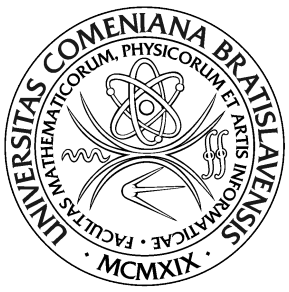
COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS



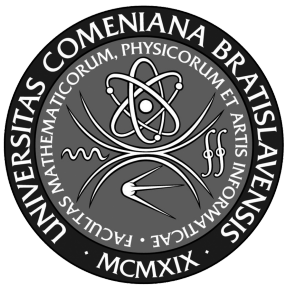
LEARNING OF OBJECT GRASPING IN A ROBOTIC SYSTEM

Master's thesis

Bratislava, 2017 Bc. Peter Kovács

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS



LEARNING OF OBJECT GRASPING IN A ROBOTIC SYSTEM

Master's thesis

Study programme:	Applied Computer Science
Field of study:	2511 Applied Informatics
Department:	Department of Applied Informatics
Supervisor:	prof. Ing. Igor Farkaš, Dr.

Bratislava, 2017 Bc. Peter Kovács

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Peter Kovács
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Applied Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Learning of object grasping in a robotic system

Aim:
1. Study the literature on reinforcement learning (RL) approaches in robotics for object reaching and grasping.
2. Choose and test a recently proposed RL algorithm and try to improve its functioning.
3. Apply and evaluate the chosen algorithm in the reaching and grasping task using a simulated robotic arm.

Literature:
Amari, S.-I. (1998). Natural gradient works efficiently in learning. Neural Computation, 10(2), 251–276.
Kakade, S. (2001). A natural policy gradient. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, Advances in Neural Information Processing Systems 14 (NIPS), pages 1531–1538. MIT Press.
Schulman, J., Moritz, P., et al. (2015). Trust region policy optimization. In International Conference on Machine Learning (ICML).

Annotation: Reinforcement learning is an area of machine learning dealing with optimization of agent's behavior, that acts in an environment, being driven by the obtained reward. RL is applicable also in tasks with many degrees of freedom (DoF), where analytical solutions, e.g. in tasks of object reaching and grasping, are not applicable. Nevertheless, finding efficient RL algorithms is still a challenge, especially in tasks with a larger number of DoF.

Supervisor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	13.12.2015
Approved:	14.12.2015

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Peter Kovács
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Learning of object grasping in a robotic system
Učenie robotického systému uchopovať objekty

Cieľ: 1. Preštudujte literatúru o učení posilňovaním v robotike so zameraním na siahanie a uchopovanie objektov.
2. Vyberte a otestujte nedávno navrhnutý RL algoritmus a pokúste sa ho vylepšiť.
3. Aplikujte a vyhodnoťte zvolený algorithm v úlohe siahania a uchopovania objektov s využitím simulátora robotického ramena.

Literatúra: Amari, S.-I. (1998). Natural gradient works efficiently in learning. Neural Computation, 10(2), 251–276.
Kakade, S. (2001). A natural policy gradient. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, Advances in Neural Information Processing Systems 14 (NIPS), pages 1531–1538. MIT Press.
Schulman, J., Moritz, P., et al. (2015). Trust region policy optimization. In International Conference on Machine Learning (ICML).

Anotácia: Učenie posilňovaním (RL) je oblasť strojového učenia zaoberajúca sa optimalizáciou správania agenta, ktorý koná v prostredí, súc vedený získavanou odmenou. RL je aplikovateľné aj v úlohách s väčším počtom stupňov voľnosti, kde analytické prístupy, napr. v úlohách siahania a uchopovania objektov, nie sú použiteľné. Avšak efektívne RL algoritmy sú stále výzvou, obzvlášť v úlohách s väčším počtom stupňov voľnosti.

Vedúci: prof. Ing. Igor Farkaš, Dr.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 13.12.2015

Dátum schválenia: 14.12.2015 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

I hereby declare that this thesis is a presentation of my original research work and that I have not used any sources and aids other than those stated in the thesis.

. . . . . .

Bratislava, 2017 Bc. Peter Kovács

Acknowledgement

I would like to express gratitude to my supervisor prof. Ing. Igor Farkaš, Dr. for the useful comments, remarks and engagement through the learning process of this master thesis.

Abstract

In this thesis we apply trust region policy optimization (TRPO) algorithm to the field of robotic reaching and grasping in a simulated environment. We show that the algorithm is suitable for learning good policies for 3D reaching and grasping and also for 3D reaching with obstacle. Reaching and grasping model is represented as a feed-forward neural network which controls force applied to each degree of freedom. We show that TRPO is highly parallelizable and we introduce two improvements for its convergence properties. They are based on reusing information from previous trajectories and from previous gradient direction, respectively.

Keywords: reinforcement learning, reaching, grasping, parallelism, TRPO

Abstrakt

V práci sa zaoberáme aplikovaním algoritmu ”trust region policy optimization” (TRPO) na simulované robotické siahanie a uchopovanie v priestore. Ukazujeme, že pomocou algoritmu TRPO sa dokáže robotické rameno naučiť nielen siahať a uchopovať predmet v trojrozmernom priestore, ale siahať naň aj po pridaní prekážky. Model siahania a uchopovania je reprezentovaný doprednou neurónovou sietou, ktorá kontroluje silu aplikovanú na jednotlivé stupne voľnosti. V práci ukazujeme, že algoritmus TRPO je ľahko paralelizovatelný a navrhujeme dve zlepšenia jeho konvergencie založené na využití informácie z jeho predchádzajúcich trajektorií, resp. predchádzajúcich gradientov.

Kľúčové slová: učenie posilňovaním, siahanie, uchopovanie, paralelizmus, TRPO

Contents

Chapter 1

Introduction

Finding a suitable grasp among infinite set of candidates is a challenging task that has been addressed frequently in the robotics community resulting in different approaches such as in Kraft et al. (2010), Castellini et al. (2007), Kalakrishnan et al. (2011) and Zdechovan (2012). The methodologies can be divided into two categories analytic solutions using methods of inverse kinematics/dynamics and data driven approaches (Bohg et al., 2014). Until recently the field of robotic grasping was clearly dominated by analytic approaches. However, with development of fast physics simulators and new advances in the field of machine learning and neural networks research lead to new data driven approaches.

Data driven approaches can be divided into three categories. The first category covers methods that assume knowledge of 3D mesh and the challenge is then to sample a set of good grasp hypotheses and rank them according to some quality measure. The second approach would be learning from humans

when robot can learn successful grasps by observing them. This method is often referred to as demonstration or imitation learning. The last category is learning through trial and error. Instead of computing possible grasps we just try multiple of them and learn some generalized knowledge from such experience. Reinforcement learning is a good representative of this category.

Kalakrishnan et al. (2011) showed successful application of RL to door opening and pen grasping task. Zdechovan (2012) showed how to use CACLA (van Hasselt & Wiering, 2009) to learn successfully grasp object of random shapes with an actor–critic algorithm. Lillicrap et al. (2015) showed that its also possible to employ a deep learning with actor–critic methods to learn good models for various robotic tasks. Recently Schulman et al. (2015) proposed TRPO algorithm which showed promising results on robotic locomotion motivated us to test this algorithm on robotic reaching and grasping task.

In this work we will first go through the necessary theory to understand policy gradient algorithm which will include ideas such as function approximations and reinforcement learning terminology and definitions (Chapter 2). Then we follow with improvements of the policy gradient such as natural policy gradient and trust region policy optimization (TRPO) in Chapter 3. We briefly describe current trends in reinforcement learning applied to not only reaching and grasping but also to domains with visual input (Chapter 4). In Chapter 5 we will present different reaching and grasping environments we designed and solved, followed by experiments with ideas which are supposed to improve TRPO convergence properties. Finally, in Chapter 6 we will show that its possible to scale TRPO over multiple cores and also how to do it.

Chapter 2

Theory

In this chapter we will start by defining neural networks model, with focus only on feed forward architectures. Then we will describe how to train them and then we will dive into the theory and framework of reinforcement learning.

2.1 Artificial Neural Networks

Artificial neural networks are mathematical models inspired by the function of the human brain. There exists a wide variety of different architectures suitable for different tasks (Haykin, 2009). In this section we will only focus on feedforward neural networks, concretely multilayer perceptron, that we use in our work.

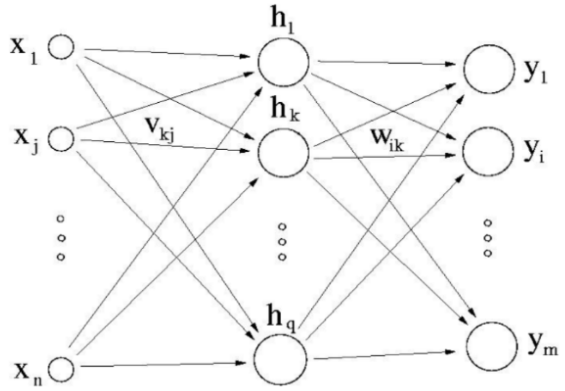


Figure 2.1: Multilayer perceptron

2.1.1 Multilayer perceptron

Multilayer perceptron (MLP) is a fully connected feedforward neural network consisting of several layers of neurons where each neuron in the current layer is connected with all neurons in lower and higher layers. Figure 2.1 illustrates a multilayer perceptron with input layer $\boldsymbol{x} = [x_1, x_2, \cdots, x_n]$, one hidden layer $\boldsymbol{h} = [h_1, h_2, \cdots, h_q]$ and one output layer $\boldsymbol{y} = [y_1, y_2, \cdots, y_n]$.

Each layer in MLP consists of several neurons. To compute the output of hidden layer we first need to have results of previous layers. Formula for computing output for MLP with input $\boldsymbol{x}$ one hidden layer $\boldsymbol{h}$ and one output layer $\boldsymbol{y}$ is:

$$\boldsymbol{y} = \boldsymbol{W}\overbrace{\varphi(\boldsymbol{x}\boldsymbol{V})}^{\boldsymbol{h}}$$

to write it in a summation

$$h_k = \sum_{i=1}^{n} v_{ki} x_i$$

$$y_i = \sum_{k=l}^{q} w_{ik}\varphi(h_k)$$

Selecting the number of neurons, layers and activation function φ depends on a problem. The general strategy for selecting the number of neurons and layers is that one starts with a small model and enlarge if the previous configuration was not successful, however take care that your model does not overfit.

