DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Reinforcement learning for robotic manipulation

ISAC ARNEKVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Reinforcement learning for robotic manipulation

ISAC ARNEKVIST

Master in Machine Learning
Date: April 24, 2017
Supervisor: Johannes A. Stork
Examiner: Danica Kragić Jensfelt
Swedish title: Reinforcement learning för manipulering med robot
School of Computer Science and Communication

Abstract

Reinforcement learning was recently successfully used for real-world robotic manipulation tasks, without the need for human demonstration, using a *normalized advantage function*-algorithm (NAF). Limitations on the shape of the advantage function however poses doubts to what kind of policies can be learned using this method. For similar tasks, convolutional neural networks have been used for pose estimation from images taken with fixed-position cameras. For some applications however, this might not be a valid assumption. It was also shown that the quality of policies for robotic tasks severely deteriorates from small camera offsets. This thesis investigates the use of NAF for a pushing task with clear multi-modal properties. The results are compared with using a deterministic policy with minimal constraints on the Q-function surface. Methods for pose estimation using convolutional neural networks are further investigated, especially with regards to randomly placed cameras with unknown offsets. By defining the coordinate frame of objects with respect to some visible feature, it is hypothesized that relative pose estimation can be accomplished even when the camera is not fixed and the offset is unknown. NAF is successfully implemented to solve a simple reaching task on a real robotic system where data collection is distributed over several robots, and learning is done on a separate server. Using NAF to learn a pushing task fails to converge to a good policy, both on the real robots and in simulation. Deep deterministic policy gradient (DDPG) is instead used in simulation and successfully learns to solve the task. The learned policy is then applied on the real robots and accomplishes to solve the task in the real setting as well. Pose estimation from fixed position camera images is learned and the policy is still able to solve the task using these estimates. By defining a coordinate frame from an object visible to the camera, in this case the robot arm, a neural network learns to regress the pushable objects pose in this frame without the assumption of a fixed camera. However, the precision of the predictions were too inaccurate to be used for solving the pushing task. Further modifications to this approach could however show to be a feasible solution to randomly placed cameras with unknown poses.

Sammanfattning

Reinforcement learning har nyligen använts framgångsrikt för att lära icke-simulerade robotar uppgifter med hjälp av en *normalized advantage function*-algoritm (NAF), detta utan att använda mänskliga demonstrationer. Restriktioner på funktionsytorna som använts kan dock visa sig vara problematiska för generalisering till andra uppgifter. För pose-estimering har i liknande sammanhang convolutional neural networks använts med bilder från kamera med konstant position. I vissa applikationer kan dock inte kameran garanteras hålla en konstant position och studier har visat att kvaliteten på policys kraftigt förvärras när kameran förflyttas. Denna uppsats undersöker användandet av NAF för att lära in en ”pushing”-uppgift med tydliga multimodala egenskaper. Resultaten jämförs med användandet av en deterministisk policy med minimala restriktioner på Q-funktionsytan. Vidare undersöks användandet av convolutional neural networks för pose-estimering, särskilt med hänsyn till slumpmässigt placerade kameror med okänd placering. Genom att definiera koordinatramen för objekt i förhållande till ett synligt referensobjekt så tros relativ pose-estimering kunna utföras även när kameran är rörlig och förflyttningen är okänd. NAF appliceras i denna uppsats framgångsrikt på enklare problem där datainsamling är distribuerad över flera robotar och inlärning sker på en central server. Vid applicering på ”pushing”-uppgiften misslyckas dock NAF, både vid träning på riktiga robotar och i simulering. Deep deterministic policy gradient (DDPG) appliceras istället på problemet och lär sig framgångsrikt att lösa problemet i simulering. Den inlärda policyn appliceras sedan framgångsrikt på riktiga robotar. Pose-estimering genom att använda en fast kamera implementeras också framgångsrikt. Genom att definiera ett koordinatsystem från ett föremål i bilden med känd position, i detta fall robotarmen, kan andra föremåls positioner beskrivas i denna koordinatram med hjälp av neurala nätverk. Dock så visar sig precisionen vara för låg för att appliceras på robotar. Resultaten visar ändå att denna metod, med ytterligare utökningar och modifikationer, skulle kunna lösa problemet.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis aims to investigate how to enable a robot to learn autonomous interaction with an object so that some goal is accomplished. Given complete knowledge of the environment we can often handcraft solutions that work well. However, as environments vary from one task to another and might even be intractable to model explicitly, especially in the presence of stochastic elements, a more general learning approach is motivated. Knowledge we have about a certain task enables us to argue about the shortcomings and successes of a learning agent, and a task for which we have plenty of insights, such as pushing an object to a target position, is therefore motivated. A pushing task having a multi-modal nature, in conjunction with real-world noisy controls and high-dimensional noisy sensor input still makes this a challenging task even though the task in a higher-level sense is relatively simple.

1.2 Machine Learning

Machine Learning is the scientific branch dealing with algorithms that can learn tasks like classification, regression, finding latent structure, performing tasks etc. In a growing number of domains, algorithms have recently been superceding, or nearing human level performance, e.g. image classification [1, 2], playing board [3] and video games [4], and speech recognition [5, 6]. This has been made possible due to several reasons such as efficient algorithms, better hardware, a surge in the amount of data, etc. Despite recent advancements, many tasks that seem trivial

to humans are continually hard for computers to learn e.g. household chores like doing dishes, washing, and cooking.

1.3 Robotic manipulation

Reinforcement Learning (RL) is the branch of Machine Learning that deals with learning what actions to do in order to reach a long-term goal and have been widely used for learning robotic manipulation tasks. Models capable of learning these tasks often need large amounts of possibly unsafe interactions with the environment in order to be learned. These tasks are therefore commonly trained in simulation rather than on real robotic systems. Recent research suggests methods capable of learning real-world robotic manipulation tasks without simulation pre-training, learning only from real-world experience [7–10]. Using visual feedback for manipulation tasks is a way to handle unknown poses of manipulators and target objects, and training these kind of tasks end-to-end have been successful in simulation tasks [11, 12]. Pose estimation for real-world robotic manipulation have been shown to work by using convolutional neural networks (CNN) [7, 10, 13], although for some cases it was shown that test-time translations severely affected manipulation task performance [7]. CNNs have also been trained to deal with relative poses [14], and this could be a possible solution in order to deal with unknown and random camera offsets. Real-world experiments often rely on human demonstrations to learn a successful policy but this might not always be available. Recently a successful demonstration of learning a door opening task from scratch without the need for human demonstration or simulation pre-training have been shown [8]. This was using a version of Normalized Advantage Function algorithms (NAF) [15] distributed over several robotic platforms. While NAF on a door-opening task was shown to outperform Deep Deterministic Policy Gradient (DDPG) [8, 12], the formulation however assumes a uni-modal shape of the advantage function, while other methods such as DDPG does not have any restrictions on the functions that can be represented.

1.4 Problem statement

Manipulation tasks that seem trivial to a human can be hard to learn for robots, especially from scratch without initial human demonstration,

due to high sample complexity. Recent research suggests ways to do this but are often based on that you know the poses of the objects and the end-effector. For some scenarios these are non-trivial to find out. Using a camera for pose detection has shown promising results but still assumes known camera offset or a fixed position of the camera.

1.5 This study

In this thesis, a series of experiments were conducted showing that NAF can learn simpler policies on a distributed real-world robotic setup. However, a pushing task with a clear multi-modal nature here fails to be learned using NAF both in a real-world setting, and in simulation. DDPG on the other hand learns a good policy in simulation, and the learned policy is successfully transferred to the real robotic setup. Pose estimation was done using a CNN in accordance with previous work [7, 10, 13] and further extended to evaluate whether a proposed pose estimation network can handle random camera poses.

Chapter 2

Background

2.1 Reinforcement learning

This entire section is descriptions of key concepts from a book on Reinforcement Learning by Sutton and Barto [16]. If no other reference is explicitly mentioned, statements are taken from this book.

2.1.1 The three tiers of machine learning

In reinforcement learning (RL) an agent interacts with an environment and tries to maximize how much *reward* it can receive from the environment. To maximize the reward in the long run might require short-time losses, making the problem more complex than just maximizing for one step at a time. To find a good strategy, commonly referred to as a *policy*, the agent uses its experience to make better decisions, this is referred to as *exploitation*. But, it must also find a balance between exploitation and to also try out new things, i.e. *exploration*. These features are specific for RL and therefore distinguishes it from supervised and unsupervised learning making it the third tier of machine learning.

2.1.2 Main elements of RL

Let S_t be the state at time t, R_t be the reward at time t, and A_t the action at time t. The interaction between an agent and its environment in RL is depicted in figure 2.1. At time step t, the agent reads the environment state and takes an action. The environment changes, maybe stochastically, by responding with a new state S_{t+1} and a reward R_{t+1} at

time $t + 1$.

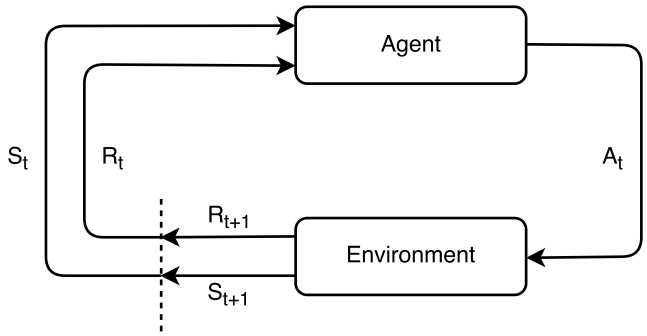

Figure 2.1: Agent and environment interaction in RL. S_t, R_t, and A_t is the state, reward, and action at time t [16].

The quantity to maximize is often not the immediate rewards, but rather the long term accumulated rewards. Let us call this quantity G_t, or *return*, for an agent at time t up to some time K:

$$G_t = \sum_{k=0}^{K} \gamma^k R_{t+k+1} \tag{2.1}$$

Some problems imply that K can go to infinity, and for $\gamma = 1$, G_t can theoretically take infinite values. It is obviously problematic to maximize something infinite and that is the reason for the $\gamma \in [0, 1]$ factor above that alleviates this problem if $\gamma < 1$ (proof omitted). For lower values of γ the agent tries to maximize short term rewards, and for larger values long-term rewards.

A policy is a function from the state of the environment to probabilities over actions, i.e. the function that chooses what to do in any situation. Since a reward is only short-term, a *value function* tries to estimate the total amount of reward that will be given in the long run for being in some state and following some policy. To enable planning of actions in the environment, RL algorithms sometimes use a *model* in order to explicitly build up an understanding of the environment. This is usually referred to as *model-based* RL in contrast to *model-free*.

2.1.3 Finite Markov Decision Processes

In a RL scenario where the environment has a finite number of states, there is a finite number of actions, and the Markov property holds is

called a *finite Markov Decision Process* (finite MDP). The dynamics of a finite MDP are completely specified by the probability distribution:

$$p(s', r|s, a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{2.2}$$

Important functions and terminology that is used throughout RL includes the *state-value function* (abbreviated as value function) and the *action-value function*. The state-value function with respect to some policy informally gives how good a state is to be in given that the policy is followed thereafter:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t|S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.3}$$

To compare the value of different actions in some state, given that you thereafter follow some policy π, is given by the action-value function:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t|S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.4}$$

According to RL theory there is always an optimal policy [17], i. e. that gives the highest possible expected return given any state. This is often denoted with $*$ and has the corresponding value and action-value functions $v_*(s)$ and $q_*(s, a)$. Given the optimal value or action-value function, it is (depending on the problem) easy to infer the optimal policy, therefore a common approach is to first approximate either of these functions.

2.1.4 Policy and value iteration

One exact method to find the optimal policy, at least in the limit, is called *policy iteration*. This builds on two alternating steps, the first called *iterative policy evaluation*. This estimates a value function given some policy and starts from a random value function v_0, except for any terminal state s_K for which are assigned $v_0(s_K) = 0$. By v_k is meant the value function estimate at iteration k. Then we iteratively update new value functions for each step:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum p(s', r|s, a) \left[r + \gamma v_k(s')\right] \end{aligned}$$

As can be seen, the dynamics $p(s', r|s, a)$ needs to be known, which of course is not always the case. The next step is called *policy improvement* and for this we first need to calculate the action-state function given the current policy π:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \end{aligned}$$

Given this, an improved policy π' is attained by:

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{2.5}$$

Iteratively performing these two steps will eventually converge to the optimal policy [18]. There is an alternative way that is done by only approximating the value function, called *value iteration*:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \end{aligned}$$

After convergence to some value function v, the optimal policy is found by:

$$\pi'(s) = \arg\max_a \sum_{s', r} p(s', r|s, a) \left[r + \gamma v(s')\right] \tag{2.6}$$

2.1.5 Monte Carlo methods and Temporal-Difference learning

Policy and value iteration are exploring the entire state-action space and finds an optimal policy if the dynamics of the environment are known. Sometimes we are dealing with samples from interacting with a system, and where we do not know the dynamics. For these cases, we can instead

estimate the action-value function given a policy. This can be done by *Monte Carlo methods* which in its simplest form is averaging of returns for samples that we have attained. The other method is *Temporal-difference methods* which estimates an error for each observed reward and updates the action-value function with this. To be more precise, one famous example of a time difference method is *Q-learning* and the updates are done according to:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (2.7)$$

Q-learning is an example of an *off-policy* method. This means that you can use a second, or derived, policy for exploration, but the algorithm still finds the optimal policy. The other family of methods is called *on-policy* methods and are characterized by that the expectation being maximized is with respect to the exploring policy. In contrast to this the expectation in the off-policy case is only dependent on the environment.

