rates. Adding perception could improve the learning because it provides a more informative state. Although adding a camera in our experimental setup might improve the results for our setup, when we increase the difficulty of the experiment by adding multiple objects in the scene, a higher level decision making process needs to be added that tells the network which of the objects needs to be grasped.

Another improvement could be made by adding regularization techniques

like batch normalization and dropout. Whilst we shortly experimented with them, although not presented in this thesis, we did not see any significant improvement. However we only briefly experimented with the two techniques and only used the standard values that were provided by the Tensorflow framework. Improvements could happen with different parameters, but this would require more experiments for which we did not have the time.

Improvements could also be made by making use of a different robotic arm. The only reason the Mico arm was chosen is because it was the only robotic arm available in our robotics lab at the time and one of the important part in robotics is that it actually works in the real world. Therefore having a real robotic arm of the same type used in simulation would be more useful then picking a robotic arm that is not accessible to us in the real world. However during the final phase of this project a new robotic arm became available in our robotics lab, the Panda robotic arm from Franka Emika[1]. The Panda is a 7 Degree of Freedom (DoF) robotic arm with a two finger gripper that is moved linearly. The main advantage of this robotic arm is that is has force sensing, meaning it can sense with how much force it is pushing against an object if it is colliding with something, and also has force sensing when grasping an object. This feedback can provide valuable information for the reward function. The other interesting part is the 7th DoF, this 7th DoF allows for the end effector to stay in the same position whilst the rest of the arm can still move. This added complexity should make it easier for the end effector to reach its final position, but it would also increase the complexity during exploration and possibly making it harder for the Actor and Critic to learn.

This research has only focused on a relatively simple scenario in the field of manipulation. Whilst this project originated from the research done with a domestic service robot, the experimental setup does not reflect real world use such that it can be used on a domestic service robot platform. In order for it to be used on such a platform it needs to have learned a lot more complex tasks; it needs to know how to handle multiple objects, learn how to grasp all sorts of different objects from all possible positions that the end effector can actually grasp the object. It will also have to learn the difference between grasping a solid object that can be moved with high speed versus the grasping of a glass full of water without spilling anything. Besides grasping it also has to learn to place objects down, poor the content of an object into an other object, hand the object to an operator, not crush the object, not cause collisions and in general safe operating around and with humans. Besides grasping objects it should also be able to open doors, closets, and tasks like cleaning a table with a wiper, or operate a machine

[1] http://franka.de/

to, for example, make coffee.

Bibliography

[1] Tesla autopilot. *https://www.tesla.com/autopilot*.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Ethem Alpaydin. *Introduction to machine learning*, volume 2. MIT press, 2010.

[4] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[5] Joost Broekens, Marcel Heerink, Henk Rosendal, et al. Assistive social robots in elderly care: a review. *Gerontechnology*, 8(2):94–103, 2009.

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] John J Craig. *Introduction to robotics: mechanics and control*, volume 3. Pearson Prentice Hall Upper Saddle River, 2005.

[8] J Denavit and R.S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, pages 215–221, 1955.

[9] Murat Dikmen and Catherine M Burns. Autonomous driving in the real world: Experiences with tesla autopilot and summon. In *Proceedings of the 8th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pages 225–228. ACM, 2016.

[10] Hobart R Everett and Douglas W Gage. From laboratory to warehouse: Security robots meet the real world. *The International Journal of Robotics Research*, 18(7):760–768, 1999.

[11] Dario Floreano and Robert J Wood. Science, technology and the future of small autonomous drones. *Nature*, 521(7553):460, 2015.

[12] Willow Garage. Xml robot description format (urdf). *h ttp://www. ros. org/wiki/urdf/XML, accessed March*, 10, 2012.

[13] James Gibson. J.(1977). the theory of affordances. *Perceiving, acting, and knowing: Toward an ecological psychology*, pages 67–82.

[14] James J Gibson. *The ecological approach to visual perception: classic edition*. Psychology Press, 2014.

[15] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[16] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[17] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 3389–3396. IEEE, 2017.

[18] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep Q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[19] IFR. Why service robots are booming worldwide - IFR forecasts sales up 12%. *https://ifr.org/downloads/press/2017-10-11_PR_IFR_World_Robotics_Report_2017_Service_Robots_ENG_FINAL_1.pdf*.

[20] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 1398–1403. IEEE, 2002.

[21] Kinova. Kinova mico arm. *http://www.kinovarobotics.com/innovation-robotics/products/robot-arms/*.

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

[23] James J Kuffner and Steven M LaValle. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[24] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[25] Gabriel Leuenberger and Marco A Wiering. Actor-Critic reinforcement learning with neural networks in continuous games. *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART)*, 2018.

[26] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 163–168. IEEE, 2011.

[27] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

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

[29] Fujio Miyawaki, Ken Masamune, Satoshi Suzuki, Kitaro Yoshimitsu, and Juri Vain. Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *IEEE Transactions on Industrial Electronics*, 52(5):1227–1235, 2005.

[30] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[31] Donald A Norman. Affordance, conventions, and design. *interactions*, 6(3):38–43, 1999.

[32] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A general purpose library for collision and proximity queries. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3859–3866. IEEE, 2012.

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

[34] Deepak Rao, Quoc V Le, Thanathorn Phoka, Morgan Quigley, Attawith Sudsang, and Andrew Y Ng. Grasping novel objects with depth segmentation. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2578–2585. IEEE, 2010.

[35] E. Rohmer, S. P. N. Singh, and M. Freese. V-REP: a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[36] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[37] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2):157–173, 2008.

[38] Jochen J Steil, Frank Röthling, Robert Haschke, and Helge Ritter. Situated robot learning for multi-modal instruction and imitation of grasping. *Robotics and autonomous systems*, 47(2):129–141, 2004.

[39] Freek Stulp, Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. Learning to grasp under uncertainty. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 5703–5708, 2011.

[40] Ioan A Sucan and Sachin Chitta. Moveit! *Online at http://moveit. ros. org*, 2013.

[41] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. `http://ompl.kavrakilab.org`.

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

[43] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *Journal of Machine Learning Research*, 11(Nov):3137–3181, 2010.

[44] Emre Ugur, Erol Şahin, and Erhan Oztop. Unsupervised learning of object affordances for planning in a mobile manipulation platform. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 4312–4317, 2011.

[45] Hado Van Hasselt and Marco A Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007. IEEE International Symposium on*, pages 272–279, 2007.

[46] Richard M Voyles and Pradeep K Khosla. A multi-agent system for programming robots by human demonstration. *Integrated Computer-Aided Engineering*, 8(1):59–67, 2001.

[47] Thomas Wisspeintner, Tijn Van Der Zant, Luca Iocchi, and Stefan Schiffer. Robocup@ home: Scientific competition and benchmarking for domestic service robots. *Interaction Studies*, 10(3):392–426, 2009.