2.1.2 Training neural networks

Finding parameter matrices $\boldsymbol{V}$ and $\boldsymbol{W}$, lets call them $\boldsymbol{\theta}$, for our MLP such that some loss function L is minimized is not easy at all due φ being a nonlinear function. In supervised learning the most common methods for optimizing neural network are some sort of gradient descent algorithm (GD). Due to increasing number of data, classical GD is rarely used. Instead the newer stochastic GD (SGD) is used. The difference between them is that GD use all of the data to compute gradient of the loss function L with respect to $\boldsymbol{\theta}$, while SGD selects only a subset of datapoints and estimate actual gradient.

To compute the gradient of any model with SGD we first need to know what loss function to choose. Loss function tells us how well we perform on our optimization problem. For regression (estimating real value based on observation) one possible loss function L can be the squared error

$$L = \sum_{i=0}^{n} (d_i - y_i)^2$$

where y_i is the predicted value and d_i is the desired(target) value. For classification task (estimating category based on observation) the most often used L is the cross-entropy loss.

$$L = -\sum_{i=0}^{n}\sum_{k=1}^{K} \boldsymbol{d}_k^{(i)} \log(\boldsymbol{y}_k^{(i)})$$

where $\boldsymbol{d}_k^{(i)}$ says what is the probability of sample i to be in the class k. Note that $\boldsymbol{y}$ and $\boldsymbol{d} \in \mathbb{R}^K$, where K is the number of classes and n is the number of samples in minibatch. Now that we have function to optimize SGD we just need to compute gradient with respect to $\boldsymbol{\theta}$ and update our parameters in the negative direction to minimize L

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} L$$

2.2 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in the environment so as to maximize cumulative reward. In operations research it is also known as approximate dynamic programming. At the first part of this section we will state the formal definition of RL, then we will take a look at how to solve RL problem.

2.2.1 Formal definition

Formal definition of reinforcement learning comes from Markov decision process (MDP) theory defined by the tuple $(S, A, P, r, \rho_0, \gamma)$.

- S is a set of states (e.g., in robotic arm grasping, S might be positions and angles of joints and joint torques),
- A is a set of actions (e.g. all possible changes in torque joints),
- $P : S \times A \times S \to \mathbb{R}$ is the transition probability distribution (model),
- $r : S \times A \times S \to \mathbb{R}$ is the reward function,
- $\rho_0 : S \to \mathbb{R}$ is the distribution of the initial state $\boldsymbol{s}_0$,
- $\gamma \in (0, 1]$ is the discount factor.

Let π denote a stochastic policy

$$\pi : S \times A \to [0, 1]$$

and $\eta(\pi)$ denote its expected discounted reward:

$$\eta(\pi) = \mathbb{E}_{\boldsymbol{s}_0, \boldsymbol{a}_0, \ldots} \left[\sum_{t=0}^{\infty} \gamma^t r(\boldsymbol{s}_t) \right]$$

where $\boldsymbol{s}_0 \sim \rho_0(\boldsymbol{s}_0)$, $\boldsymbol{a}_t \sim \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$, $\boldsymbol{s}_{t+1} \sim P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$. The goal of RL is to find a policy $\pi : S \to A$ mapping from states to actions that maximizes $\eta(\pi)$.

MDP dynamics is shown in Figure 2.2. The agent observes the current world state $\boldsymbol{s}_t$, and performs action $\boldsymbol{a}_t$, while receiving a reward r_t and transforming

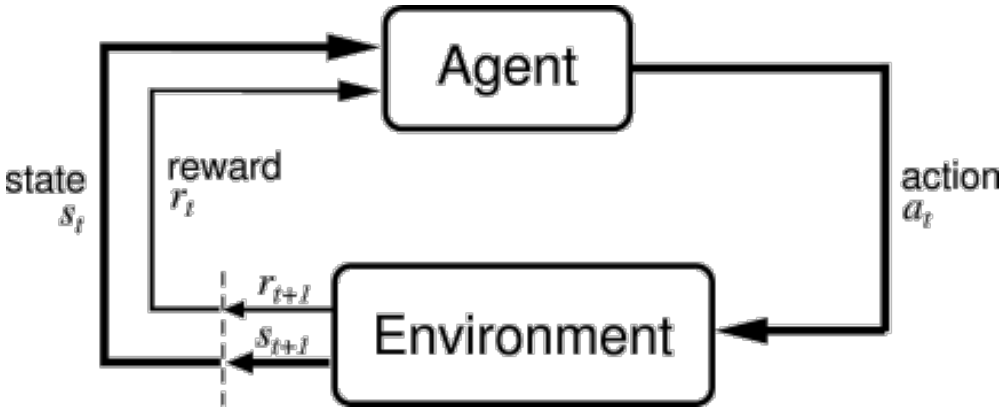


Figure 2.2: RL dynamics

the world into new state $\boldsymbol{s}_{t+1}$.

2.2.2 Bellman equation and value functions

In order to determine which action to take in which state we need some estimate of how good it is for the agent to be in a given state. The notion of "how good" here is defined in the terms of expected discounted reward function

$$V_\pi(\boldsymbol{s}_t) = \mathbb{E}_{\boldsymbol{a}_t, \boldsymbol{s}_{t+1}, \ldots} \left[\sum_{k=0}^{\infty} \gamma^l r(\boldsymbol{s}_{t+k}) \right]$$

according to which, the value of a state $\boldsymbol{s}_t$ under policy π is the expected return when starting in state $\boldsymbol{s}_t$ and following policy π thereafter. This function is called *state value function for policy* π.

Similarly we can define *state–action value function for policy* π or sometimes referred to as Q function.

$$Q_\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathbb{E}_{\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}, \ldots} \left[\sum_{k=0}^{\infty} \gamma^l r(\boldsymbol{s}_{t+k}) \right]$$

A fundamental property of value functions is their recursive relationship, that for any policy π and any state $\boldsymbol{s}$

$$V_\pi(\boldsymbol{s}) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{s}|\boldsymbol{a}) \sum_{\boldsymbol{s}'} P(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}) \left[r(\boldsymbol{s}) + \gamma V_\pi(\boldsymbol{s}')\right]$$

This relationship is called Bellman equation and expresses the relationship between the value of a state and the values of its successor states. The derivation of Bellman equation can be found in Sutton & Barto (1998).

2.2.3 Optimal value function

Solving MDP task means finding a policy which maximizes reward over time. Value functions can be used to define a partial ordering of policies such that a policy π is better or equal than a policy $\pi^{'}$ if its expected return is greater than or equal to $\pi^{'}$ for each state. Similarly, we can define ordering for Q functions. The policy π_* that is better or equal to than every other policy is called the optimal policy. Although there may be more than one such policy, they share the same state value function $V_*(\boldsymbol{s}) = \max_\pi V_\pi(\boldsymbol{s})$.

Even though we have defined what is an optimal value function and shown Bellman equation, it is rarely possible to compute its exact values. A critical aspect of the RL problem is its computational feasibility. In the tasks with small and finite state space it is possible to compute optimal value function using arrays or tables with one entry per state. However in many cases of practical interest, tasks are inherently continuous (with continuous states and sometimes also actions, e.g. robotic tasks). In these cases the function must be approximated by some compact parametrized function representation.

2.2.4 Finding an optimal policy

One way to find an optimal policy is the algorithm called policy iteration. The idea behind this algorithm is that we iteratively evaluate current V_π which gives us expected returns for each state. After having all value functions computed we can argue (the proof is provided in Sutton & Barto 1998), that taking actions greedily with respect to V_π leads to a better policy and therefore to a new value function. These two processes depicted in Figure 2.3 are called policy evaluation and policy improvement, or also referred to as prediction and control, respectively.

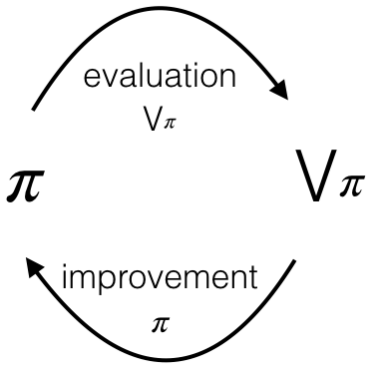


Figure 2.3: Policy iteration consisting of two steps, policy evaluation and policy improvement.

2.2.5 Monte Carlo methods

In previous method we silently assumed that we know the transition distribution. In many RL problems we do not have such information available, however, it is often possible to generate sample transitions from which we can learn. Monte Carlo methods allow to estimate the value function based on averaging sample returns.

Exploration versus exploitation

If the model of the environment is not available, then it is useful to estimate the Q function instead of the value function V. Deterministic policy would cause the algorithm to follow only one way, which is currently the best performing one. To make policy evaluation work, we must assure continual exploration in order not to miss possible better actions. Introducing action exploration will help mitigate this issue, e.g. with a probability ϵ we would choose a random action and with probability $1 - \epsilon$ we would pick an action according to policy π. This is called ϵ-greedy policy which ensures that in infinity each action–state pair is selected infinitely many times.

Monte Carlo prediction and control

Algorithm 2.1 Monte Carlo Prediction

$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$rewards(\boldsymbol{s}) \leftarrow$ an empty list, for all $\boldsymbol{s} \in S$
while TRUE **do**
  Generate an episode using π
  for all $\boldsymbol{s}$ in episode **do**
    $G \leftarrow$ reward following the first occurrence of $\boldsymbol{s}$
    append G to $rewards(\boldsymbol{s})$
    $V(\boldsymbol{s}) \leftarrow average(rewards(\boldsymbol{s}))$
  end for
end while

Algorithm 2.1 will learn a state–value function for a given policy π. By the law of large numbers, the sequence of averages of these estimates converges to its expected value. The same algorithm will work for problem without a model. Instead of V we will estimate the Q function and policy will be

ϵ-greedy. To update the policy we would again change it such that it follows the best actions greedily according to the Q function.