2.2 Neural networks

2.2.1 Basic idea

The simplest neural network could be considered to be a linear transformation of some data points X:

$$X_1 = WX + B \quad (2.8)$$

Since for some column in X, all its values influence all the values in the same column in X_1, the *input layer* X and *layer* X_1 are said to be *fully connected*. The convention of describing neural networks with layers comes from the inspiration of physical neurons arranged in layers although the description is not as apparent when describing the networks in this fashion. The above transformation is obviously restricted to learning linear transformations, so to learn non-linear functions, a non-linear *activation-function* is added:

$$X_1 = f(WX + B) \quad (2.9)$$

To learn more complex functions, transformations can be recursively stacked:

$$X_2 = f(W_1 X_1 + B_1) \tag{2.10}$$
$$... \tag{2.11}$$
$$Y_{k+1} = f(W_k X_k + B_k) \tag{2.12}$$

A loss function is a function $\ell : f, \mathbb{R}^d \longmapsto \mathbb{R}$, where f is the neural network and $\mathbb{R}^d$ is some data. Usually, the data are inputs to the network, often along with corresponding target values. This specifies the error, and implicitly the wanted behavior of the network. By all parts being differentiable we can specify the loss function and calculate its derivatives with respect to all the parameters $W, W_1, ...$ and $B, B_1, ...$ Using this we can then minimize the loss using gradient descent. A common and effective way to do this is called *back-propagation* [19]. The first layer X is commonly referred to as the input layer and the last layer as the output layer. The intermediate values are referred to as hidden layers.

2.2.2 Common activation functions

Two common activation functions include *tanh* and *Rectified Linear Unit* (ReLU) [20] shown in figure 2.2. The tanh function is defined as:

$$\tanh(x) = \frac{2}{1 + e^{-x}} - 1 \tag{2.13}$$

The ReLU is defined as:

$$\text{relu}(x) = \max(0, x) \tag{2.14}$$

Another addition to the family of activation functions is the *exponential linear unit* (ELU) [21] which has the benefit of propagating the gradient also for negative input values since the limit for large negative input values is -1 rather than 0 as for the ReLU. This was shown to speed up training and lowering classification test errors. The ELU is defined as (with α usually being set to 1.0):

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{2.15}$$

For classification networks, a common function to have in the output layer is the softmax function shown in figure 2.3. A property of the softmax is that all values will be output in the range $[0, 1]$ and sum

to one, making it conveniently interpreted as probabilities of different classes or categories. The mathematical definition is for a set $x_1, ..., x_K$:

$$\text{softmax}(x_k|x_1, ..., x_K) = \frac{e^{x_k}}{\sum_{i=1}^{K} e^{x_i}} \tag{2.16}$$

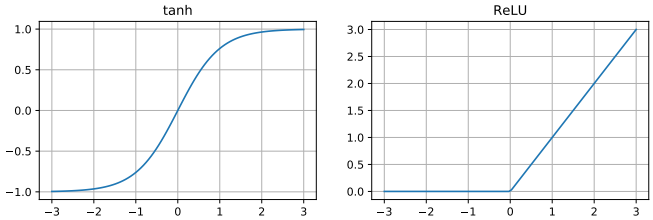

Figure 2.2: Common activation functions for neural networks.

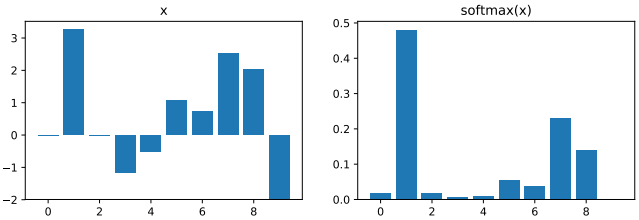

Figure 2.3: Softmax function commonly used to convert a set of numbers $\in \mathbb{R}$ to probabilities of a discrete set of categories.

2.2.3 Weight sharing

In order to construct effective models with less number of parameters the fact is used that some features are invariant of where in the data they are, e.g. the pattern $1, 2, 3, 4$ in the beginning, middle, or the end of the data. To this end, convolutions are used (where $*$ is the convolution operator):

$$y = w * x + b \tag{2.17}$$

These kind of networks are usually referred to as *convolutional neural networks* (CNN) [22]. The notation becomes a bit more tricky here since the convolutions are commonly done in two dimensions with 3-dimensional kernels (w). These are commonly used in neural networks

for images. It is common to use an other form of layer in conjunction with convolutional layers called *pooling* layers. These are used to reduce the dimensionality from one layer to another and are very similar to convolutions, but instead of kernels with varying weights, a max operation [23] or average operation [24] is applied (average pooling is a kernel with all weights being equal).

2.2.4 Optimizers

A method commonly used for optimizing neural networks is called Adam [25]. Updates to parameters of the function with loss function $f(\theta)$ are done by first and second order estimates of the gradients $g_t = \nabla_{\theta_t} f(\theta_{t-1})$. The current timestep is annotated t and starts with $t = 1$, this is used to bias-correct the first and second order estimates:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.18}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.19}$$

$$\mathbb{E}[g] \approx \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.20}$$

$$\mathbb{E}\left[g^2\right] \approx \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.21}$$

Here $\beta_1, \beta_2 \in [0, 1)$ are hyperparameters. The updates are done with stepsize α according to ($\epsilon > 0$ to ensure no division by zero):

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t - \epsilon}} \tag{2.22}$$

Adam can be seen as a generalization of RMSProp where $\beta_1 = 0$ [26] which implies that only a running second order estimate is used. This has been successfully used for training of on-policy algorithms [27] where gradients from previous policies included in the running mean might not be suitable for current policy updates. Earlier commonly used methods include Adadelta [28], Adagrad [29], and momentum [30].

2.2.5 Avoiding overfitted networks

A classical approach to regularize neural networks is adding a scaled L2-norm of the parameters to the loss. Another way shown to be effective, both for fully connected and convolutional neural networks, is called dropout [31]. During training time, units are with probability p set to zero and non-zeroed outputs are scaled by $1/p$. Effectively, this means inference during training time is done using random subgraphs of the neural network, and inference during test time is done by average voting from a collection of those subgraphs. To alleviate the problem of overfitting when using images, it is common to use data augmentation, i. e. applying random scaling, translations, noise etc. to the images (e.g. [1, 32, 33]). Another approach is to synthetically generate a larger image database by overlaying parts of the images with patches of other objects [34].

2.2.6 Batch normalization

As the weights in one layer changes during training, the following layers have to adapt to this change. In fact, later layers constantly have to adopt to changes in any of the previous layers during training, something called *internal covariance shift*. It was shown that this problem can be solved by adding intermediate normalization layers, called *batch normalization* [35]. These layers whiten the activations of the previous layer, i. e. element-wise subtraction of the mini-batch mean and divide by the square root of the variance. Since the statistics are calculated per mini-batch, they argue that this acts as a regularizer, and they empirically show that dropout in some cases are no longer needed. They argue that this is due to that the representation of one sample will shift differently depending of the other samples in the mini-batch. For some cases, whitening the outputs of the previous layer decreases what the next layer can represent, e. g. not saturating the sigmoid function in the subsequent layer. To alleviate this problem, the authors propose learnable parameters that ensure that the normalization layer, if needed, can represent the identity function. During inference, normalization is done on population estimates of the mean and variance. These population estimates are inferred using running mean and variance estimates attained during training. Using batch normalization showed to decrease the number of training steps by a factor of 14 for some cases, and improving the test errors of previous state-of-the-art networks.

2.3 Reinforcement Learning for Robotics

2.3.1 Q-function estimation using neural networks

For continuous state spaces, Q-learning can no longer be solved by tabular methods, and in practice many discrete state space problems are also too large to be represented and learned this way. Therefore it was proposed that the Q-function be estimated using a neural network updated in a standard Q-learning fashion, i. e. for one sample at a time after one interaction with the environment [36]. Here, the action maximizing the value going from the successor state is found by searching since there is a finite amount of actions. However this was shown to need several tens of thousands of episodes before convergence to optimal or near optimal policies [36]. Instead of updating the Q-function on-line one sample at a time, all state-action-successor state tuples (s, a, s') can be stored and used to update the network off-line using batches, which was found to converge faster [37].

2.3.2 Recent progress in the discrete action setting

For playing the board game Go, a four-step process was proposed [3]. The first step was to train a policy in a supervised fashion predicting expert moves. Thereafter, the policy was improved by playing games against previous iterations of the same policy and updating using policy gradient methods. A value function was then estimated given the best policy from the previous step, this was then used to perform Monte Carlo tree search to find the best action. The resulting policy/algorithm later beat the world champion Lee Sedol [38].

For continuous state spaces in contrast to Go, namely images from the screen of video games, a deep-Q-network (DQN) was proposed [4]. Here, a sequence of images (neighboring in time) from the games were input to a convolutional neural network. The final layer was a fully connected layer which outputs a state-action estimate for each action. This enabled faster selection of the optimal action due to one single forward pass of the network. The network was trained and evaluated on seven Atari games, surpassing a human expert in three of the games, and surpassing previous methods in six of the games.

2.3.3 Normalized Advantage Functions (NAF)

In order to extend Q-learning to continuous state and action spaces, Gu et al. [15] proposes a relatively simple solution called normalized advantage functions (NAF). They argue after doing simulation experiments that this algorithm is an effective alternative to recently proposed actor-critic methods and that it learns faster with more accurate resulting policies. First, the action-value function is divided into a sum of the value function V and what they call an advantage function $A : \mathbb{R}^d \longmapsto \mathbb{R}$:

$$Q(\mathbf{x}, \mathbf{u}) = A(\mathbf{x}, \mathbf{u}) + V(\mathbf{x}) \tag{2.23}$$

Here, $\mathbf{x}$ is the state of the environment and $\mathbf{u}$ are controls or actions. The advantage function is a quadratic function of u:

$$A(\mathbf{x}, \mathbf{u}) = -\frac{1}{2}(\mathbf{u} - \mu(\mathbf{x}))^T \mathbf{P}(\mathbf{x})(\mathbf{u} - \mu(\mathbf{x})) \tag{2.24}$$

There are more terms that need to be defined, but first consider equation (2.24). The matrix $\mathbf{P}$ is a positive-definite matrix, this makes the advantage function have its maximum when $\mathbf{u} = \mu(\mathbf{x})$. The purpose of μ is to be a greedy policy function, thus $\mathbf{Q}$ is maximized when $\mathbf{u}$ is the greedy action. The purpose of this is that given an optimal Q, we do not need to search for the optimal action, since we know μ. Now the definition of $\mathbf{P}$:

$$P(\mathbf{x}) = \mathbf{L}(\mathbf{x})\mathbf{L}(\mathbf{x})^T \tag{2.25}$$

Here, $\mathbf{L}$ is a lower-triangular matrix where diagonal entries are strictly positive.

After these definitions, we are left with estimating the functions V, μ, and $\mathbf{L}$. To this end the authors use a neural network, here shown in figure 2.4. The $\mathbf{L}$ output is fully connected with the previous layer and not passed through an activation function (it is linear). The diagonal entries of L are exponentiated. Hidden layers consisted of 200 fully connected units with rectified linear units (ReLU) as activation functions except for $\mathbf{L}$ and A as already defined.

The NAF algorithm is listed in algorithm 1. All collected experiences are stored in a replay buffer that optimization is run against. Exploration is done by adding noise to the current greedy policy μ.

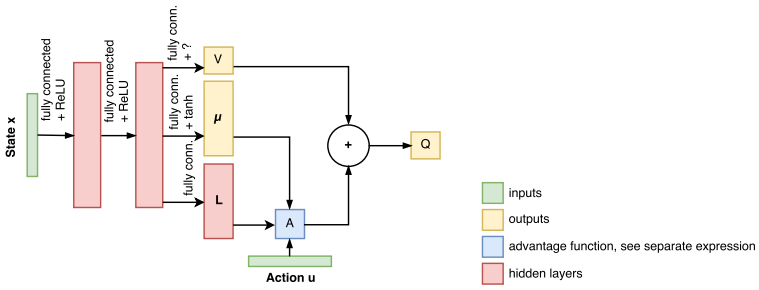

Figure 2.4: Neural network design for NAF [15]. The activation function to V was not specified. The tanh activation was added in figure due to Gu et al. [8] using it in order to have bounded actions for safety reasons.

Algorithm 1 NAF algorithm

Randomly initialize network $Q(x, u|\theta)$
Initialize target network Q', $\theta' \leftarrow \theta$
Initialize replay buffer $R \leftarrow \emptyset$
for episode $= 1$ to M **do**
  Initialize random process $\mathcal{N}$ for action exploration
  Receive initial exploration state x_1
  for $t = 1$ to T **do**
    Select action $\mathbf{u_t} = \mu(\mathbf{x_t}|\theta) + \mathcal{N}_\mathbf{t}$
    Execute $\mathbf{u_t}$ and observe r_t and $\mathbf{x}_{t+1}$
    Store transition $(\mathbf{x}_t, \mathbf{u}_t, r_t, \mathbf{x}_{t+1})$ in R
    for iteration $= 1$ to I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\mathbf{x}_{i+1}|\theta')$
      Update θ by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(\mathbf{x}_i, \mathbf{u}_i|\theta))^2$
      Update target network $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
    end for
  end for
end for

2.3.4 Distributed real-world learning using NAF

Real-world experiments were done by Gu et al. [8] on door opening tasks using the NAF algorithm and 7-DoF torque controlled arms. They extended the algorithm to be distributed on several robots/collectors and one separate trainer thread on a separate machine. They state that this was the first successful real-world experiment with a relatively high com-

plexity problem without human demonstration or simulated pretraining. They used the layout of the network shown in figure 2.4 but with hidden layers of 100 units each. Also, in this article it was explicitly mentioned that the activation functions for the policy μ was tanh in order to bound the actions. The state input consisted of the arm pose and target pose. The target pose was known from attached equipment. The modified version of NAF is listed in algorithm 2.

The authors conclude that there was an upper bound on the effects of parallelization, but hypothesize that the speed of the trainer thread has a limiting factor in this matter. They used CPU for training the neural network so instead using a GPU might increase the effect of more collectors.