2.3 Value function approximation

Working with large state-action spaces can be memory demanding. So instead of remembering all the states we will try to approximate V function or action–state Q function. This is not as easy as it may seems because not all function approximation methods are well suited for reinforcement learning.

In RL it is important for a method to learn on-line while interacting with the environment. To efficiently learn in this setting we require methods which are able to learn in such way. In addition we will also require from the learning algorithm to be able to handle non-stationary target function. For example in policy iteration we often seek to learn Q_π while π is changing. Methods that can not handle these requirements well enough are less suited for reinforcement learning.

2.3.1 Learning objective

Most function approximators are based on gradient descent optimization. In order to train one we need some learning objective which measures how well we approximate function. One possibility for such a measure would be mean squared error (MSE):

$$\text{MSE} = \sum_{\boldsymbol{s} \in S} d(\boldsymbol{s}) \Big[V_\pi(\boldsymbol{s}) - V(\boldsymbol{s}|\boldsymbol{\theta}) \Big]^2$$

The square root of this function gives a rough estimate how much the predicted and real values differ. As part of the equation there is a term $d(\boldsymbol{s})$ which says of how much we care about errors in different states. The most common choice of $d(\boldsymbol{s})$ is a fraction of time spent in $\boldsymbol{s}$ under the policy π.

2.3.2 Linear methods

One way to optimize such function is to use a linear model $\hat{V}(\cdot, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is a vector of parameters. Corresponding to every state $\boldsymbol{s}$, there exists a real valued vector of features $\phi(\boldsymbol{s}) = [\phi_1(\boldsymbol{s}), \phi_2(\boldsymbol{s}), ...]^\top$. These features can be constructed in many ways such as with polynomial basis, Fourier basis or use various types of encoding (tile or coarse), or its generalization radial basis functions. Computing the state–value function is as easy as doing the dot product of two vectors:

$$\hat{V}(\boldsymbol{s}, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \phi(\boldsymbol{s}) = \sum_{i=1}^{n} \boldsymbol{\theta}_i \phi_i(\boldsymbol{s})$$

To find optimal weights $\boldsymbol{\theta}$ we use standard methods such as GD or SGD.

2.3.3 Nonlinear methods

As nonlinear models, neural networks are the most common choice. They were successfully used for creating meaningful representations from raw images or other different sensor signals. Compared to linear models they are much harder to train due to the introduced nonlinear functions in hidden layers and we can not guarantee finding an optimal solution.

2.4 Policy gradient methods

In contrast to previous method where we learned the state value or action–state value function and acted greedily with respect to this function, now we will learn parametrized policy. We will control parameters of parametrized policy $\pi_{\boldsymbol{\theta}}$ which affects probability distribution over actions and according to this distribution we will choose our next action. Note that in policy gradient method we do not need to consult value function prior to selecting the next action, but the value function may still be used to learn the policy weights.

2.4.1 Score function gradient estimator

Let us introduce how to compute the gradient $\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\boldsymbol{x}}\Big[f(\boldsymbol{x})\Big]$ of expectation of function f:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\boldsymbol{x}}\Big[f(\boldsymbol{x})\Big] &= \nabla_{\boldsymbol{\theta}} \int p(\boldsymbol{x}|\boldsymbol{\theta}) f(\boldsymbol{x}) dx \\
&= \int \nabla_{\boldsymbol{\theta}} p(\boldsymbol{x}|\boldsymbol{\theta}) f(\boldsymbol{x}) dx \\
&= \int \frac{p(\boldsymbol{x}|\boldsymbol{\theta})}{p(\boldsymbol{x}|\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} p(\boldsymbol{x}|\boldsymbol{\theta}) f(\boldsymbol{x}) dx \\
&= \int p(\boldsymbol{x}|\boldsymbol{\theta}) \overbrace{\frac{1}{p(\boldsymbol{x}|\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} p(\boldsymbol{x}|\boldsymbol{\theta})}^{\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}|\boldsymbol{\theta})} f(\boldsymbol{x}) dx \\
&= \int p(\boldsymbol{x}|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}|\boldsymbol{\theta}) f(\boldsymbol{x}) dx \\
&= \mathbb{E}_{\boldsymbol{x}}\Big[f(\boldsymbol{x}) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}|\boldsymbol{\theta})\Big]
\end{aligned}$$

In order to use this estimator, we need to be able to differentiate density $p(\boldsymbol{x}|\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. This is an unbiased gradient estimator. We can

just sample from $\boldsymbol{x}_i \sim p(\boldsymbol{x}|\boldsymbol{\theta})$, and compute $\hat{g} = f(\boldsymbol{x}_i)\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}_i|\boldsymbol{\theta})$.

2.4.2 Policy gradient

To update parameters of our policy, we first need its gradient and to acquire the gradient we need some loss function. So to formulate our problem we want to

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \quad \mathbb{E}_\tau\Big[R(\tau)|\pi_{\boldsymbol{\theta}}\Big]$$

where $R(\tau)$ is the cumulative reward across the whole trajectory (episode). Intuitively, we want to make good trajectories more probable, therefore making good actions more probable in specific states. Note that here we want to maximize the cumulative reward, so instead of the gradient descent we will be doing gradient ascend.

Sampling the trajectory $\tau = (\boldsymbol{s}_0, \boldsymbol{a}_0, r_0, \boldsymbol{s}_1, \boldsymbol{a}_1, r_1, ..., \boldsymbol{s}_{T-1}, \boldsymbol{a}_{T-1}, r_{T-1}, \boldsymbol{s}_T)$ and using it as $\boldsymbol{x}$ (Section 2.4.1) we can compute gradient of our loss function

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathbb{E}_\tau\Big[R(\tau)\Big] &= \mathbb{E}_\tau\Big[R(\tau)\nabla_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta})\Big] \\ &= \mathbb{E}_\tau\Big[R(\tau)\nabla_{\boldsymbol{\theta}} \sum_{t=0}^{T-1} \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{\theta})\Big]\end{aligned}$$

Sampling just one trajectory and computing the gradient will not be a really good estimate of expected value and therefore of the gradient. To reduce the variance we can sample more such trajectories and compute the gradient with respect to all of them. To summarize this, we have the following algorithm:

Algorithm 2.2 Policy gradient

1: Initialize policy parameter $\boldsymbol{\theta}$
2: **while** TRUE **do**
3: Collect a set of trajectories by executing current policy
4: For each trajectory compute return $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t$
5: Update the policy using policy gradient estimate
6: **end while**

Please note that this is just a brief summary of theory of RL. We have only covered parts required for understanding a policy gradient and there exists much more methods and approaches. For further material refer to Sutton & Barto (1998).

Chapter 3

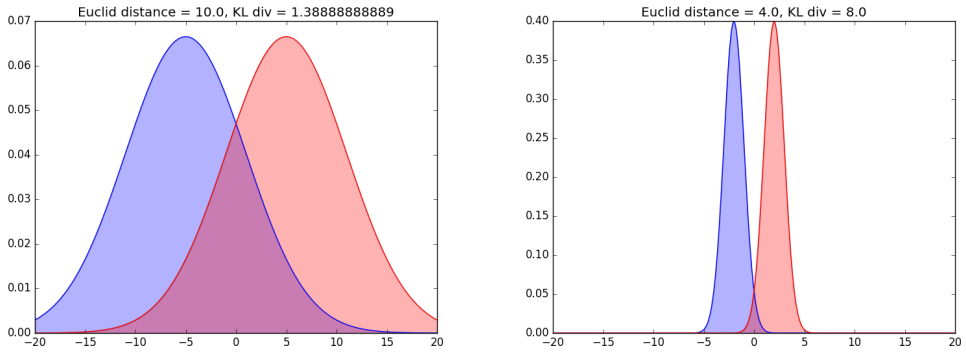
Policy gradient upgrades

In this chapter we describe two quite recent methods which build on policy gradient by exploiting ideas from information geometry. The first such method is the natural policy gradient (NPG) which applies the idea of natural gradient proposed by Amari (1998). The second method is called the trust region policy optimization (TRPO) which improves the NPG by introducing trust region constraint to policy optimization (Schulman et al., 2015).

3.1 Natural gradient

Natural gradient can be followed back to Amari (1998) and his work on information geometry. Later the algorithm was applied in reinforcement learning community by Kakade (2001). To understand the difference between PG and NPG, we first need to reconsider the notion of distance. The folk axiom that "the shortest distance between two points is straight line" represented by Euclidean distance often does not hold in the real world, e.g. in geograph-

ical maps the shortest distance is often some curve due to the shape of our planet.



(a) (b)

Figure 3.1: Notion of distance in the realm of Gaussian distribution. (a) Large Euclidean distance and small KL divergence. (b) Small Euclidean distance and large KL divergence.

It is similar also in the case of probability distributions. In Figure 3.1a we can see that Euclidean distance between the parameters of two normal distributions is 10 while they are much more similar than distributions in Figure 3.1b where Euclidean distance is only 4. The better distance measure is KL divergence which basically tells us how similar two distributions are. The formula for continuous distributions is

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}$$

Natural policy gradient uses exactly this idea, i.e. instead of using Euclidean distance it applies the KL divergence to measure distance between distributions. In mathematical form the Euclidean distance in N- dimensional space between two points $\boldsymbol{v}$ and $\boldsymbol{v} + \delta\boldsymbol{v}$ is

$$d_E(\boldsymbol{v}, \boldsymbol{v} + \delta\boldsymbol{v}) = \sqrt{\sum_{i=1}^{N} \delta\boldsymbol{v}_i^2} = \sqrt{(\delta\boldsymbol{v})^\top \delta\boldsymbol{v}}$$

In Riemannian geometry upon which natural gradient adaptation is built, we generalize the distance metric $d_{\boldsymbol{w}}(\cdot, \cdot)$ at point $\boldsymbol{w}$ as

$$d_w(\boldsymbol{w}, \boldsymbol{w} + \delta\boldsymbol{w}) = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N} \delta\boldsymbol{w}_i \delta\boldsymbol{w}_j g_{ij}(\boldsymbol{w})} = \sqrt{\delta\boldsymbol{w}^\top \mathbf{G}(\boldsymbol{w})\delta\boldsymbol{w}}$$

where the Riemmanian metric tensor $\mathbf{G}(\boldsymbol{w})$ is the positive definite matrix of dimensions $N \times N$ where(i, j)-th entry is $g_{ij}(\boldsymbol{w})$. The matrix $\mathbf{G}(\boldsymbol{w})$ characterizes the curvature of particular manifold in N-dimensional space. In case of Euclidean distance $\mathbf{G}(\boldsymbol{w}) = \mathbf{I}$ so the $d_{\boldsymbol{w}}(\cdot, \cdot)$ reduces to $d_E(\cdot, \cdot)$. Another similar example would be in polar coordinates where Riemmanian metric tensor is

$$\mathbf{G}(\boldsymbol{w}) = \begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix}.$$

As we already pictured, using Euclidean distance in parametrized distributions is not the best idea. The standard gradient of loss function does not represent the steepest direction in the parameter space and thus using the standard gradient update is not appropriate. Therefore, Amari (1998) proposed natural gradient adaptation as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \mathbf{G}^{-1}(\boldsymbol{\theta}_t) \frac{\partial L}{\partial \boldsymbol{\theta}}$$

The main advantage of natural gradient adaptation over standard gradient descent is that we are also estimating the curvature of underlying parameter space which allows as to converge much faster. One such example where

standard SGD fails is when it is trapped in a plateau. It might take a long time for SGD to escape while for natural gradient it is easy due to having curvature information available.

3.2 Natural policy gradient

When Kakade (2001) applied natural gradient to policy gradient he named it natural policy gradient. With a slightly abused notation we will write expected reward as $\eta(\boldsymbol{\theta})$ instead of $\eta(\pi_{\boldsymbol{\theta}})$.

The average reward is technically a function on the set of distributions $\{\pi_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in R^N\}$. For each state $\boldsymbol{s}$ there corresponds a probability manifold, where the distribution $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$ is a point on this manifold with coordinates $\boldsymbol{\theta}$ and the Fisher Information Matrix (FIM) of this distribution is

$$\boldsymbol{F}_{\boldsymbol{s}}(\boldsymbol{\theta}) = \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}) \left[\frac{\partial \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\partial \boldsymbol{\theta}_i} \; \frac{\partial \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\partial \boldsymbol{\theta}_j} \right]$$

As shown by Amari (1998), FIM is an invariant metric on the space of the parameters of probability distributions up to some scale. Invariant in the sense of how we choose the coordinates $\boldsymbol{\theta}$. Since the expected reward is defined on the set of these distributions, Kakade's (2001) choice of the metric was

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{\rho^{\pi}(\boldsymbol{s})}[\mathbf{F}_{\boldsymbol{s}}(\boldsymbol{\theta})]$$

where the expectation is taken with respect to stationary distribution $\pi_{\boldsymbol{\theta}}$. Intuitively, FIM measures the distance on a manifold in state $\boldsymbol{s}$ and $\mathbf{F}_{\boldsymbol{s}}$ is an

average distance. Hence our policy update will be

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \mathbf{F}^{-1}(\boldsymbol{\theta}_t) \nabla \eta(\boldsymbol{\theta})$$

.

3.3 Trust region policy optimization

Trust region policy optimization (TRPO) algorithm is quite similar to NPG method. It builds on Kakade & Langford (2002) work where they introduced policy updating scheme called conservative policy iteration, for which they could provide explicit lower bounds on improvement of η when used with mixture policies. Schulman et al. (2015) showed that it is possible to extend this result by applying it to general stochastic policies. Since mixture policies are rarely used in practice, this result was crucial for extending the improvement guarantee to practical problems. Using local approximation on such algorithm led to TRPO.

3.4 Optimization problem

After few approximations to theoretical results shown in Kakade (2001) and Schulman et al. (2015), we end up with the optimization problem formulated in the following way:

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{maximize}} \quad & \mathbb{E}_{s \sim \rho_{\boldsymbol{\theta}_{\text{old}}}, \boldsymbol{a} \sim q} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{q(\boldsymbol{a}|\boldsymbol{s})} Q_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a}) \right] \\ \text{subject to} \quad & \mathbb{E}_{s \sim \rho_{\boldsymbol{\theta}_{\text{old}}}} \left[D_{\text{KL}}(\pi_{\boldsymbol{\theta}_{\text{old}}}(\cdot|\boldsymbol{s}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s})) \right] \leq \delta \end{aligned}$$

where $\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{q(\boldsymbol{a}|\boldsymbol{s})}Q_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{s},\boldsymbol{a})$ is an unbiased cumulative reward of new policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$, $q(\boldsymbol{a}|\boldsymbol{s})$ is our sampling scheme and $\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{q(\boldsymbol{a}|\boldsymbol{s})}$ is an importance sampling ratio. D_{KL} is an KL-divergence of two probability distributions, in this case our stochastic policies and δ is hyperparameter of the model describing how much the two policies can differ.

Algorithm 3.1 TRPO

1: Initialize policy parameter $\boldsymbol{\theta}$
2: **while** TRUE **do**
3: Use sampling scheme to collect a set of state-action pairs with cumulative rewards.
4: By averaging over samples, construct the estimated objective and constraint.
5: Compute search direction using linear approximation to objective and quadratic to constraint.
6: Perform line search in that direction such that we improve $L(\boldsymbol{\theta})$ and satisfy constraint.
7: **end while**

3.4.1 Sampling scheme

Schulman et al. (2015) describe two different sampling schemes - single path and vine. In single path we sample multiple states from which we then follow policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$ for some number of time-steps to generate trajectories. In vine method we additionally select multiple actions in multiple states to better approximate cumulative reward. However, this would require simulator to be restored to particular states. In our implementation we decided to focus on a simple path sampling scheme due to its better computational efficiency. Also it is results turned out to be sufficient for not implementing a much more complicated vine procedure.

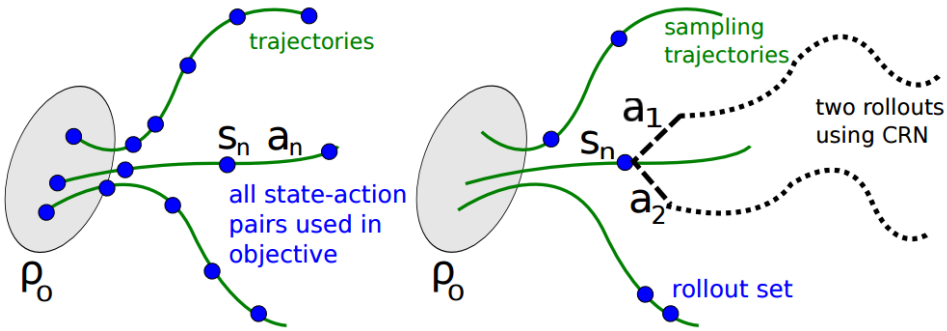


Figure 3.2: Demonstration of two sampling schemes, single path (left) and vine sampling procedure (right). (Source: Schulman et al. 2015)

3.4.2 Trust region and search direction

In order to update our policy parameters $\boldsymbol{\theta}$ we first need to compute the direction in which it will increase $\eta(\boldsymbol{\theta})$. For this we will use natural gradient, which in turn means that we need to solve the equation $\mathbf{F}\boldsymbol{x} = \boldsymbol{g}$ where $\boldsymbol{x}$ is a search direction which we need, $\boldsymbol{g}$ is the policy gradient and $\mathbf{F}$ is a Fisher Information Matrix and $\mathbf{F}_{ij} = \frac{\partial}{\partial \boldsymbol{\theta}_i}\frac{\partial}{\partial \boldsymbol{\theta}_j} D_{\mathrm{KL}}(\boldsymbol{\theta} \parallel \boldsymbol{\theta}_{\mathrm{old}})$. However, computing $\mathbf{F}$ directly with over-parametrized models such as neural networks can be memory and computationally expensive. Using theory on multiplying Hessian with vector (Pearlmutter, 1994) with conjugate gradient method we can approximately solve $\boldsymbol{x} \approx \mathbf{F}^{-1}\boldsymbol{g}$.

Having the search direction computed, we need to find the maximal step length β such that $\boldsymbol{\theta} + \beta\boldsymbol{x}$ satisfies our optimization constraint. Applying the quadratic approximation to

$$D_{\mathrm{KL}}(\boldsymbol{\theta} \parallel \boldsymbol{\theta}_{\mathrm{old}}) \approx \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{old}})^{\top}\mathbf{F}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{old}})$$

we can derive that

$$\delta = D_{\mathrm{KL}} \approx \frac{1}{2}(\beta \boldsymbol{x})^{\top} \mathbf{F}(\beta \boldsymbol{x}) = \frac{1}{2}\beta^2 \boldsymbol{x}^{\top} \mathbf{F} \boldsymbol{x}$$

$$\beta = \sqrt{\frac{2\delta}{\boldsymbol{x}^{\top} \mathbf{F} \boldsymbol{x}}}$$

Now we computed the trust region β for our search direction we can employ it to update policy parameters.

Chapter 4

Related approaches

In this chapter we describe other RL approaches which are currently considered state-of-the-art in various domains. A lot of them fall into the category of action-value function estimation where we act greedily by taking the action with the highest estimated value. At first we describe Deterministic Policy Gradient and its update Deep Deterministic Policy Gradient which are methods built on the policy gradient idea. Then we offer overview on Deep-Q Learning methods based on estimation of Q values. Then we continue by its improvements called Double Deep Q learning and Dueling Deep Q learning followed by brief report about Asynchronous Advantage Actor Critic and Normalized Advantage Function algorithms. In the end of we describe some other methods for robotic grasping.