Algorithm 2 Asynchronous NAF - N collector threads and 1 trainer thread

// trainer thread
Randomly initialize network $Q(\mathbf{x}, \mathbf{u}|\theta)$
Initialize target network Q', $\theta' \leftarrow \theta$
Initialize shared replay buffer $R \leftarrow \emptyset$
for iteration $= 1$ to I **do**
  Sample a random minibatch of m transitions from R
  Set $y_i = \begin{cases} r_i + \gamma V'(x_i|\theta) & \text{if } t_i < T, \\ r_i & \text{if } t_i = T \end{cases}$
  Update θ by minimizing the loss: $L = \frac{1}{m}\sum_i (y_i - Q(\mathbf{x}_i, \mathbf{u}_i|\theta))^2$
  Update the target network: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
end for
// collector thread n, $n = 1...N$
for episode $= 1$ to M **do**
  Sync policy network weights $\theta_n \leftarrow \theta$
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state x_1
  for $t = 1$ to T **do**
    Select action $\mathbf{u}_t = \mathbf{mu}$
    Execute $\mathbf{u}_t$ and observe r_t and $\mathbf{x}_{t+1}$
    Send transition $(\mathbf{x}_t, \mathbf{u}_t, r_t, \mathbf{x}_{t+1}, t)$ to R
  end for
end for

2.3.5 Deep deterministic policy gradient

An alternative continuous Q-learning method to the NAF formulation is an actor-critic approach called Deep Deterministic Policy Gradient (DDPG) [12]. Separate networks are defined for the Q-function (critic) and a deterministic policy μ (actor), and the target quantity for the critic to predict becomes:

$$Q^{\mu}(s_t, u_t) = \mathbb{E}_{s_{t+1}} \left[r_t + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1})) \right] \tag{2.26}$$

The Q-function is trained by optimizing the temporal difference error, and the policy is trained by updating its parameters θ_μ according the policy gradient. The policy gradient in equation (2.28) was shown to be the gradient of the expected value of the return with respect to its parameters [39]. The expected return is below notated J.

$$\nabla_{\theta_\mu} J = \nabla_{\theta_\mu} \mathbb{E}_s \left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \right] \tag{2.27}$$

$$= \mathbb{E}_s \left[\nabla_u Q(s, \mu(s)) \nabla_{\theta_\mu} \mu(s|\theta_\mu) \right] \tag{2.28}$$

Note that the product in the expectation in equation 2.28 is a matrix multiplication. The left hand side is a $1 \times n_u$ matrix where n_u is the dimensionality of the actions. The right hand side is a $n_u \times n_\theta$ matrix, where n_θ is the number of parameters in the actor model. The expectation is over the states s in the environment, and can be approximated by drawing N samples from interactions with the environment:

$$\nabla_{\theta_\mu} J = \frac{1}{N} \sum_{n=1}^{N} \nabla_a Q(s, \mu(s)) \nabla_{\theta_\mu} \mu(s|\theta_\mu) \tag{2.29}$$

Since the Q-network fits to its own values in a recursive way, it was experienced to be unstable. This was solved by having separate target networks Q' and μ'. Mean square error loss L of the temporal differences errors were calculated by:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1})) \tag{2.30}$$

$$L = \frac{1}{N} \sum_{i=1}^{N} \left[y_i - Q(s_i, u_i)) \right]^2 \tag{2.31}$$

Target networks were then slowly updated by $\tau < 1$:

$$\theta_{\mu'} = \tau\theta_{\mu} + (1-\tau)\theta_{\mu'} \tag{2.32}$$

$$\theta_{Q'} = \tau\theta_{Q} + (1-\tau)\theta_{Q'} \tag{2.33}$$

The networks were setup to have two hidden layers of 400 and 300 units each. All hidden layers had ReLU activation functions, output of the action network was bounded by using tanh activation functions. The method was shown to solve a variety of task with continuous controls. Instead of performing the matrix multiplication in equation (2.29) explicitly when using computational graph libraries, actor parameters θ_μ are easily updated by maximizing the quantity (keeping critic parameters θ_Q fixed):

$$Q(s, \mu(s|\theta_\mu)|\theta_Q) \tag{2.34}$$

2.3.6 Guided Policy Search

Guided policy search (GPS) [40] maximizes the expected return $J(\theta)$ using trajectories $\zeta^i_{1:T} = \{(x^i_1, u^i_1), ..., (x^i_T, u^i_T)\}$ and the respective rewards $\{r^i_1, ..., r^i_T\}$. The superscript i denotes the sample. In order to use trajectories from previous policies, importance sampling is used and the quantity to maximize is becomes:

$$\mathbb{E}[J(\theta)] \approx \sum_{t=1}^{T} \frac{1}{Z(\theta)} \sum_{i=i}^{m} \frac{\pi_\theta(\zeta^i_{1:t})}{q(\zeta^i_{1:t})} r^i_t \tag{2.35}$$

Here $q(\zeta^i)$ is the probability of the trajectory ζ^i under the distribution it was sampled from. The distribution π_θ gives the probability of a sequence of the current policy parameterized by θ. The GPS algorithm iteratively maximizes this expectation and provides new guiding samples by optimizing sampled trajectories using methods such as *iterative Linear Quadratic Regulator* (iLQR) [41] or Policy Improvement with Path Integrals ($\mathbf{PI}^2$) [42]. The found locally optimal trajectories can be added as guiding samples in the above expectation.

2.3.7 Asynchronous Advantage Actor-Critic (A3C)

An on-policy actor-critic method was proposed that distributes the training and exploration in simulated environments over several CPU cores [27]. Each thread executes a sequence of actions and then calculates the

gradients, given that sequence, which are then sent to a central parameter thread. The actor output $\pi(a|s)$ is parameterized as a Gaussian distribution with a mean vector μ and covariance matrix $\sigma\mathbf{I}$ where μ and σ are outputs from the actor network. Here $\pi(a|s)$ is the probability of the action a given some state s. During exploration, actions $a_1, ..., a_T$ given states $s_1, ..., s_T$ are sampled from this distribution and the quantity being maximized is:

$$\ell(a_{1:T}, s_{1:T}, \theta_\pi, \theta_V) = \sum_{t=1}^{T} \log \pi(a_t|s_t, \theta_\pi)(R_t - V(s_t|\theta_V)) \tag{2.36}$$

Here V is the critic network, and R_t is the estimated return given by:

$$R_t = \begin{cases} r_t + V(s_{t+1}|\theta_V) & \text{if } t = T \\ r_t + R_{t+1} & \text{if } t < T \end{cases} \tag{2.37}$$

In the central thread, gradients are applied to the parameters and local models then update their parameters by querying the central thread. This was shown to solve a range of manipulation tasks in simulation, although the authors are somewhat vague about the results for the continuous action case. For the discrete action case, the formulation differs somewhat, but results show a substantial cut in training time on Atari games when using 16 CPU cores compared to the original method where a GPU is used [4].

2.3.8 Prioritized experience replay

When randomly sampling experiences from a replay buffer, one alternative is to sample from a uniform distribution. However, if samples are sampled from a distribution where experiences with larger temporal-difference errors are more likely to be chosen, training times can be reduced with resulting policies that outperform those trained by uniform sampling [43]. Let R_t be the reward at time t, $Q(S_t, A_t)$ the Q-function of state S_t and action A_t at time t and $V(S_t)$ the value function for S_t at time t. Given a sample tuple $x_i = (S_t, S_{t+1}, A_t, R_{t+1})$, let the temporal-difference error for this sample to be defined as:

$$\delta_i = Q(S_t, A_t) - [R_{t+1} + \gamma V(S_{t+1})] \tag{2.38}$$

Define a priority p_i of sample x_i as:

$$p_i = |\delta_i| + \epsilon \tag{2.39}$$

Here $\epsilon > 0$ ensures all priorities are strictly positive. Sample experiences according to the probability:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.40}$$

The hyperparameter $\alpha \geq 0$ enforces more uniform probabilities for values close to zero and more non-uniform probabilities for larger values.

Since the gradient is biased by a biased choice of samples, the magnitude of the gradient with respect to each sample x_i can be weighted by:

$$w_i = \left(\frac{1}{P(i)}\right)^\beta \tag{2.41}$$

The parameter $\beta \geq 0$ can be varied during training, and it is argued that the unbiased gradients are more important towards the end when the policy is converging. They varied this parameter reaching $\beta = 1$ only towards the end of the training. It is not clear from the article but it is assumed that training starts with $\beta = 0$.

2.4 Spatial softmax in pose estimation

Levine et al. [13] proposed an architecture for a CNN that gives pose estimates for robotic manipulation tasks. After the last convolutional layer, a softmax is applied, but only normalized over each kernel's response map, called *spatial softmax*:

$$s_{cij} = \frac{e^{a_{cij}}}{\sum_{i',j'} e^{a_{ci'j'}}} \tag{2.42}$$

Here, a_{cij} is the output of the c:th kernel at coordinate ij. After this, they calculate the expected 2D position for each feature, which they argue is better suited for pose estimation. The expected 2D position is expressed as a tuple (f_{cx}, f_{cy}) calculated according to:

$$f_{cx} = \sum_{i,j} s_{ij} x_{ij} \tag{2.43}$$

$$f_{cy} = \sum_{i,j} s_{ij} y_{ij} \tag{2.44}$$

The scalar value x_{ij} is the position in image space of the pixel at coordinate (i, j). This can reasonably easy be simplified to a matrix multiplication with constant weights from each of the response maps. Arguably, it could also be possible to rewrite the above expressions as:

$$f_{cx} = \sum_{i,j} i s_{ij} \tag{2.45}$$

$$f_{cy} = \sum_{i,j} j s_{ij} \tag{2.46}$$

As a measure of certainty of the expected position, is was proposed to use the spatial softmax output at the expected position [44]. Other possible methods are naturally the estimated variance of the 2D position as well as the maximum output of the spatial softmax.

Chapter 3

Related work

This section will present related research findings, starting with articles related to reinforcement learning. An overview of some related work with pose estimation will then be discussed.

3.1 Reinforcement learning for robotic manipulation

Improvements on previous methods for Atari games using an on-policy actor-critic method called *Asynchronous Advantage Actor Critic* (A3C), where simulation and gradient calculations were distributed on multiple cores on a CPU, were shown to result in a reduction in training time from 8 days on GPU to 1 day compared to original results. A stochastic actor output was parameterized as a Gaussian distribution with mean vector μ and covariance matrix $\sigma\mathbf{I}$. The algorithm was also successfully used on continuous action/state-space 2D-reaching and 3D-manipulation tasks using simulated robots [27]. Using an off-policy actor-critic method called *Deep Deterministic Policy Gradient* (DDPG) [12], where the actor output instead is deterministic, solves similar tasks in simulation. This was later also shown to be capable of learning in real-time on real robotic systems on a door opening task [8] where data collection was distributed over a collection of robots. The door opening task was however learned faster in the same setting using a *Normalized Advantage Functions*-algorithm (NAF) suggested by Gu et al. [15] where the advantage function is parameterized as a quadratic expression, giving the Q-function estimate an easily accessible known global maximum. Door

opening tasks using NAF and DDPG used known door poses and arm poses from attached sensor equipment as inputs. Chebotar et al. [10] demonstrates door opening tasks initialized from human demonstration using GPS and Policy Improvement with Path Integrals ($\mathbf{PI}^2$) [42]. They demonstrate that they can learn this task using pose estimation from visual input. This is extended by Yahya et al. [7] to be trained simultaneously on several robots. In both cases, torque commands were the output of a neural network where the first part takes visual input and is pretrained with pose estimates of the door and the robot arm. The quality of the trained policies were evaluated when the camera position was changed by $4 - 5$ cm translations and was shown to half the performance. The neural network is shown in figure 3.1. The above methods are all summarized in table 3.1.

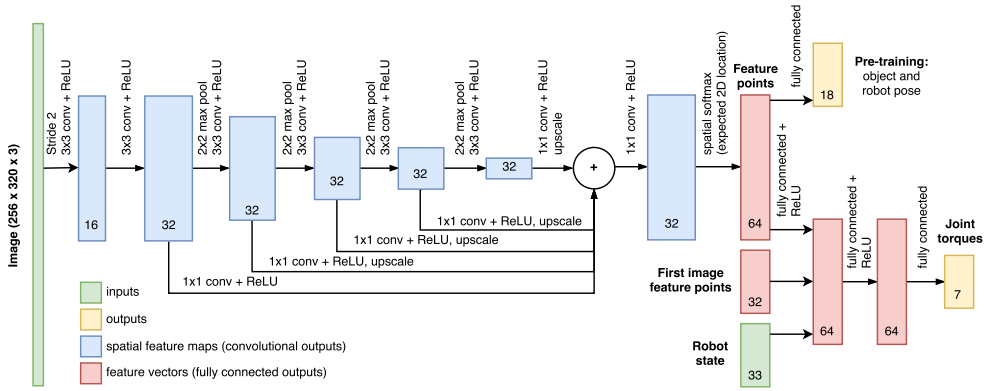

Figure 3.1: Network used for door opening task from visual inputs [10]

Algorithm	**Actor/ Critic**	**-policy**	**Possible limitations**	**Deterministic action output**	**Empirical results**
NAF [15]	No	Off	Quadratic advantage function	Yes	Real-world trained door opening without human demonstration where it performed better than DDPG
DDPG [12]	Yes	Off	Possibly unstable	Yes	Real-world trained door opening, no demonstration
A3C [27]	Yes	On	Uni-modal gaussian action distribution	No	Substantial speedup on training of Atari games using distributed simulations and gradient calculations, also solved continuous tasks in simulation
GPS [40]	No	Off	Uni-modal gaussian action distribution	No	Real-world trained door opening, from demonstration

Table 3.1: Comparison between different reinforcement learning algorithms used for robotic manipulation tasks.

3.2 Pose estimation

3.2.1 Predicting poses as 2D image coordinates

Since 2D coordinates have been shown to sufficient for 3D pose estimation [10], alternative methods could include first regressing to known 2D image coordinates as a pre-training step. Directly regressing to poses labeled in the image have been successfully implemented by first creating Gaussian heat-maps as targets and then regressing last layer convolution feature maps directly to these heat-maps [45, 46]. These networks were used to predict human poses in the image frame, defined as position of joints and other key points like the nose. The networks successfully learned not only visible parts, but could infer parts faced away from the camera. State-of-the-art performance was attained by stacking several identical (regarding architecture) networks called "hourglass"-modules. The modules apply a series of convolutions, followed by a series of upscalings, to output the same shape as the input. Shortcut connections are also introduced between feature maps of the same size, passed through $1x1$ convolutions [45].

3.2.2 Human joint 3D relative pose prediction

Research by Park et al [14] regresses 3D joint poses relative to a root joint from images by using a convolutional neural network. The network ends with two separate fully connected parts, one which regresses to the 2D image coordinates of each joint, and one which regresses to the 3D relative joint pose. They argue that by simultaneously training the network with 2D and 3D labels, this relationship is implicitly learned.

Chapter 4

Method

I present in the following sections a high-level description of the process, followed by describing environments, derivations, tools, and equipment.

4.1 Research question

How can deep and distributed reinforcement learning be used for learning and performing dynamic manipulation tasks in a stochastic environment with unknown poses.

4.2 The task and partial goals

The final goal, or task, to accomplish was pushing a cube from and to some arbitrary positions in the workspace of the robot. The task was researched using a reinforcement learning approach, only specifying the wanted behavior by defining a reward function $f(s, a, s') \in \mathbb{R}$, where s, a, s' is the state, action, and successor state. The state here includes the robot and cube state along with the goal. First-order Markov property is assumed together with assuming the state as observable, even if observed with uncertainty in a higher dimensional space, formulating the problem as a Markov Decision Process (MDP). This process is modeled with discrete time steps, with the state space being the 2D coordinates in a horizontal plane of the robot end-effector, center of the cube, and the goal position. The action space is continuous and consists of bounded 2D relative movement of the end-effector.

An iterative approach was taken in the process of solving the task,

starting by solving simpler tasks before increasing the complexity. This way algorithms are sanity checked in a natural fashion and making limitations of model formulations clearer as complexity is added. The first task was to investigate appending arbitrary goals to the state for a reaching task with a simulated agent and environment. This was later extended to being trained on data from distributed real robotic systems. Pushing tasks were then attempted in simulation followed by evaluating trained policies on real robots. As a second part to this thesis, pose estimation from camera was researched with the goal to relax the assumption of a fixed position camera or known offset to the camera. This was also done iteratively starting with a fixed position camera and then using simulations to investigate neural network capabilities given perfect features. Finally, trained pose estimation models were evaluated on the real robotic systems, replacing pose estimates from other sensors. The exact steps could of course not be known in advance, but would depend on the partial results. An overview of the final approach and the overall steps are shown diagrammatically in figure 4.1.

4.3 Robotic environment

Low cost robotic arms (uArm Metal) with reported repeatability of ± 5 mm and noisy sensors were used. The robot arms were used along with a dedicated computer, and a *LIght Detection And Range*-device (LIDAR) for pose estimation of the cube. The LIDAR also return noisy distance estimates, described in more detail in section 4.5. For pose estimation from camera, an RGB-camera with an optional RGB-aligned depth channel was used. A workspace for the entire setup was defined to be $x \in [-0.1, 0.1]$, $y \in [0.1, 0.3]$ (meters), both for the reaching and pushing task. The coordinate frame used is shown in figure 4.2 (top left), along with the placement of the LIDAR and camera (top right).

The arms were in this thesis controlled by commanding servo angles, with an implemented controller on top for commanding the end-effector to some cartesian coordinates. The arms have 4 degrees of freedom, but only three were used in this thesis, ignoring the end-effector rotation servo. The arms were shipped with controllers and forward/inverse kinematics but were not reliable, therefore a new derivation of the kinematics was done along with a new controller implementation.

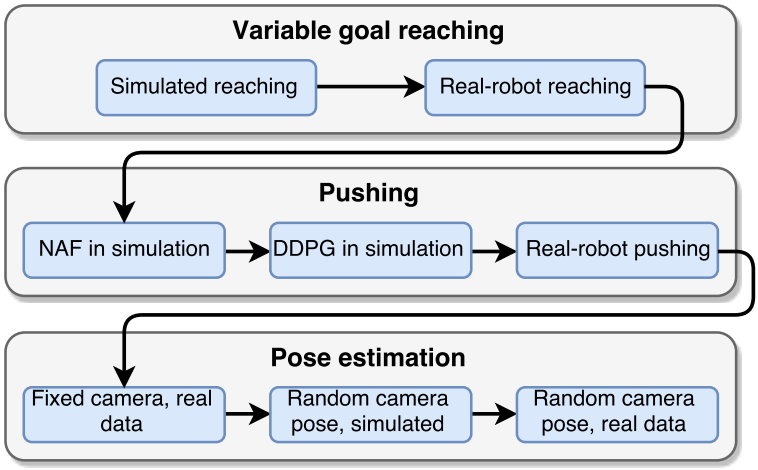

Figure 4.1: Diagram of the working process. The first part was to investigate setting arbitrary goals and appending those as a part of the state, first in simulation and then on the real robotic system. Next, complexity is added by attempting to solve the pushing task. Attempts were made in simulation, and then finally on the real robots. Final part was pose estimation, starting with a fixed position camera. For random and unknown camera positions, an ideal setup was first experimented with in simulation and then extending it to autonomous feature extraction and inference from RGB images.

4.3.1 Inverse kinematics derivation

In order to implement a controller for cartesian movement of the arm, inverse kinematics of the robot were derived. Definition of landmarks and distances can be seen in figure 4.3. We are given the coordinates of the point D and want to find to angles α, β, and θ.

The angle θ of the bottom servo that rotates the arm around the z-axis is first simply found by:

$$\theta = \arctan2(y, x) \tag{4.1}$$

The problem can then be simplified by only considering the 2D plane defined by the points A, B, C, where the origin is placed at A. Denote the coordinates of C in this new frame as:

$$r = \sqrt{x^2 + y^2} - d_2 - d_3 \tag{4.2}$$

$$h = z - d_1 - d_4 \tag{4.3}$$

Distances d_{AB} and d_{BC} are known constants, d_{AC} is simply $\sqrt{r^2 + h^2}$. Define the part of the α angle pointing towards C from A as γ. This is found by:

$$\gamma = \arctan2(h, r) \tag{4.4}$$

All sides of the triangle ABC are known, so the angles are found by the rule (applied in the same way to all angles):

$$\cos(BAC) = \frac{d_{AB}^2 + d_{AC}^2 - d_{BC}^2}{2 d_{AB} d_{AC}} \tag{4.5}$$

The angle α is found by:

$$\alpha = BAC + \gamma \tag{4.6}$$

The angle between the vertical line going down from A and AB is $\alpha - \frac{\pi}{2}$ which implies:

$$\beta = \frac{\pi}{2} - ABC - (\alpha - \frac{\pi}{2}) = \pi - ABC - \alpha \tag{4.7}$$

4.4 Simulated environment

In order to do faster evaluations of algorithms, and constructing more ideal environments with less noise etc., simulation experiments were also done. For this, a simple 2D environment was implemented with a circular pushable object, and a point-like manipulator. Actions were sent to the environment as relative movement of the manipulator. The environment does not take friction into account, only moving the pushable object the minimum distance such that the manipulator never exists within the object boundaries. Positions of goal, manipulator, and object could be sampled according to different sampling strategies, in the full problem each position is sampled uniformly within the workspace. Samples of this environment is shown in figure 4.4. After each interaction, the environment returns successor state and reward. For reaching tasks in simulation, the pushable object is simply ignored.

4.5 Estimating the cube position using LIDAR

A LIDAR was used to estimate the position of the cube. These estimates were both used as ground truth for training pose estimation neural networks, and used as a part of the state for the reinforcement learning algorithms. The LIDAR gives 360 distance measurements at approximately evenly spaced angles, with the angles having some variation from sweep to sweep. By sweep here is meant a new set of 360 measurements from a full rotation of the laser. Sweeps are returned at 10 Hz. Only scans inside the defined workspace (section 4.3) were considered for pose estimation of the cube. Only the cube is assumed to reflect light from the LIDAR, the end-effector of the robot was placed at a z-coordinate such that it is above the plane of the LIDAR-scans.

To estimate the cube pose, the Hough-transform was used [47]. This algorithm takes as input a 2D matrix and returns scalar values for a set of angles and distances $\{\theta_1, ..., \theta_M\} \times \{d_1, ..., d_N\}$ each defining a line in the plane, see figure 4.5. A large scalar for some θ_i and d_i means that the matrix entries along the coordinates specified by these parameters have higher values, i.e. pixels form a line along the parameterized line.

The angles with corresponding distance measures from the LIDAR are converted to cartesian space and plotted as ones onto a matrix of zeros (Hough transform input), an example is shown in figure 4.5 where the

black pixels correspond to LIDAR scans. After applying the transform on this image, the line with the highest corresponding scalar value is chosen, in figure 4.5 shown as the red line. One of the cube's sides can then be estimated using this line, limited by the outermost scans lying approximately on this line, see red dots in figure 4.5. The center of the cube, which is $4 \times 4 \times 4$ cm, is then estimated by adding the orthogonal vector of length 2 cm from the center of the estimated cube side (shown as blue dot). Only the center of the cube was used for the experiments, although the full pose of the cube is easily inferred from the found line.

The scans from the LIDAR showed variation between sweeps resulting in variations in the pose estimates. This can be seen in figure 4.6. The noise being relatively large imposes difficulties in training policies using this data, and also for training neural network pose estimators using LIDAR estimates as ground truth. Averaging estimates is one solution to lowering errors but implies slower updates of the pose estimation.

4.6 Software and libraries

For high-level computational graphs, Keras [48] was used. For lower-level extensions not supported by Keras, Theano was used [49]. For image pre-processing, plotting, and overall linear algebra and mathematical computations, SciPy was used [50].

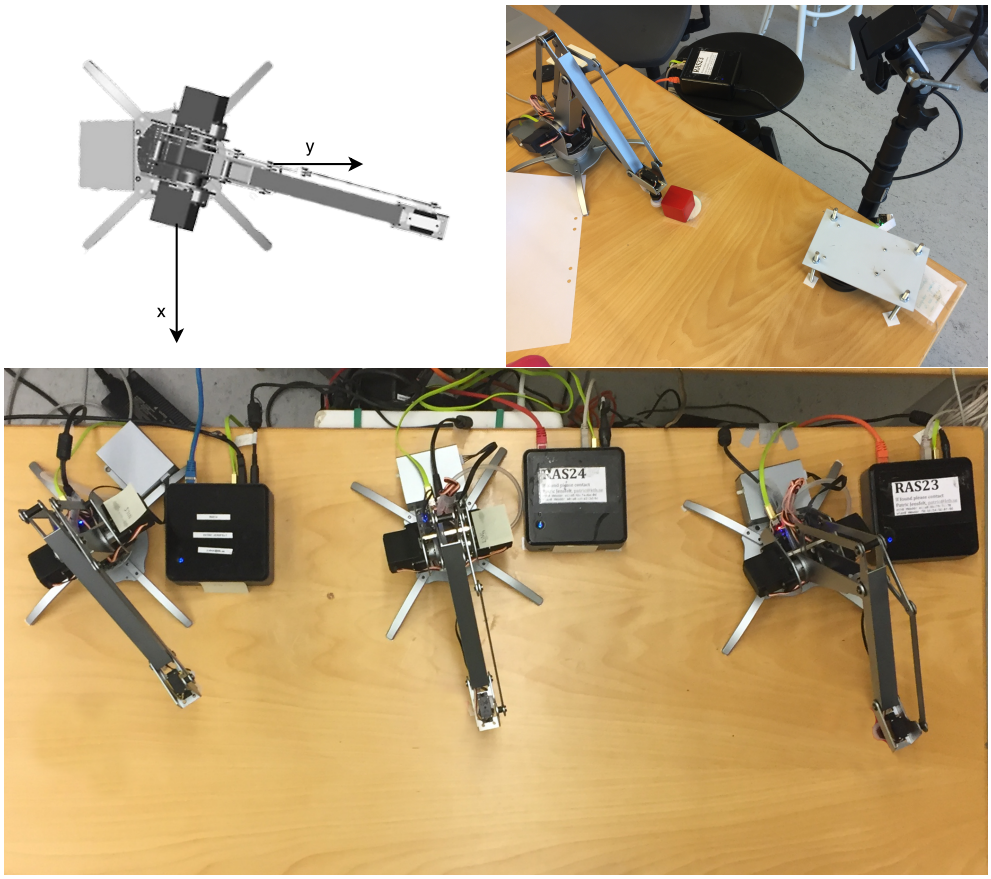

Figure 4.2: Top left: Coordinate frame used for the robot. Top right: Placement of the LIDAR, seen attached to a gray housing in the bottom right corner. The housing was built in order to mount the LIDAR such that it only registers reflecting light from the cube. The fixed camera position for cube pose estimation can also be seen in the top right corner of this picture. Bottom: For distributing data collection, several setups, each with a dedicated computer, was used.

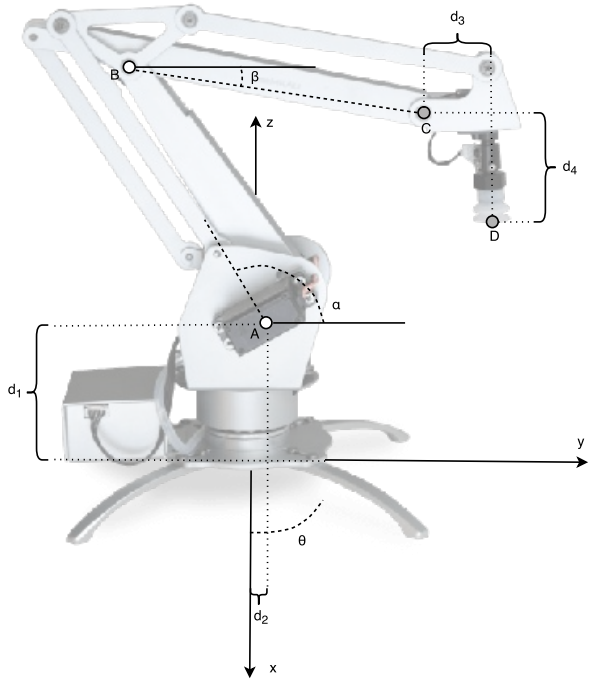

Figure 4.3: Notation for landmarks used in derivation of inverse kinematics. A simplified problem is to consider the angles of the triangle ABC.