4.1 Modern RL approaches

4.1.1 Deterministic policy gradient (DPG)

DPG comes from the family of actor–critic methods. In this method authors showed that it is possible to change a stochastic policy used in the standard policy gradient or TRPO for deterministic one (Silver et al., 2014). This brought the slight advantage of easier estimation of gradients. Silver et al. (2014) showed that in many multidimensional cases having deterministic policy brings advantage because we do not have to do Monte-Carlo estimations over the action space.

Deep DPG

Closely after DPG was proposed, Lillicrap et al. (2015) showed how to employ deep learning with DPG by using the idea of replay memory and training network with Q value. They suggested to minimize a correlation between the samples by utilizing the memory replay and applied deep learning with batch normalization for easier convergence (Ioffe & Szegedy, 2015) in Q function estimation. They also showed that this algorithm was able to learn good policies for various motor control tasks such as locomotion, reaching and grasping.

4.1.2 Deep Q learning

However, the most famous application of deep networks in reinforcement learning was when Mnih et al. (2013) combined Q-learning with deep neural

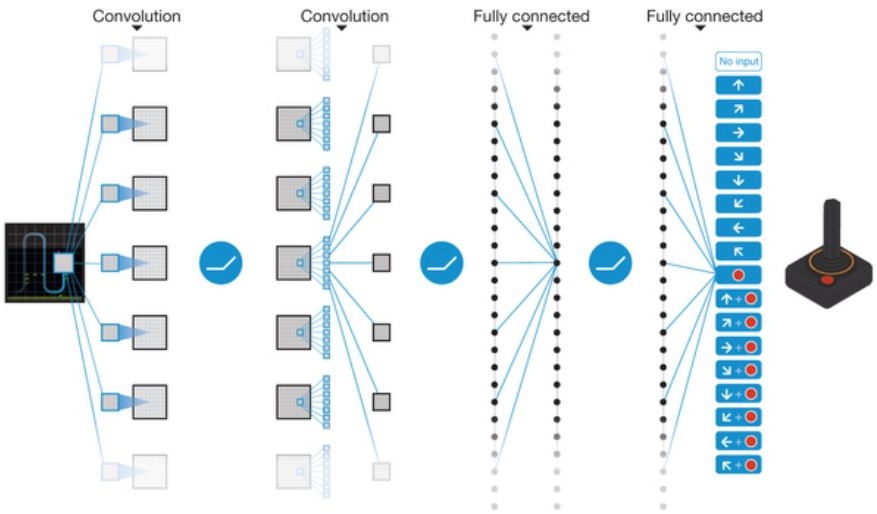


Figure 4.1: Neural network architecture used by Deep Q-learning, (Mnih et al. 2015).

nets on the Atari gaming domain (Mnih et al., 2015). To be more specific, they modified Q learning by adding additional target network whose purpose was to stabilize learning and they also utilized experience replay (Lin, 1992) as another stabilisation technique. What made it famous was that they designed end-to-end system (Figure 4.1) for learning from raw image pixels received from a game simulator. Their resulting policy was able to outperform a human player in multiple games.

Zhang et al. (2015) successfully applied this algorithm on robotic reaching in simulated environment. They also unsuccessfully tried to transfer the learned policy on real world Baxter robot. The author's belief of reason for failure in knowledge transfer from simulation to real world was a large difference in an input image between real and simulation scenarios.

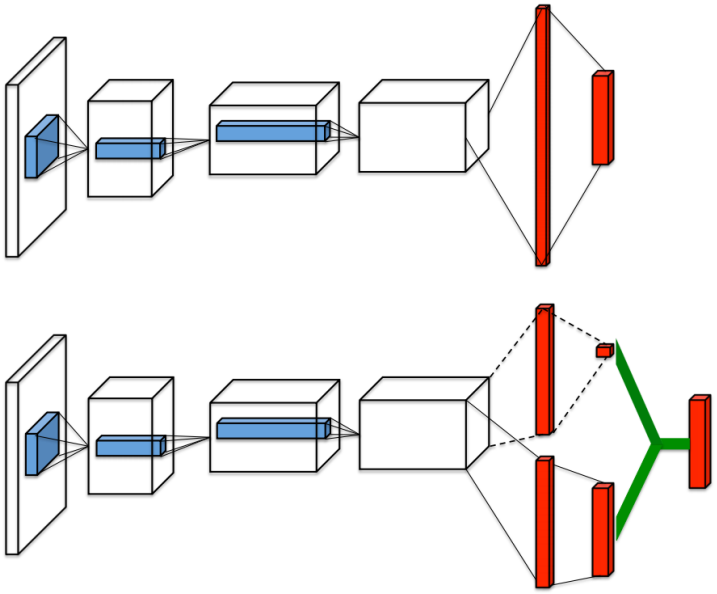


Figure 4.2: Comparison of standard DQN and new Dueling DQN, (Wang et al. 2016).

Double DQN

Closely after DQN was published, van Hasselt et al. (2015) described the known problem of Q learning which is overestimation and also showed why it happens. They also showed that its possible to fix this problem and proposed Double DQN (DDQN) which reduced this overestimation bias and made learning more stable. As they showed this led to better policies for almost each game played in Atari simulator.

Dueling DQN

Following the great results of previous two architectures the next step for update was development of Dueling DQN. The crucial insight was that sometimes it is not necessary to estimate each action separately but instead only

estimate state value and therefore they decided to decompose Q function estimation into estimating a value function V and advantage function A, where $Q = V + A$ (Figure 4.2). After decomposing it and predicting each one separately they again merge it together so they could still train the whole network in DQN/DDQN fashion.

Following research

Many new papers followed the research of DQN, one such method was proposed by Mnih et al. (2016) called asynchronous advantage actor critic (A3C) where they showed that it is possible to use multiple parallel actor learners for stabilizing effect on the learning process instead of experience replay. However, they also stated that using both techniques could lead to even better learning efficiency. They also showed that it is possible to beat current state-of-the-art trained on modern GPU with only 16 core CPUs by using the A3C algorithm.

However, these algorithms proved to be really good for high-dimensional observation spaces they are hard to apply on continuous action spaces. An obvious approach would be to discretize the action space but this have many limitations, most notably the curse of dimensionality. In high-dimensional action spaces policy gradient methods show much better performance.

Gu et al. (2016) proposed a way how to mitigate the need of discretizing continuous action spaces while still estimating Q function by constructing a network which estimate state function V and advantage function A. They also proposed an imagination rollouts which were trying to use learned model

to help estimate Q values more precisely, but it was shown not to help.

We believe that these updates for faster convergence and better policies could be applied in the same fashion as in Zhang et al. (2015). There definitely exist more similar techniques performing really well, but the one described we believe to be the most promising ones.

4.2 Other approaches for grasping

4.2.1 Demonstration

Castellini et al. (2007) created a dataset consisting of reach and grasp trajectories collected by the CyberGlove (a glove with sensors for recording movements of human hand) consisting of 22 sensors and then used it with support vector machines to predict a degree value for each DoF on human hand. This could be then used by robotic arm for generating valid reaches and grasps.

4.2.2 Cognitive approaches

Kraft et al. (2010) studied the means of learning robotic grasp affordances such as relative gripper-object poses that lead to stable grasps. They use a visual model of object to autonomously search and infer stable grasps. They also showed that it is possible to repeatedly grasps an object laying in arbitrary pose where each pose imposed a specific reaching constraint.

4.2.3 Vision

Visual grasping learning is modern approach which use a big datasets of grasps to train a classifier which can either tell if some grasp will be successful or even predict where to grasp object (Saxena et al., 2008). Later, Lerrel Pinto (2016) introduced a large scale dataset for object grasping learning and similar approach of large scale data collection was described by Levine et al. (2016). Similar research was perform by Joseph Redmon (2015) where they trained classifier for predicting grasp successfulness. These methods were inspired by enormous success of convolutional network popularized by Krizhevsky et al. (2012).

Chapter 5

Experiments

In this chapter we describe our experiments of reaching and grasping on multiple environments. Firstly, we describe how we represent policy and the value function, then we continue by describing an experiment on simple 2D reacher, followed by 3D reaching and grasping. Finally, we experiment with 3D grasping behind the static wall with randomly placed target behind it. In each environment degrees of freedom (DoF) are controlled by our policy outputting torques. These torques are then applied to the robotic model inside the simulator.

5.1 Model specifications

5.1.1 State representation

In our simulated world the state **s** was represented by a vector of positions for each DoF followed by the target position and the actual forces/torques

on each DoF. After observing the current state and reward for the last step, agent provides one action for each DoF. The action was represented as a force/torque applied to each DoF, respectively.

5.1.2 Policy

As our stochastic policy we used a fully connected artificial neural network, as described in Section 2.1. In order to scale on each environment we have taken a relatively big network consisting of two hidden layers with 100 neurons in each. As a nonlinear activation function we used the hyperbolic tangent, and as an initialization method we used Xavier initialization (Glorot & Bengio, 2010).

We experimented with activation functions such as ReLU variations and sigmoidal nonlinearities. However, we found that TRPO is quite invariant to these changes and converged almost in all settings with minor improvements in some combinations, therefore we decided to stick to a standard tanh activation function. Our experiments with number of layers and neurons in each layer showed that using smaller networks improved convergence in easier tasks such as 2D reaching, but was not sufficient in 3D grasping. Therefore, we decided to further proceed with already mentioned architecture which provides more than enough parameters for learning grasping policy.

5.1.3 Value function

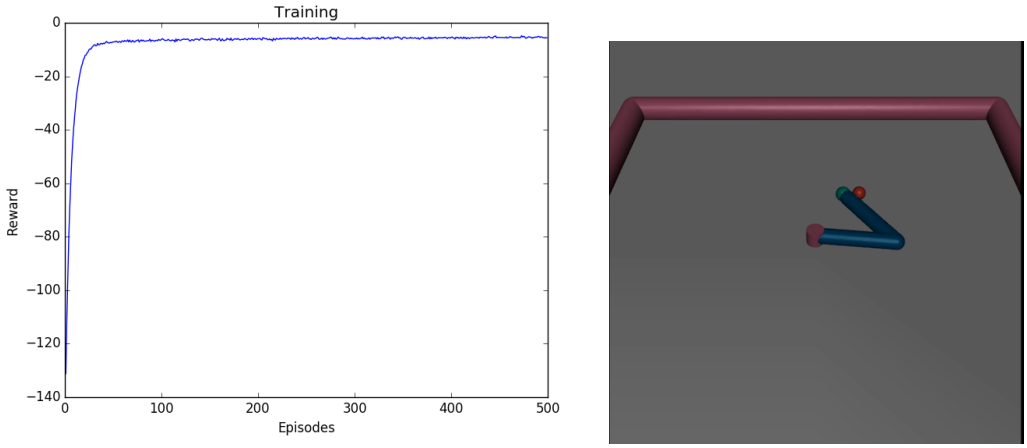
As a value function approximator we decided to employ a simple linear regression. so for training we could calculate closed form for least squares.