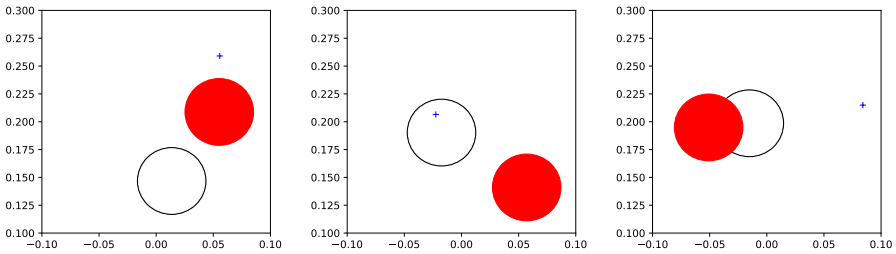

Figure 4.4: The simulated environment for pushing, the cross represents the robot end-effector, red circle the pushable object, and the black/white circle the goal position.

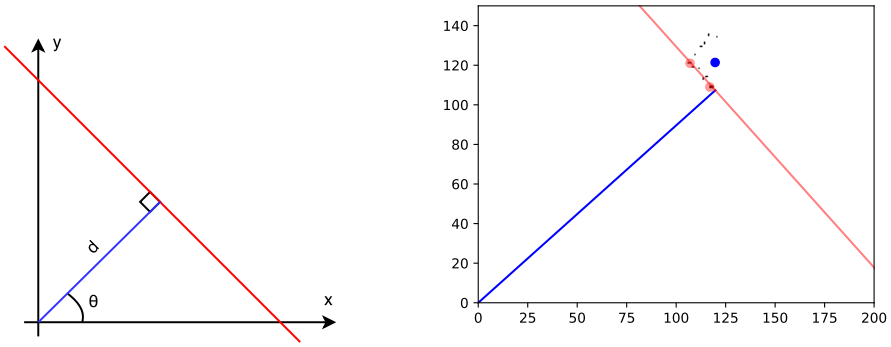

Figure 4.5: Left: The Hough transform returns scalar values for a set of lines each parameterized by some θ and d. In this figure the red line is parameterized by the angle θ and distance d. Right: LIDAR scans are plotted onto a binary matrix. The parameters with the maximum output from the Hough transform corresponds to the red line. The side of the cube is estimated by taking the outermost scans approximately lying on this line. The center of the cube is then found by adding an orthogonal vector from the center of this side with length corresponding to half of the cube size.

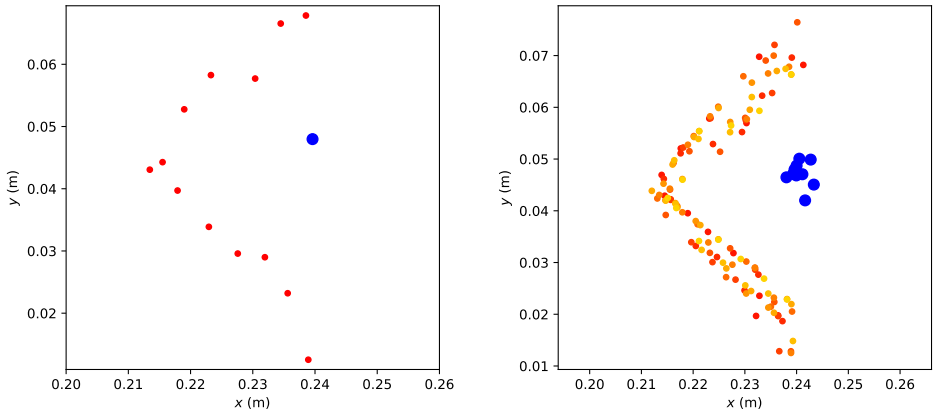

Figure 4.6: LIDAR scans (red) on a cube with the corresponding position estimate (blue). To the left one single sweep with the LIDAR clearly showing noisy measurements. To the right, plotting consecutive scans for the same cube position along with cube pose estimates shows noisy measurements and also noisy pose estimates of the cube. The different colors of the scans to the right represent different sweeps and show that the noise does not have a bias that is unique to the current sweep.

Chapter 5

Reaching I: Simulated reaching task

The pushing task defined puts no assumption on a constant goal position, rather this should be able to be put randomly within the workspace. By appending the goal position as part of the state, theoretically the goal could be constantly moving and the policy should adjust accordingly. To investigate whether a neural network easily can represent this policy, experiments were first performed where the target is to reach randomly set goal positions with the end-effector. If this cannot be accomplished in simulation, it is unlikely to be accomplished on a real robotic system. The task was therefore learned in a simulated environment first. The NAF algorithm was used since it showed promising results on the real world door opening task without the need for human demonstrations, and it has a straightforward way of distributing it should it succeed.

5.1 Method

5.1.1 Definition of the MDP

The set of states

$$\mathcal{S} \subset \mathbb{R}^2 \times \mathbb{R}^2 \tag{5.1}$$

is the cartesian product of end-effector end goal positions. The set of actions

$$\mathcal{A} = \{\mathbf{a} \in \mathbb{R}^2 \mid ||\mathbf{a}|| < 0.05\} \tag{5.2}$$

is interpreted as the relative movement of the end-effector, although the resulting position might vary due to a stochastic environment. The dynamics is most succinctly described in two parts, the first part being the function giving a successor end-effector position $\mathbf{e}'$ given previous position $\mathbf{e}$ and action $\mathbf{a}$:

$$\mathbf{e}' = \epsilon(\mathbf{e} + \mathbf{a}), \epsilon \sim \mathcal{N}(1, 0.1^2) \tag{5.3}$$

The goal $\mathbf{g}$ is constant during an episode which for formal completeness is stated:

$$\mathbf{g}' = \mathbf{g} \tag{5.4}$$

The reward is a function of the successor state

$$r(\mathbf{e}', \mathbf{g}') = \begin{cases} -2 & \text{if } \mathbf{e}' \text{ outside workspace} \\ \exp\left(-k||\mathbf{e}' - \mathbf{g}'||^2\right) - 1 & \text{otherwise} \end{cases} \tag{5.5}$$

Setting $k = 1000$ makes the reward equal 0 close to the goal and rapidly decay to -1 further from the goal. The value of k was chosen to be 1000 because it was seen, by plotting the reward function, to be large enough to create a clear peak at the goal position, and small enough to avoid having close-to-zero gradients of the reward function at other positions in the workspace. The discount factor γ was defined to be equal to 0.98.

5.1.2 Environment

A very simple simulated environment was used, as described in section 4.4, here ignoring the pushable object. This leaves only workspace boundaries and an end-effector that can be controlled by 2D cartesian relative movements. The environment capped commands with norm larger than 5 cm. Commands were, in the environment, multiplied by Gaussian noise $\mathcal{N}(1, 0.1^2)$ according the MDP definition, and the environment was reset when reaching the outside of the workspace or getting within a 1 cm radius of the goal. When resetting the environment, end-effector and goal poses were randomly sampled within the workspace. The goal pose remained the same until the environment was reset.

5.1.3 Algorithms

A NAF neural network was implemented with the same layout as described in section 2.3.4 with two hidden layers of 100 units each. The

two dimensional μ output had a tanh function scaled by 0.05 as activation. These action outputs are not strictly correct according to the above definition of the MDP, but this parameterization made more sense than for example using polar coordinates. All poses were 2D where end-effector and goal pose were concatenated as input to the network. A discount factor of 0.98 was used and the Adam optimizer [25] was used with learning rate 0.0001 and a batch size of 512. The replay buffer was sampled from as described in section 2.3.8 with $\alpha = 1$ and β in iteration i out of a total amount of iterations i_{tot} was set according to:

$$\beta_i = \exp\left(10(i - i_{tot})/i_{tot}\right) \tag{5.6}$$

For sampling from the replay buffer, a binary tree was used where the value of a parent equals the sum of its children [43]. This way, drawing one sample from a total of N samples is $\mathcal{O}(\log_2(N))$. The exact procedure is to first draw a sample from $U(0, \sum_i p_i)$ and then start from the top of the tree and recurse down to the corresponding leaf node. The loss was defined as the mean square error of the temporal-difference errors. The training process was alternating between generating new experiences by interacting with the environment, and between sampling batches from the replay buffer to optimize the neural network. Explicit criterion for a successful policy were not defined, rather training was run until plots of the policy showed to move the end-effector towards the goal position from all directions.

5.2 Results & Discussion

The algorithm was running for approximately one hour collecting $\approx 100k$ state transitions. The learned policy and value functions are shown in figure 5.1. The implementation, using the NAF formulation, clearly solves the task and is able to handle changing goal positions given as a part of the state. The value function estimates where less peaky around the goal position than expected, but has to do with the formulation of the reward. The reward being calculated from successor states, and a single action allowing the agent to move 5 cm, gives a plateau of approximately 5 cm around the goal from where the goal, and the maximum immediate reward, is reachable within one action.

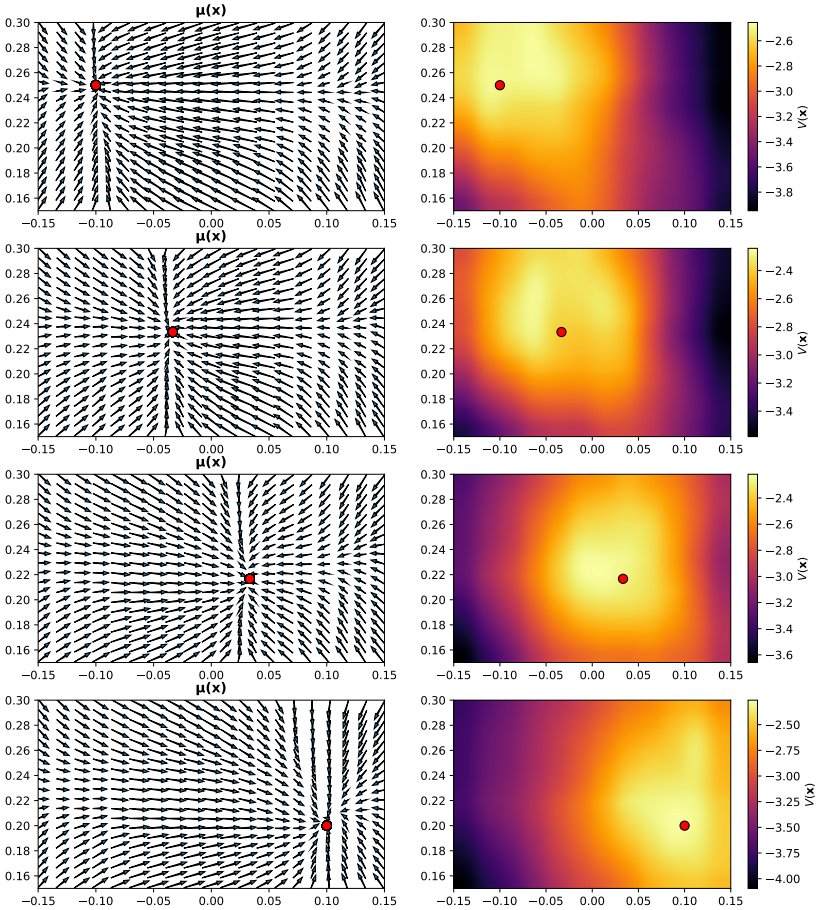

Figure 5.1: Trained policy and value function for moving a simulated end-effector to randomly set goals. Vertical and horizontal axes are end-effector positions. Red dot is goal position. Left figure shows the learned policy μ, right side shows the learned value function for different goal poses.

Chapter 6

Reaching II: Real-robot reaching

After the first experiment, we know that a simulated robot can move its end-effector to goal positions arbitrarily set within the workspace. Now, to investigate whether the same method would generalize to real robotic arms, robots were set up to perform a reaching task, where a central server maintained a replay buffer, parameters, and executed training. Data collection was distributed over three robots with respective computers.

6.1 Method

6.1.1 Definition of the MDP

The same definition used in the previous experiment was used, with the exception of a smaller action space for the sake of smoother trajectories:

$$\mathcal{A} = \{\mathbf{a} \in \mathbb{R}^2 \mid ||\mathbf{a}|| < 0.01\} \tag{6.1}$$

Also, the dynamics of the system can of course no longer be defined since it is a real system.

6.1.2 Environment

Instead of a simulated environment, a real robotic system was used. Three 3-DoF robotic arms controlled by cartesian commands were controlled by policies running on dedicated computers. The state was observed by reading servo angles and calculating cartesian coordinates using forward kinematics. The environment is still 2-dimensional by keeping the z-coordinate fixed.

6.1.3 Algorithms

The same NAF implementation as in the previous simulated experiment was used, but with the modifications for distributed data collection. The action output μ of the network was scaled to have maximum norm 0.01 to enforce smoother movements. The policy was trained on a separate server equipped with a GPU. For every arm, there was one dedicated computer each that evaluated the latest policy given the arm pose and sent the transitions to the server. The entire setup (excluding server) is shown in the bottom part of figure 4.2. In order to facilitate communication of recorded state transitions and fetching updated parameters, a server was implemented enabling PUT and GET requests from the local workers.

Action steps could only be done at approx. $2 - 4$ Hz, not including arm re-positioning for environment reset, making re-runs from scratch time consuming. Therefore, data gathering was first run on the three robots for 4 hours without policy updates, resulting in approximately $80k$ state transitions. The actions during the data collection-only phase were randomly drawn from $\mathcal{N}(\mathbf{0}, 0.005^2\mathbf{I})$, and when the robot arm reached outside the workspace, or reached within 1 cm of the target, the end-effector was replaced at a new random starting position. The commands were 2-dimensional vectors representing the relative movement in x and y direction respectively. When training of the parameters was started on the server, the robots kept collecting and pushing data to the server, and synchronized parameters before each reset of the end-effector position. During this phase, noise was added to the policy during every 3 out of 4 runs, while every 1 out 4 runs the policy was evaluated without noise in order to track progress.

6.2 Results & Discussion

The performance of the policy was evaluated by measuring the average distance to the target position on the last state before every reset. The progress of this metric is shown in figure 6.1, here shown with a running mean of width 128. The trained policy and value function are shown in figure 6.2. The distributed version of NAF solves the task of approaching arbitrarily set goals on a set of distributed real-world robots.

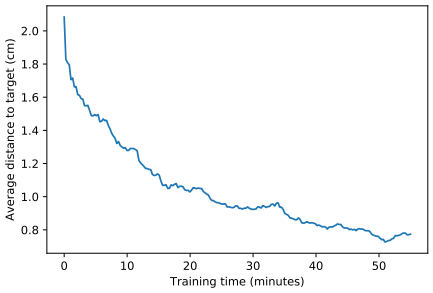

Figure 6.1: Average final distance of end-effector to randomly set target poses learned using a pool of robots collecting experience. NAF was used for training the policy on a separate server from a growing set of transitions sent from the collecting robots.

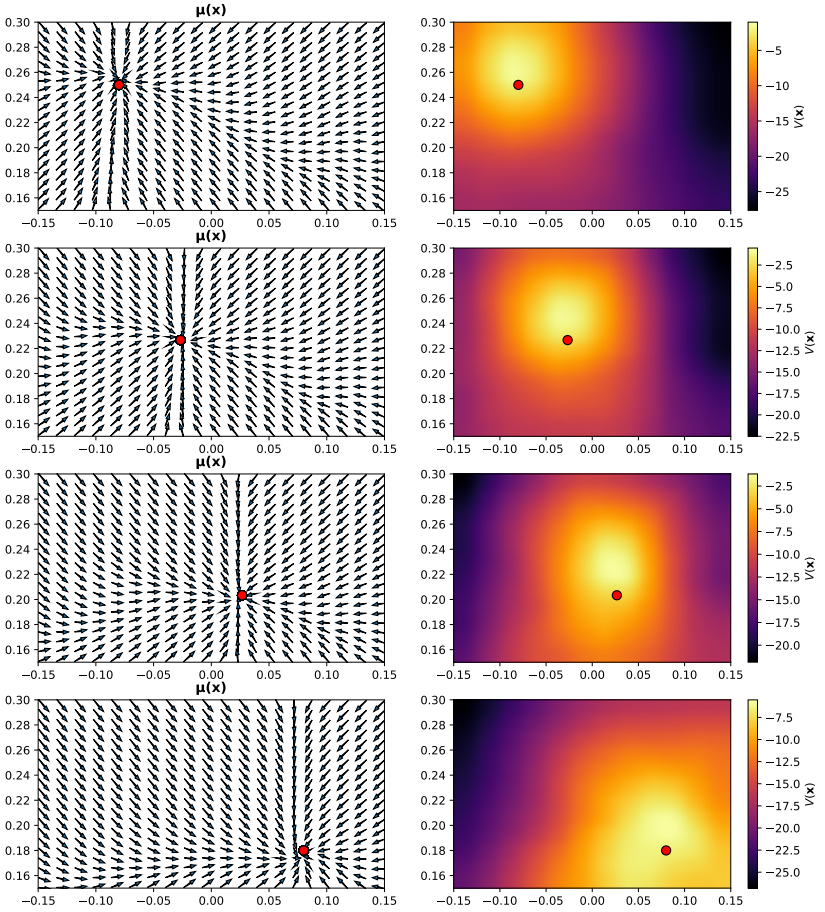

Figure 6.2: Learned policy using NAF with distributed collection of experience from real-world robots.

Chapter 7

Pushing I: Making NAF sweat

Previous experiments showed that using NAF, a policy can be learned both in simulation and on real robots to perform a reaching task to arbitrarily set goal positions within the workspace. Extending this, we want to investigate whether an object can be pushed to these goal positions. Preliminary experiments on the real robotic systems on a cube pushing task using NAF did however not converge to an intuitively good policy. To further investigate the capabilities of the NAF algorithm under more controlled circumstances, simulation experiments were first done.

7.1 Method

7.1.1 Definition of the MDP

The set of states

$$S \subset \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \tag{7.1}$$

was extended by including the center position of a pushable, circular, object. The set of actions was defined in the same way as previous experiments. The state transitions are now not as straightforward to describe as in the simulated reaching task since they depend on a physics model, albeit a simple one, for moving the object after contact with the simulated end-effector. The object was modeled to have no velocity, and since this is the case, a state only including positions still satisfies the Markov property.

The reward was defined from the state of the end-effector $\mathbf{e}$ and the object $\mathbf{c}$, both in the successor state. The goal $\mathbf{g}$ was constant before and

after actions:

$$r(\mathbf{e}, \mathbf{c}, \mathbf{g}) = \exp(-k_1||\mathbf{e} - \mathbf{c}||) + \exp(-k_2||\mathbf{c} - \mathbf{g}||) \tag{7.2}$$

During the experiments, several values were tried for k_1 and k_2.

7.1.2 Environment

The environment consisted of the circular object to push into some goal position, and a simulated, point-like end-effector. To simplify the problem, locations of the pushable object and goal were sampled approximately at the same position at every reset. This was in order to ignore rotations and translations of the problem and only examine whether the algorithm could learn this simpler scenario. The simulated end-effector was sampled uniformly in the entire workspace. There was at this stage a concern that when the end-effector has to go 180 degrees around the object, a clear multi-modal solution exists and might be problematic due to the quadratic shape of the advantage function. To further make this point clear, moving the end-effector clockwise around the pushable object, or counter-clockwise, are equally good. But staying at the same position, or even worse moving towards the object pushing the object further from the goal, lowers the return. Clearly, this has a multi-modal nature, with one mode being going around the object clockwise, and the other going counter-clockwise.

7.1.3 Algorithms

The NAF algorithm was used where the network consisted of 2 hidden layers of 200 ReLU activated units each. The activation function for the μ output was a tanh-function scaled by 0.01, the environment was also changed to cap actions with norm larger than this distance. A prioritized experience replay buffer was used to store and sample mini batches of size 64. Samples were drawn with probability proportional to $p_i = |\delta_i| + \epsilon$, where δ_i is the latest temporal difference error of sample i and $\epsilon = 10^{-9}$. The loss for each sample i was scaled by using importance sampling weights $\left(\frac{1}{p_i}\right)^{\beta}$ where β was linearly annealed from 0 to 1. A decay factor $\gamma = 0.99$ was used.

7.2 Results & Discussion

The algorithm was able to approach the pushable object, but unable to push it to the target position including going around the object to push it towards the target. This is illustrated in figure 7.1. In figure 7.2, the Q-function is shown for a scenario where intuitively the optimal policy should be to go up or down, but not left or right. It is unclear if this is due to the quadratic shape of the advantage function being unable to represent this, or some other detail in the implementation. By an intuitive policy, what is meant is to never push the object until the end-effector is positioned on the correct side of the object, but naturally another possible policy could be to push the object in a circle towards the goal. This did however not happen when evaluating the policy by running test runs in the environment, as can be seen in figure 7.1.

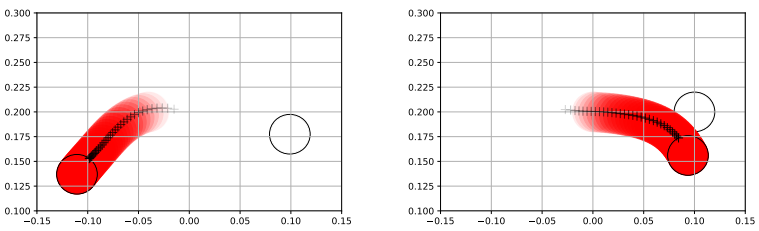

Figure 7.1: Results of NAF in simulation on pushing task. The red circle is the object being pushed, the white circle is the goal, and the cross is the simulated end-effector.

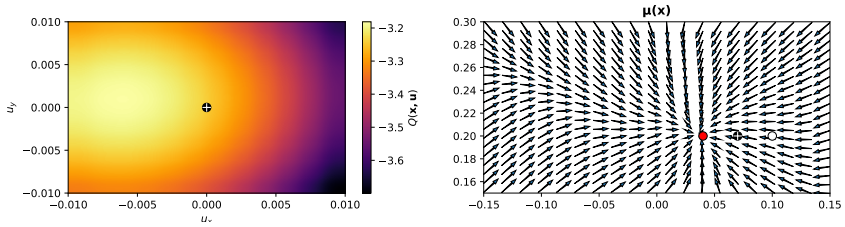

Figure 7.2: Results from using NAF on a simulated pushing task. Q-function and policy of the state given by the pushable object (red), the goal (white circle), and the end-effector (white cross). Since the advantage function is parameterized by a quadratic expression, which is uni-modal, the Q-function cannot represent the bi-modal nature of this scenario.

Chapter 8

Pushing II: DDPG on a modified problem

Previous experiments showed that reaching different goals within the workspace can be learned both in simulation and on real robotic systems. Increasing the complexity of the task by introducing an object to push to these goal positions however could not be solved. One hypothesis is that the task has a clear multi-modal nature which does not suit the quadratic shape of the advantage function in the NAF formulation well. To not enforce any constraints on the shape of the Q-function, experiments were done with Deep Deterministic Policy Gradient (DDPG) where the Q-function can be approximated by any differentiable function without any theoretical constraints (see section 2.3.5). Also, simplifications regarding translations, and rotations, of the problem were taken advantage of.

8.1 Method

8.1.1 Definition of the MDP

The MDP definition is the same as the previous simulated pushing task, except for the reward function. The reason for the change of the reward function was to include a positive reward for a positive change in the state. The reward was as before defined from the state of the end-effector $\mathbf{e}$ and the object $\mathbf{c}$, with the addition of including both the former state and the successor state denoted by $'$. The goal $\mathbf{g}$, once again, was constant

k_1	1000.0
k_2	1.0
k_3	1000.0
k_4	0.1
k_5	200.0

Table 8.1: Constants used in the reward function for pushing in simulation.

before and after actions.

$$\begin{aligned} r(\mathbf{e}, \mathbf{c}, \mathbf{g}, \mathbf{e}, \mathbf{c}') = & k_1 \left(||\mathbf{c} - \mathbf{g}|| - ||\mathbf{c}' - \mathbf{g}|| \right) + \\ & k_2 \exp\{-k_3 ||\mathbf{c}' - \mathbf{g}||\} + \\ & k_4 \exp\{-k_5 ||\mathbf{e}' - \mathbf{c}'||\} \end{aligned}$$

Constants are listed in table 8.1.

8.1.2 Environment

The environment was in the basic aspects the same as for the experiments in section 7. However, to simplify the task for the learning agent, the state representation and action space was changed to become rotation and translation invariant. The environment was translated so that the pushable object was positioned at the origin, and rotated so that the goal position lies on the negative x-axis, see figure 8.1. Actions in the environment were interpreted in this new coordinate frame.

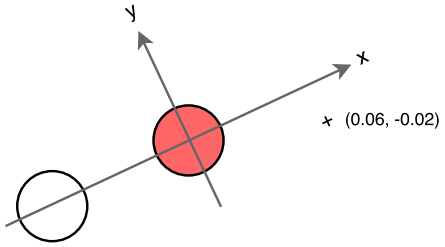

Figure 8.1: The environment is translated and rotated so that the agent sees the same problem, invariant of translations and rotations. The state of the environment reduces to three dimensions, the distance from the pushable object (red) to the goal position (white), and the coordinates of the end-effector/pointer (+) in the new coordinate frame.

8.1.3 Algorithms

DDPG was used, where two separate networks represent an actor and critic respectively. Both networks had two hidden layers with 400 units in the first layer, and 300 in the second layer. Batch normalization was used on the input, and between the two hidden layers. Activation functions were exponential linear units (ELU) except for the Q-value and policy outputs which had linear and tanh activation functions. The critic network was regularized using the L2-norm multiplied by 10^{-2}. Gradient descent was done on both networks using Adam with learning rate 10^{-3} for the critic and 10^{-4} for the actor and the other parameters set to the same as suggested by Kingma et al [25]. Parameters of target networks (θ') were updated using soft updates $\tau = 10^{-3}$:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{8.1}$$

A priority buffer was used in the same manner as stated in section 7. A single state transition was sampled from the environment using the current policy with noise, followed by a single gradient update for each of the networks from a sampled mini-batch and thereafter applying soft update to the target networks. Actions were sampled with added noise, using an ϵ linearly annealed from 1 to 0.1, according to:

$$\mathbf{u} = (1-\epsilon)\mu(s) + \epsilon\,[U(-1,1), U(-1,1)] \tag{8.2}$$

Action outputs from the networks were in the range $[-1, 1]$ and were scaled by 0.01 before being applied in the environment. Policies were evaluated by estimating the return from sampled start states by regularly executing a set of trials in the environment.

NAF was evaluated alongside DDPG with a separate replay buffer and environment and with the same procedure as for DDPG. Several runs with NAF were done with different sizes of hidden layers, using batch normalization, and changing to ELU activations. The NAF experiments did however not succeed to show any signs of converging at a good policy, and further results with NAF are omitted here.

8.2 Results & Discussion

Good policies were found after approximately 30′000 iterations, shown in figure 8.2 and 8.3. However, as can be seen in figure 8.4, estimated

returns varied greatly between runs despite weight initialization from the same distributions. Also, it was common for good policies to be found, but subsequent policies showed deteriorating estimated returns which did not recover.