However when trying to learn policy we discovered that we were unable to learn any meaningful value function approximation due to non-linearity of an underlying value function. We partially solved this by nonlinearizing input to regression by giving it also squares of each input dimensions.

We also experimented with the neural value function approximator which we believed to be better suited for the underlying value function. We tried different feedforward architectures with different training procedures but we were not able to beat linear regression. Computing closed form of linear regresson with squared input features was fast and robust, therefore we decided to continue with it.

5.1.4 TRPO parameters

In TRPO the most sensitive hyperparameters are maximum KL divergence and the number of rollouts per update. This is probably intuitive since the whole algorithm convergence properties depend on these two. In all our experiments we used maximum KL divergence $\delta = 0.01$. Regarding the number of sampled trajectories we increased it in the case of more difficult environment.



(a) (b)

Figure 5.1: A cumulative reward during training (left). Rendered Reacher2D environment (right).

5.2 Environments

5.2.1 2D Reacher

As the first testing environment (Figure 5.1b) we decided to use a 2D reacher. This environment consists only of two DoF which should make it easy to solve. Therefore we decided to use this environment as testing ground for TRPO implementation. The goal of the environment is to reach for the red cube(target) with the green dot(palm). The red cube and both arm angles are given a random position at the beginning of each iteration to force the policy to find parametrisation that generalizes well and learns the most about relationship between the arms and the target.

As already described in Chapter 2 after each step environment provides a state and a reward. Based on this information the agent selects an action.

The agent's state is given by an angles for each DoF, position of the target in Cartesian coordinate system and the current velocity of each DoF. The reward is given by the following formula

$$R = -d(\text{palm,target}) - f(\text{actions}) \tag{5.1}$$

where

1. d(palm,target) is the distance from palm to target. The higher the distance the lower the reward is. This forces algorithm to find policy which tries to have green and red dot as near as possible.

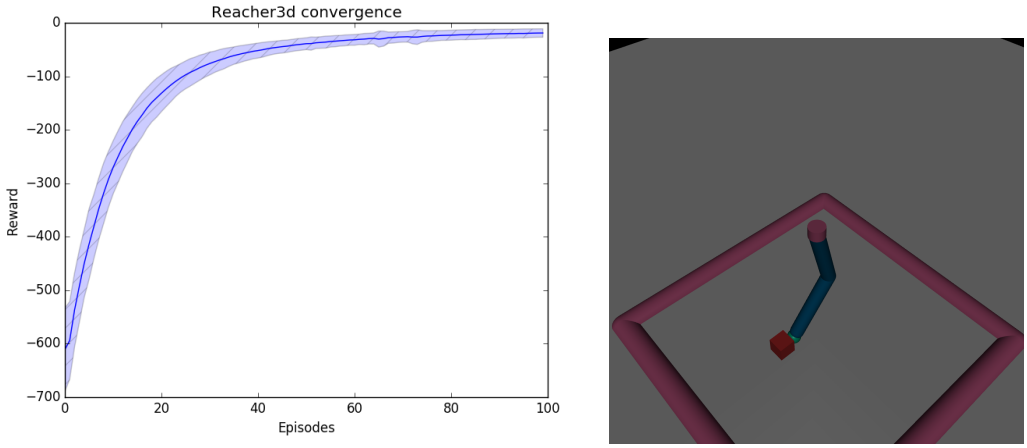
2. f(actions) is penalty for movement. We penalize robot for applying force on joints to ensures that robot try to find an optimal way to reach the target and do the minimal work.

After observing the current state and the reward for last step agent performs one action for each degree of freedom. This action is represented as force/-torque applied to each DoF respectively.

As we can see in Figure 5.1a such a problem formulation allows our algorithm to successfully converge to a good policy for reaching on randomly placed target. Being confident that our algorithm works we moved to a more complex environment.

5.2.2 3D Reacher

We have placed rigid ceiling into our new environment shown in Figure 5.2b from which our arm will hang. In order to reach almost every point on our



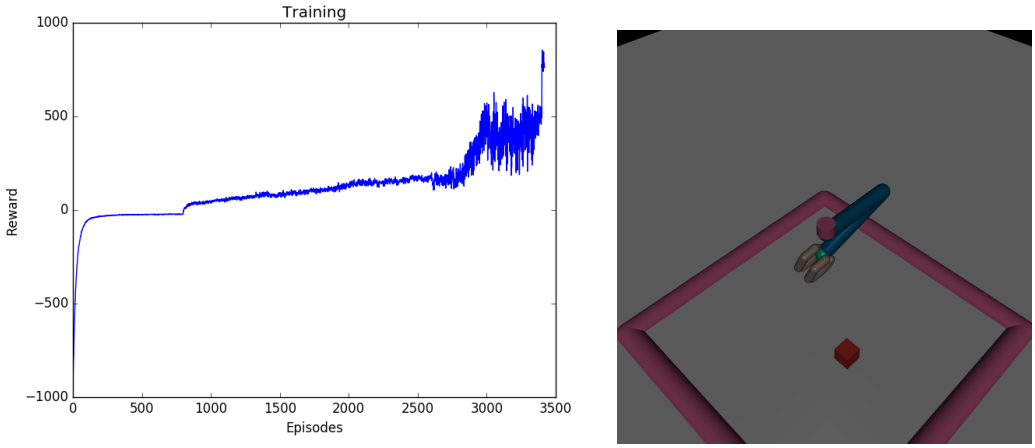
(a) (b)

Figure 5.2: A cumulative reward during training averaged over 10 runs with its standard deviation(left). Rendered Reacher3D environment (right).

table which is bounded by red walls, it required to create two additional degrees of freedom.

State, action and rewards were almost the same as in 2D reaching only now all computation were being done in 3D, i.e. to the state observation we added information about the additional DoFs and one dimension for the target position. Moreover, we needed to output two additional values in the action space.

When we compare Figures 5.1a and 5.2a we can notice that solving new environment is approximately as hard as solving the former one. In graph 5.2a we can also see that our reward during the early episodes starts with much lower value. This is partly caused by the two additional DoF. Firstly due to the reward penalizing forces applied to DoF and secondly by the fact that we can now achieve higher distances between the palm and the target.



(a) (b)

Figure 5.3: A cumulative reward during training 3D Grasper (left). Rendered 3D Grasper environment (right).

Policy controlling the robotic arm can also be viewed on video[1] from training.

5.2.3 3D Grasping

Extending 3D reaching environment to 3D grasping we added 4 fingers (Figure 5.3b). To generate valid grasps we decided to add rotational degree of freedom on the robotic forearm. Together this meant controlling seven degrees of freedom. After a several training trials we have noticed that having 2 degrees of freedom for each part of the gripper was unreasonable and therefore we decided to couple both sides together. The constraint property was to insure the inverse positions of grasper part to each other. This meant if left part of gripper is at 40 degrees than right part must be at -40.

A reward function in this environment was a bit more complicated than

[1] `https://youtu.be/_8uSyV750tE`

in the reaching part. It still had same two parts as in Equation 5.1 but with some additions which force arm to grasp the target. The first such part is a positive reward for contact between grippers and the target. The second part is penalty for touching ground with the gripper when there is contact between the gripper and target. The last part of the reward was for height of the target.

The penalty for ground contact, forced the arm not to touch the ground when it successfully grasped the target which in turn enforces the arm to lift the object. In the end such behaviour lead to big positive reward for target height. In Figure 5.3a there is a big spike around 0 and then the reward starts to have more variance. This was caused by the fact that the arm managed to pick up the target for the first time. After this event the reward slowly goes to the positive spectrum of values, until it reaches average reward around 900 which means that it was able to pick up the target almost in each trial. The high variance in the end of training was caused by the successfulness of grasping. Sometimes we were able to grasp a lot of times and other times we picked up the cube only occasionally. Training progress is visualized in Figure 5.3a as well as in the video[2] from training.

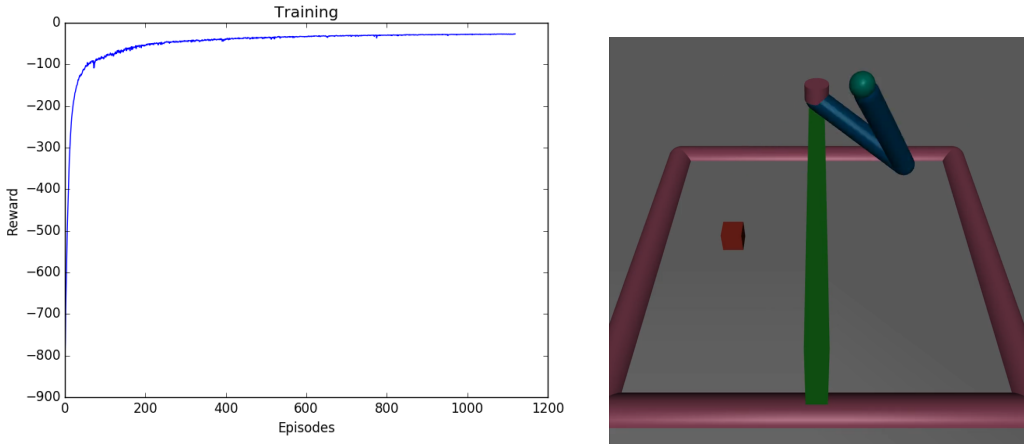
5.2.4 Reaching behind the wall

Simple grasping as shown in previous section is really useful for various robotic tasks, but for robots operating in the real world there are often different obstacles on the way to target which needs to be avoided so that neither robot nor obstacle are destroyed. Hence, we have designed another simple

[2] `https://youtu.be/EGvkJBXAOi4`

environment (Figure 5.4b) with the wall in the middle of the reaching area.