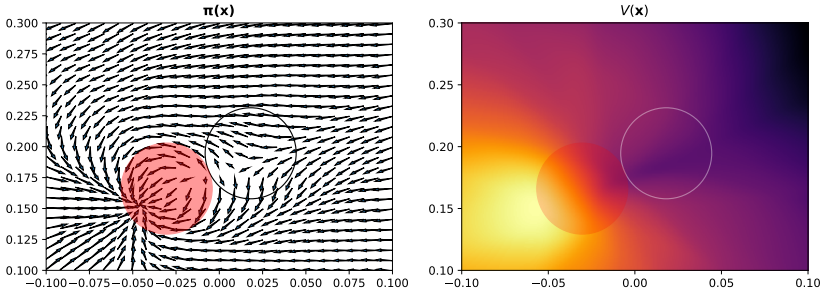

Figure 8.2: Trained policy and value function on a simulated pushing task using DDPG. The red circle is the pushable object, and the hollow circle is the goal. The value function was derived using $Q(\mathbf{x}, \pi(\mathbf{x}))$

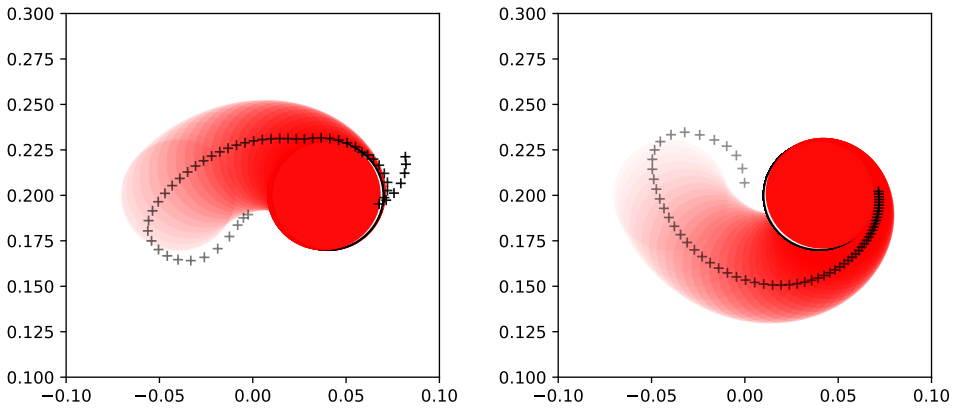

Figure 8.3: The policy trained using DDPG manages to go around the object to push it towards the goal position.

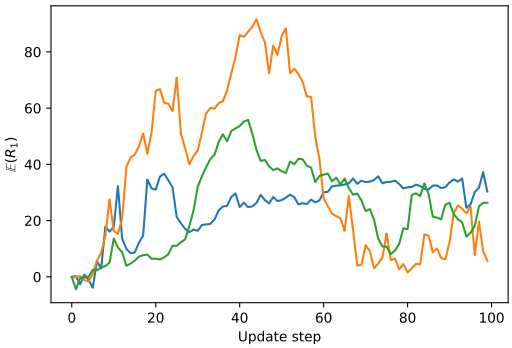

Figure 8.4: The estimated returns of policies trained using DDPG showed large variations between runs even though network initializations were sampled from the same distributions. Also, a good policy could be found but then showed a ”crashing” behavior for which the estimated returns did not seem to recover from.

Chapter 9

Pushing III: Real-robot application

At this point, we know that a pushing task can be learned, at least with some simplifications and using the Deep Deterministic Policy Gradient (DDPG) approach. Preliminary experiments were done in simulation, where noise was added equivalent to the noise expected from the real robotic system, but failed to converge to any decent policy. Also, all the previous experiments in simulation were harder than anticipated and consumed several weeks before finally solving the problem. Instead of further investigating training on the real robotic systems from this point on, the policy trained in simulation is briefly evaluated on the real robotic system before continuing with investigating pose estimation methods.

9.1 Method

To estimate the pose of the cube, a LIDAR was used by placing it in front of the robot arm (figure 9.1). The cube to push was a red wooden cube with all sides measuring 4 cm. One of these sides can be found from the LIDAR-measurements by plotting the points onto a matrix/image and using the Hough transform [47], see section 4.5. For the following experiments, only the position of the cube was kept, ignoring the rotation. Conversion of the found position of the cube to robot-frame was done using a least-squares approach after randomly placing the cube at several positions known from the forward-kinematics of the arm.

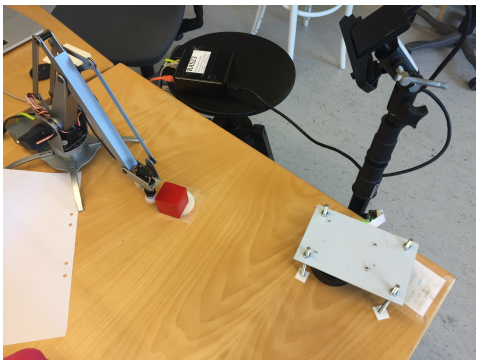

Figure 9.1: LIDAR seen on the right side in the picture measuring positions of the cube. The camera is not used in this experiment.

9.2 Results & Discussion

The policy trained in simulation successfully pushes the cube towards a goal set in the center of the workspace, video of this can be seen here: `https://youtu.be/82XNDBPbJH0`. Sometimes the policy fails to push the cube further due to small action commands not affecting the robot, and due to differences in the shapes of the simulation object (circle) and the cube. Adding a small noise term alleviated these problems and was used in the linked video.

Chapter 10

Pose estimation I: Using a fixed camera

The policy trained in simulation for pushing a cube to some goal position was shown to work on the real robotic system with only some minor tweaks. To know where the cube was positioned, a LIDAR was used. Using the LIDAR for pose estimation of the cube however required that no other objects were accidentally measured, and used the fact that the position of the cube can be described by a bounded line in the plane. A more general approach is to infer the pose from a camera. A usual setup in robotics is to have a camera at a fixed position and infer poses under the assumption that the camera does not shift position. Experiments are here first described for estimation under this assumption and then extended for estimation from a non-fixed camera.

10.1 Method

A camera was mounted at a fixed position as shown in figure 10.1. A LIDAR was used to label the pose of the cube in the coordinate frame of the robot. In total, 10'000 RGB images were collected during execution of the trained policy. For each picture, depth channel data was also collected to evaluate the difference in performance when adding this information. For the depth channel, the camera returns numbers in the range $[0, 255]$ aligned for each pixel in the color image. The distance 0 is in practice never registered but used to represent that depth for this pixel is missing.

The first part of the network in figure 3.1 was used, with two hidden layers of 100 hidden ELU activated units each after the 2-D expected

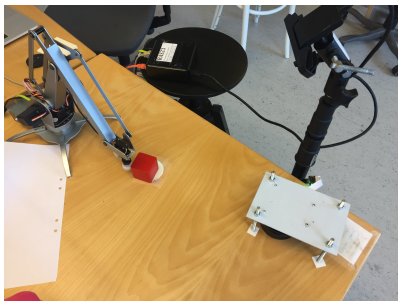

Figure 10.1: Fixed camera placement for pose estimation of the cube.

position. The output was a 2-dimensional linear layer predicting the cube position in the robot coordinate frame. For incorporating depth into the network, I propose a layout, shown in figure 10.2, where depth is input at two locations in the network. The first place is simply as a fourth dimension into the first convolutional layer. The second place is to append the expected distance $\mathbb{E}[D_k]$ for each feature $k \in [1, 32]$ to the fully connected layers, using the spatial softmax as the probability when calculating the expectation:

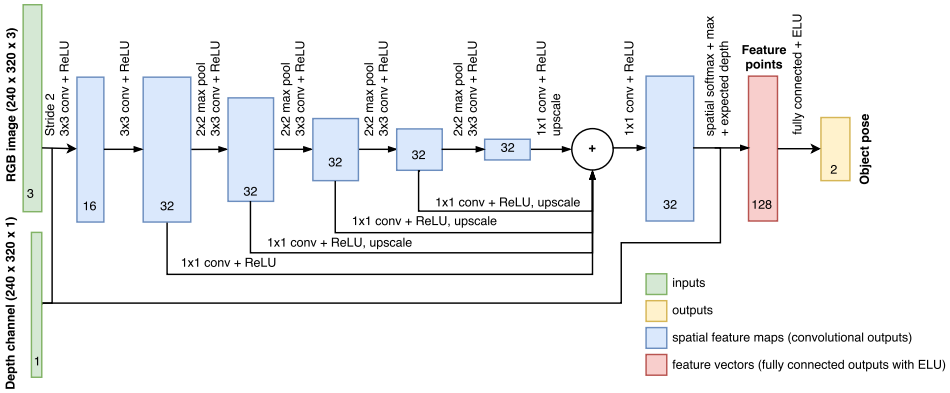

Figure 10.2: Proposed pose estimation network incorporating depth information. Input to the first convolutional layer is a 4-channel RGBD image. Using the spatial softmax as a per-feature probability distribution, expected depths per feature map are also concatenated with the expectation over image coordinates. As a measure of uncertainty in each feature, the maximum activation for each feature map, after the spatial softmax, are also concatenated with the input to the fully connected layer.

$$\mathbb{E}[D_k] = \sum_{i,j} p(c_{i,j,k}) d_{i,j} \tag{10.1}$$

Here, $p(c_{i,j,k})$ is the output of the spatial softmax at pixel (i, j) for feature map k and $d_{i,j}$ the registered depth at pixel (i, j). The network was trained by minimizing the mean square error and using the Adam optimizer. The training and validation losses are plotted in figure 10.3 showing that only using RGB converges faster and that adding depth did not contribute to better scores in this experiment.

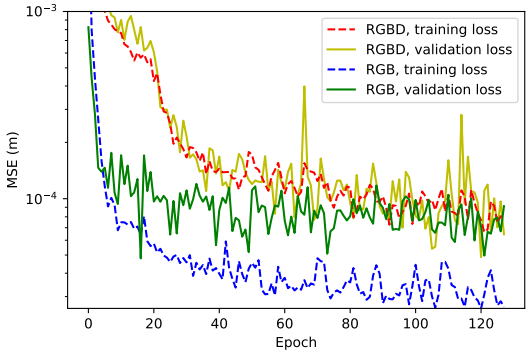

Figure 10.3: Training curves for pose estimation from a fixed position camera. Adding depth information did not lower training scores, only slowing down the time needed to converge.

10.2 Results & Discussion

The trained network using RGB was used instead of the pose estimates from the LIDAR and tested on a robot. The network needed significantly more time to evaluate than when using the LIDAR estimates, the algorithm now running at 1-2 Hz compared to approximately at 10Hz using LIDAR estimates. The bottleneck when using LIDAR estimates are not the actual LIDAR, but rather controlling the robot and measuring servo angles etc. A video was recorded of the result and can be seen here: `https://youtu.be/vakM-xvhEmE`. A dark gray cup was added into the workspace without previous tests during recording of the video showing that the estimation could handle some distraction. Further tests showed that any red object will disturb the pose estimates, making the policy

unsuccessful in pushing the cube. This was expected though since no distractors were added during data collection.

Chapter 11

Pose estimation II: Relative pose from synthetic data

We now know that a neural network can extract 2D image coordinates of some features and use these to infer poses in some other coordinate frame. This is however under the assumption that the camera position is constant with respect to the target coordinate frame. We have also seen that this estimation is reliable enough to perform the pushing task using the resulting cube pose estimates. What if we could extract 2D image locations of features but the camera position is unknown and changing? If some features in the image defines a coordinate frame that are accessible to us, then an algorithm should theoretically be able to infer these poses in this coordinate frame as long as those features are visible. For the following experiment, we assume that we know the projected 2D locations of some features in 3D space, where the 2D plane is randomly placed.

11.1 Method

Since the region around the end-effector in these experiments is non-symmetric (see left part of figure 11.1), this area could be used to define a coordinate frame as long as this part is visible to the camera. Let this coordinate frame be defined as in the right side of figure 11.1.

Inspired by the three marked points in figure 11.1, an environment was setup where these points together with different camera positions and orientations could be sampled and projected onto a 2D plane (the image). Some samples are shown in figure 11.2. The target values were given in the frame of the end effector; a 2D horizontal coordinate frame

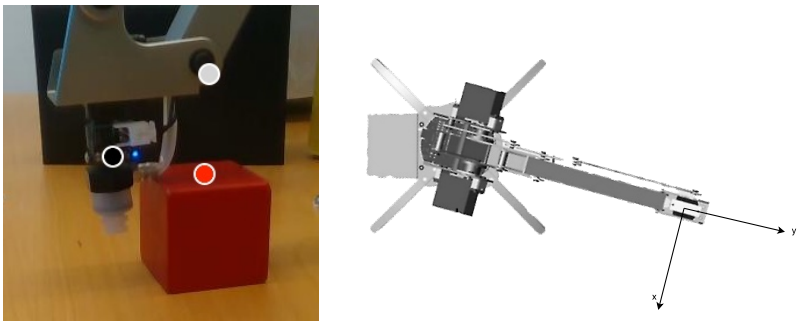

Figure 11.1: If the section of the robot to the left is visible, it can define a coordinate frame for the cube invariant to camera rotation and translation.

with the origin at the suction cup and x-, and y-axes parallel to the table.

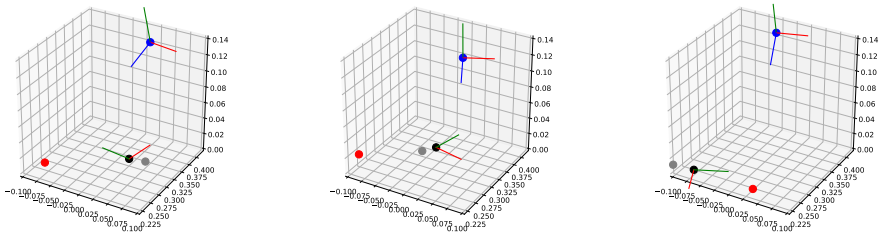

Figure 11.2: Sampled camera (blue), cube (red), and robot poses (black and gray). Target values were cube poses in the frame given by the robot, here seen with origin in the black point with x-, and y-axis shown as the red and green lines. Input data were the three points projected onto a 2D plane given by the camera and a focal length f. Comparisons were made when appending a third dimension to the picture, which was the length from camera to each point.