At the beginning of the episode the target is spawned on a random position in the reaching area but not on the wall. Then robotic arm is moved to the opposite side of the wall as its target and its goal is to reach for that target without touching the wall. We started with testing the reward function used in previous environment but we were unable to find policy which avoided the wall, thus we again modified it. The modification penalized for each contact with wall as well as for being closer to the wall then to the target. This lead to desired behaviour of the arm going over the wall and then reaching for the target.



(a) (b)

Figure 5.4: A cumulative reward during training (left). Reacher 3D with a wall in the middle (right).

5.3 Improvements of TRPO algorithm

We proposed a few TPRO improvements we have tried in order to speed up the computation time or convergence rate. Firstly, we introduce experience replay adaptation (Lin, 1992) and an experiment on improving sample efficiency by using this idea. Secondly, we look into reusing previous gradient direction information by using it as an initialization in conjugate gradient algorithm. In the end, we experiment with our parallel version of TRPO algorithm and show that it is computationally superior to standard TRPO.

5.3.1 Replay memory

After performing multiple experiments we noticed that training TRPO is too time consuming. Hence we decided to speedup the algorithm convergence properties by reusing previous information. One of the first ideas was to use collected trajectories not only in current iteration but also in the future. Specifically we decided to keep trajectories from past k steps. The first idea was to use these previous trajectories for better value function approximation. However, it idea turned out to be unsuccessful because we would estimate the value function for the current policy using rewards from previous iterations. Since the value function is dependent on the policy, this would introduce an error. The second idea was to simply store trajectories from a few previous iterations and run algorithm on such data. However, after trying this and failing, we noticed that we are no longer optimizing our objective but instead we use value function approximations from previous iterations instead of the value for current policy. Finally, we corrected this flaw by using past samples with correctly computed state–action transitions

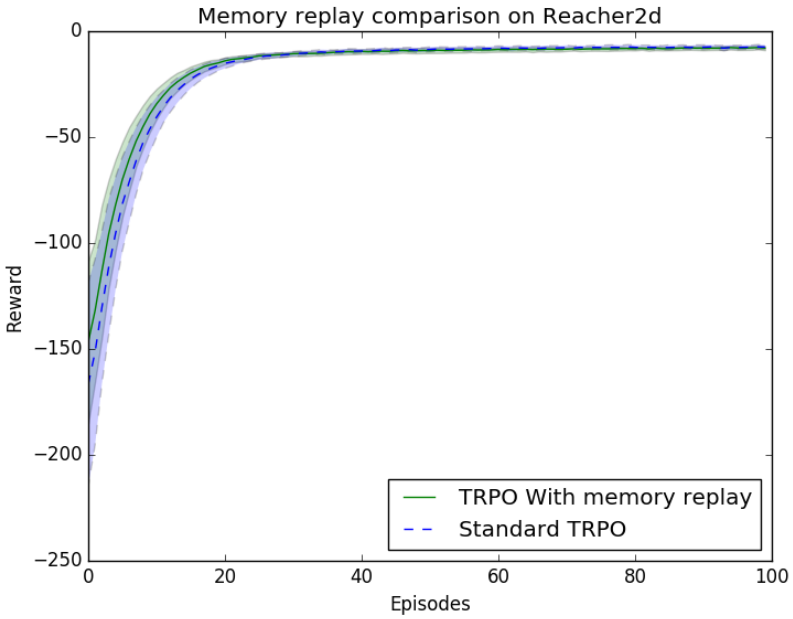


Figure 5.5: Memory replay convergence comparison on Reacher2D.

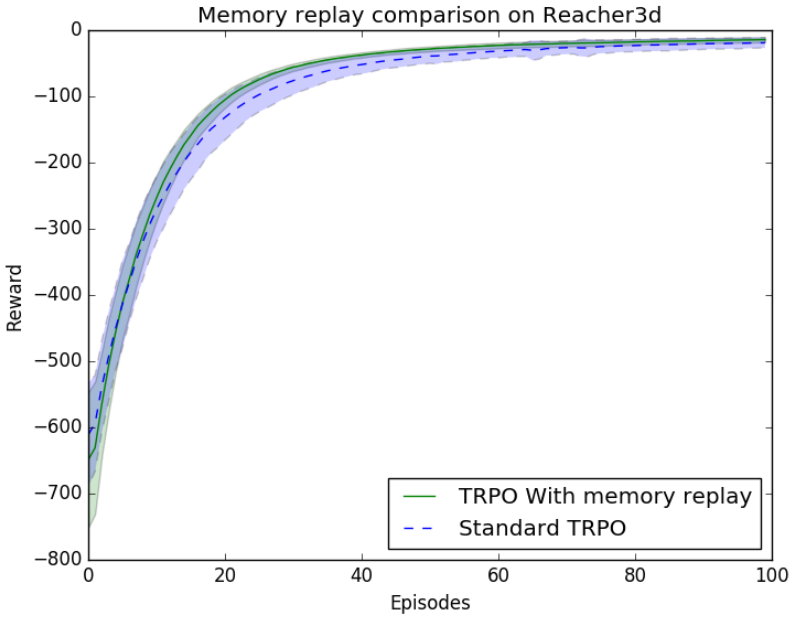


Figure 5.6: Memory replay convergence comparison on Reacher3D.

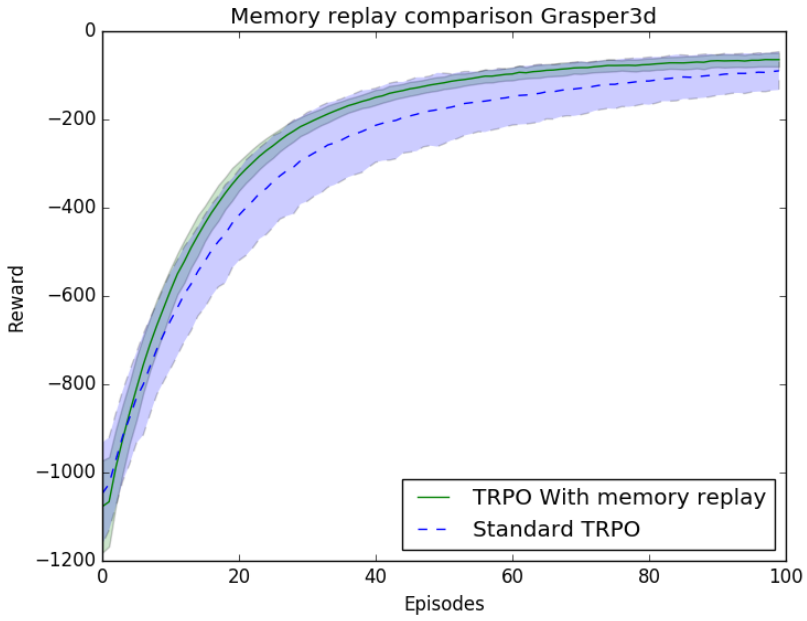


Figure 5.7: Memory replay convergence comparison on Grasper3D environment.

$Q_{\theta_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a})$ also for trajectories form past iterations. This idea led to minor improvement on each tested environment as can be observed in Figures 5.5, 5.6 and 5.7.

Experiment was conducted on 10 runs with a different random seed for each alternative. The dark blue/green line indicates the average of average sum of rewards per episode while a transparently colored surrounding indicates the variance. In each episode there were sampled 100 trajectories and replay memory hyperparameter was set to $k = 3$. For value function approximation we have used linear regression with nonlinearized input as described in Section 5.1.3.

As we can observe from Figures 5.5, 5.6 and 5.7 this technique reduces vari-

ance of rewards and also slightly improves performance of original algorithm consistently across multiple environments. It seems that the more complex the environment the more this method helps. Such a statement would require further investigation which we could not afford due to time/computational power constraints.

5.3.2 Using previous direction in CG

Another possibility of reusing previous information is based on reusing previous gradient instead of reusing experience explicitly as was done in replay memory case. Martens (2010) described that a simple enhancement to HF algorithm which they found improves performance by the order of magnitude was to reuse previous search direction found by conjugate gradient (CG) algorithm and apply it as starting point in CG in the current iteration. However, Martens (2010) only tried it with Gauss-Newton approximation to the Hessian matrix now we decided to also employ it with Fisher information matrix.

As the first experiment we incorrectly used only search direction instead of the whole update vector. This led to results which were either comparable with zero initialization or even worse. Using full step direction from previous iteration in initialization of conjugate gradient led to much better convergence rate and visible variance reduction of rewards. Parameter setting of experiments was exactly the same as in the replay memory section, only the algorithm was changed a little.

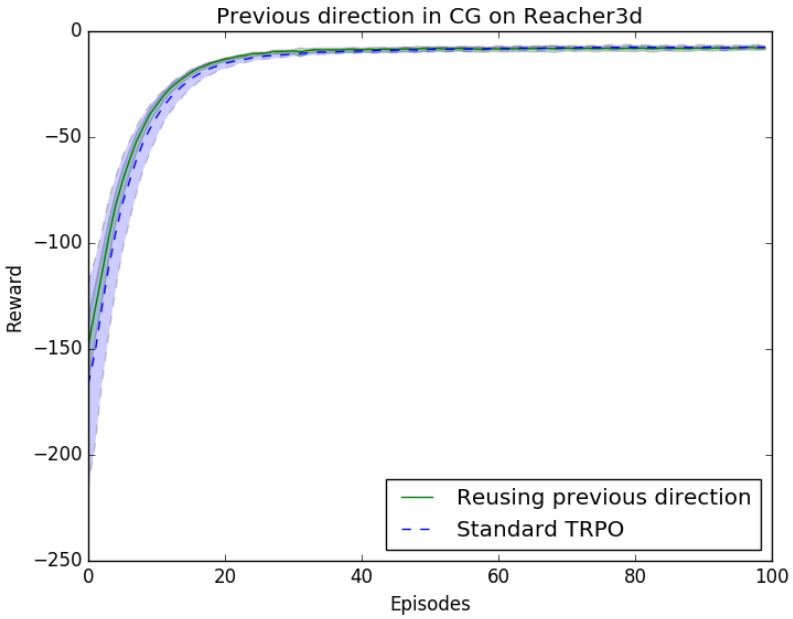


Figure 5.8: Previous direction reuse convergence on Reacher2D.