All the operations needed to infer the target values from known 3D coordinates were linear transformations, giving rise to question if it is sufficient to regress a linear model. Experiments were therefore done with a linear model, and a one and two hidden layer model with 100 hidden ELU-activated units and batch normalization, and including or excluding depth information. The loss was mean squared error and the networks were trained using the Adam optimizer.

11.2 Results & Discussion

As can be seen in figure 11.3 and 11.4, deeper networks achieve better results. 2D coordinates were also shown to be sufficient, although depth information here improves the results. Interestingly, the predictions for a 2 hidden layer with depth information are still quite noisy, even though the 2D feature points are readily available and given without noise. This raises concerns to whether good enough results can be achieved with real data where feature coordinates are inferred by a convolutional neural network, and also where labels are produced with noise.

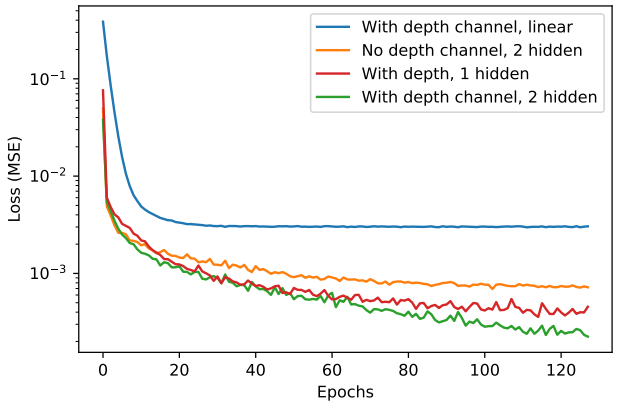

Figure 11.3: Training losses for pose estimation of simulated robot and cube. Losses for neural networks given different parametrizations and input data.

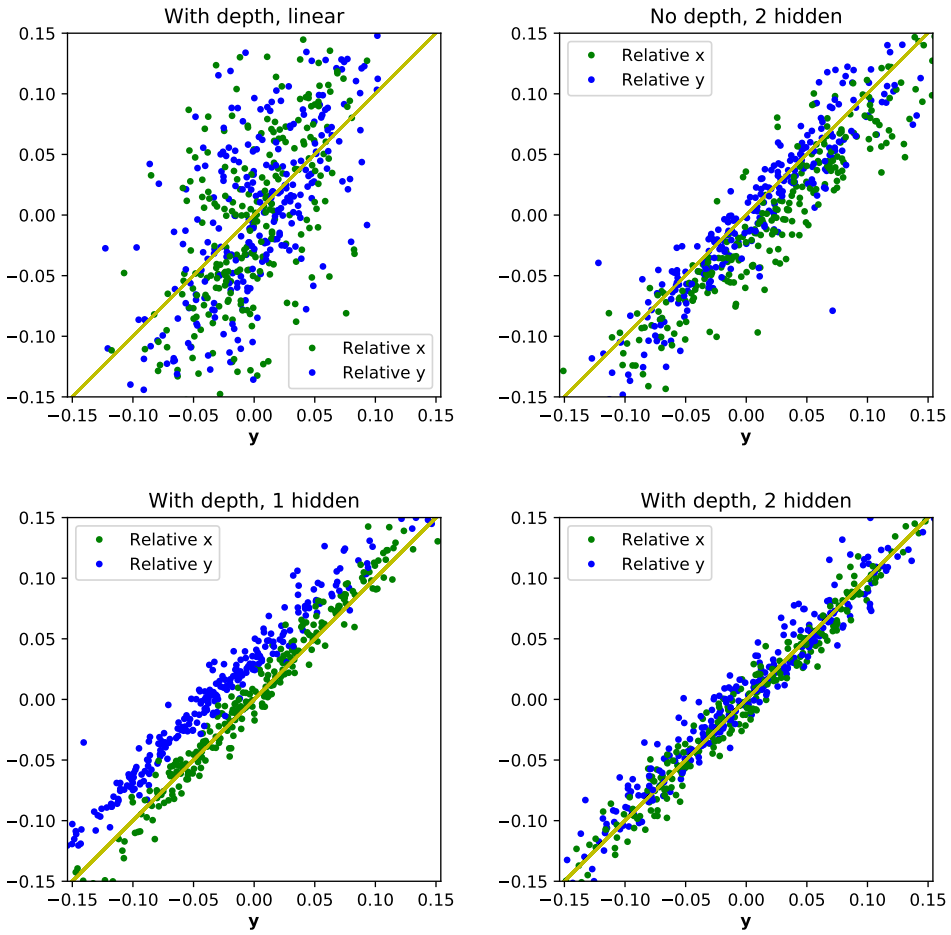

Figure 11.4: Pose estimation from a simulated environment and camera. Target x (green) and target y (blue) values (horizontal axis) plotted against the predicted x and y values (vertical axis). Plot 1: Depth information was provided to the two points on the robot and the cube. Network was only a linear layer. Plot 2: Only 2D coordinates were given as inputs, network had 2 hidden layers. Plot 3: Depth was given, network using 1 hidden layer. Plot 4: Depth was given, network using 2 hidden layers.

Chapter 12

Pose estimation III: Relative pose from real data

Now knowing that a neural network given known 2D coordinates of features can find the 3D relative pose, experiments were done using RGB images and ground truth labels collected on a real robotic system.

12.1 Method

RGB images were collected while executing the pushing policy on one of the robots. The images were labeled with relative poses to the end-effector using the position estimates of the cube using the LiDAR, and the arm using the forward kinematics on the servo angles. The camera was placed at random positions around the workspace, always capturing at least the portion of the arm as shown in figure 11.1, and capturing at least the top surface of the cube. Distracting objects were put into the scene such as different kinds of rubble in the background, changing the appearance of the table surface, changing light conditions, and playing up videos in the background. Some of the training images are shown in figure 12.1. Images were augmented during training by adding Gaussian noise, randomly scaling and translating the image, and randomly shifting the color channels. The network in figure 10.2 was used with some modifications. As previous results (section 11.1) showed better performance using 2 fully connected hidden layers with 100 units each, this was used as last layers, together with batch normalization. Depth images were also collected and trained on, but showed to be overfitting before good models were found. Adding regularization in the form of dropout and

weight decay did help against overfitting when using depth, but instead resulted in worse validation scores.

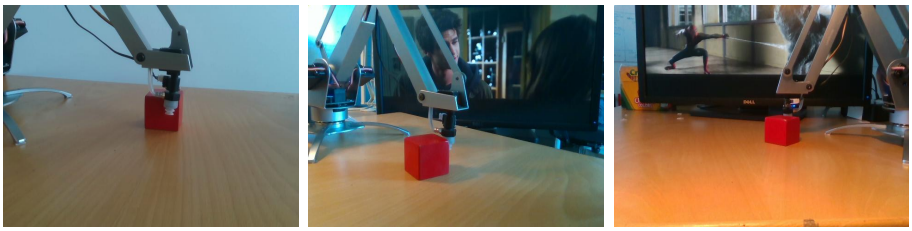

Figure 12.1: Data gathering for pose estimation in coordinate frame defined by the position and orientation of the end-effector. Camera was randomly placed at different positions, angles, and heights. To make predictions robust, images were recorded using different lighting conditions, different table surfaces, and different backgrounds.

12.2 Results & Discussion

Plotting the labels against the predicted values show that the network can learn the relative poses, albeit the precision being roughly ± 1 cm was not good enough to get robust results on the robots. Plots of labels plotted against predicted values are shown in figure 12.2. Roughly estimating the influence of the resolution gives the following: Assume the cube is at such a distance that 8 cubes fit next to each other while all of them are still fitting inside the image borders. The input image has a width of 340 pixels, dividing the total width of 8 cubes by the 340 pixels gives:

$$4 \cdot 8 \text{ cm} \;/\; 340 \text{ pixels} \;\approx 0.09 \text{ cm} \;/\; \text{pixel} \;\approx 1 \text{ mm} \;/\; \text{pixel} \tag{12.1}$$

The intermediary representation has half the resolution implying approximately 2 mm / pixel which can only to a small extent explain the noisy predictions. On top of this, pose estimates were shown to vary approximately ± 5 mm from scan to scan, partially explaining the appearance of the plots in figure 12.2. The fact remains though that when running estimation on one of the real robots, it failed to give stable results.

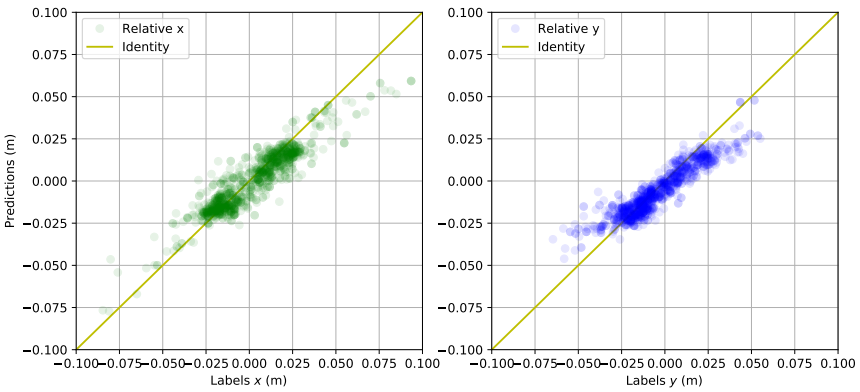

Figure 12.2: Results for estimating the cube pose in the end-effector frame using RGB-images from a camera at random positions, heights, and orientations. Target values (horizontal axis) are plotted against predicted values (vertical axis). The model predictions were too inaccurate to produce good results on the robots for the pushing task.

Chapter 13

Conclusions and future work

In this thesis, NAF was evaluated in both a simulated setting and distributed over several robots. Initial experiments with simple reaching tasks worked fine, but extending this to a pushing task failed, even after simplifications like keeping starting and goal positions fixed. This could either be due to some error in the implementation that did not affect the simpler reaching task, or that the restriction to a quadratic advantage function could not represent the bi-modal nature of the task. DDPG however could solve this task but did not prove stable under noisy controls and observations. Reformulation of the problem as a partially observable Markov decision process might be an interesting way forward since highly accurate sensors are not always what are seen in reality.

Regarding pose estimation, a network was proposed to incorporate depth information, a common feature nowadays even in consumer products. Simulation showed better results when adding this feature, although for real data no benefit was seen. Future experiments could include replacing zero representing missing depth values with something else, for example the maximum distance registered. Relative pose estimation was clearly shown to be capable of learning the geometric interpretation from 2D data, although the precision was too low for this use case. This could have been due to the neural network not being able to represent this accurately enough, the resolution of input data or intermediary representations being too low, or labels during training being too noisy requiring more data for higher quality predictions. The former case is somewhat supported by the case that simulated data giving 2D coordinates also produced noisy estimates.

Bibliography

[1] 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.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[3] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

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

[5] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[6] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[7] Ali Yahya, Adrian Li, Mrinal Kalakrishnan, Yevgen Chebotar, and Sergey Levine. Collective robot reinforcement learning with distributed asynchronous guided policy search. *arXiv preprint arXiv:1610.00673*, 2016.

[8] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. *arXiv preprint arXiv:1610.00633*, 2016.

[9] Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. *arXiv preprint arXiv:1610.00696*, 2016.

[10] Yevgen Chebotar, Mrinal Kalakrishnan, Ali Yahya, Adrian Li, Stefan Schaal, and Sergey Levine. Path integral guided policy search. *arXiv preprint arXiv:1610.00529*, 2016.

[11] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

[12] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

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

[14] Sungheon Park, Jihye Hwang, and Nojun Kwak. 3d human pose estimation using convolutional neural networks with 2d pose information. In *Computer Vision–ECCV 2016 Workshops*, pages 156–169. Springer, 2016.

[15] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *arXiv preprint arXiv:1603.00748*, 2016.

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

[17] M Sniedovich. A new look at bellman's principle of optimality. *Journal of Optimization Theory and Applications*, 49(1):161–176, 1986.

[18] Martin L Puterman and Shelby L Brumelle. On the convergence of policy iteration in stationary dynamic programming. *Mathematics of Operations Research*, 4(1):60–69, 1979.

[19] DRGHR Williams and GE Hinton. Learning representations by back-propagating errors. *Nature*, 323(6088):533–538, 1986.

[20] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[21] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[22] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[23] Fu Jie Huang, Y-Lan Boureau, Yann LeCun, et al. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

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

[25] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.

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

[28] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[29] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[30] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[31] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[32] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[33] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962. Citeseer, 2003.

[34] Ali Ghadirzadeh, Atsuto Maki, Danica Kragic, and Mårten Björkman. Deep predictive policy training using reinforcement learning. *arXiv preprint arXiv:1703.00727*, 2017.

[35] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[36] Martin Riedmiller. Concepts and facilities of a neural reinforcement learning control architecture for technical process control. *Neural computing & applications*, 8(4):323–338, 1999.

[37] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[38] DeepMind. Explore the alphago games. `https://deepmind.com/research/alphago/`, 2017.

[39] Guy Lever. Deterministic policy gradient algorithms. 2014.

[40] Sergey Levine and Vladlen Koltun. Guided policy search. In *ICML (3)*, pages 1–9, 2013.

[41] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012.

[42] 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.

[43] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[44] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 512–519. IEEE, 2016.

[45] Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation. In *European Conference on Computer Vision*, pages 483–499. Springer, 2016.

[46] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807, 2014.

[47] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[48] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[49] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[50] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2017-04-01].