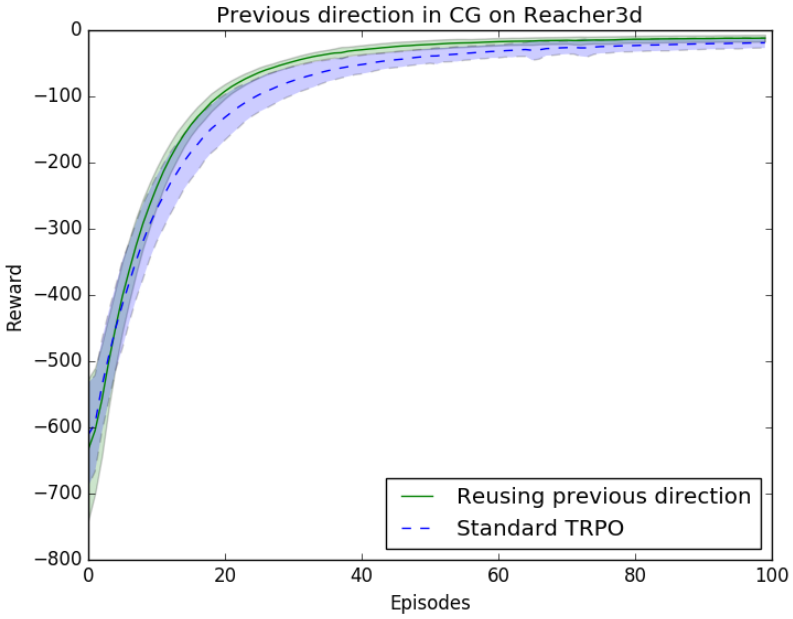


Figure 5.9: Previous direction reuse convergence on Reacher3D.

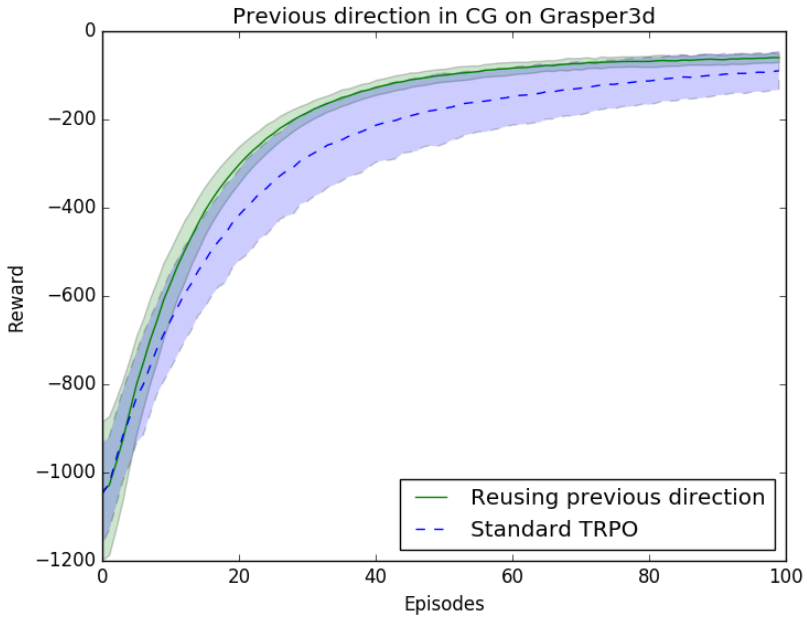


Figure 5.10: Previous direction reuse convergence on Grasper3D environment.

5.3.3 Combining information from previous iterations

Assuming that both previous methods improve convergence rate, we decided to combine them with the goal of maximizing usage of previously acquired information. However as depicted in Figures 5.11, 5.12 and 5.13, combining both approaches proved to be not a good idea. It seems that when we combine them, they contradict each other and perform worse compared to using either approach separately. It might be interesting to investigate why these two approaches do not work well together and achieve better convergence property than either of them alone.

Evaluation was again performed on all three environments, where each method was run exactly 10 times and rewards from different runs were averaged.

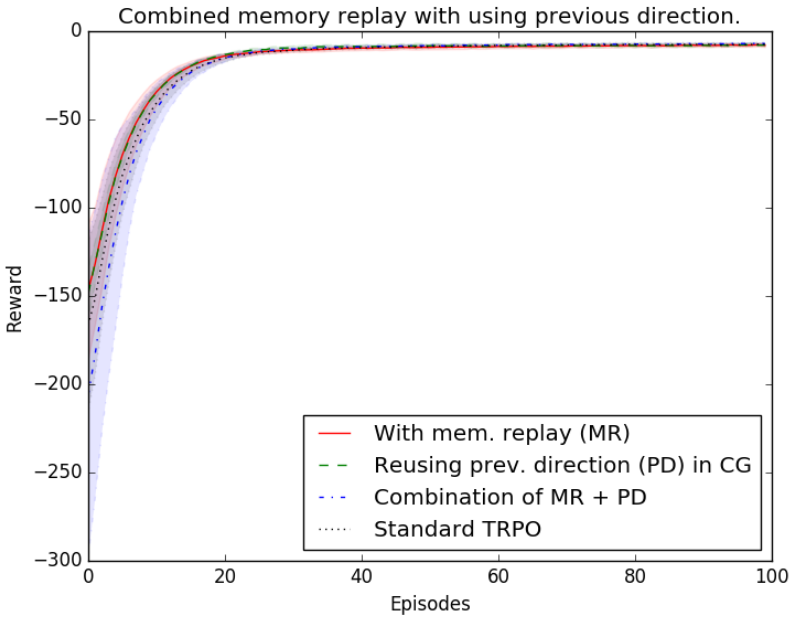


Figure 5.11: Combination of both methods on Reacher2D.

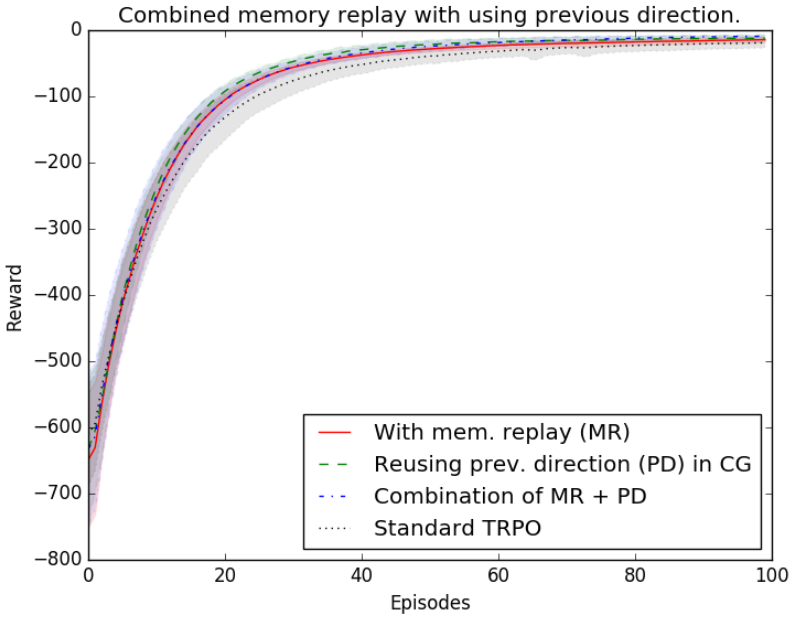


Figure 5.12: Combination of both methods on Reacher3D.

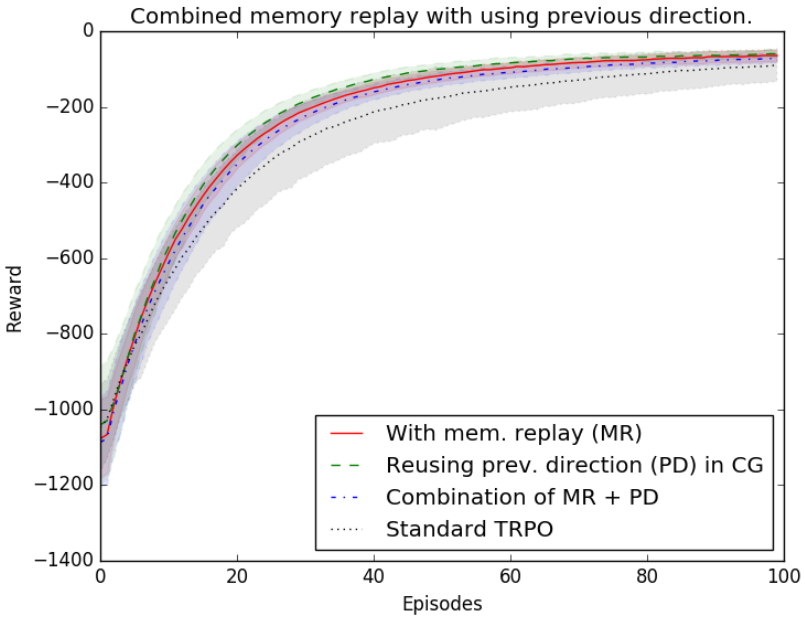


Figure 5.13: Combination of both methods on Grasper3 environment.

5.3.4 Parallelism

Nowadays when CPU frequency is no longer increasing parallelism is becoming hot topic in computing. Therefore using multiple processors for our training was of the shelve method for increasing computation speed. We have observed that we spend more than 95% of our computational time sampling paths and simulating rather than computing gradient and updating network. Gathering multiple trajectories on different cores is inherently parallel task. That means there is no computational dependency of one on an other. Knowing this, we expected the speedup to be almost linear.

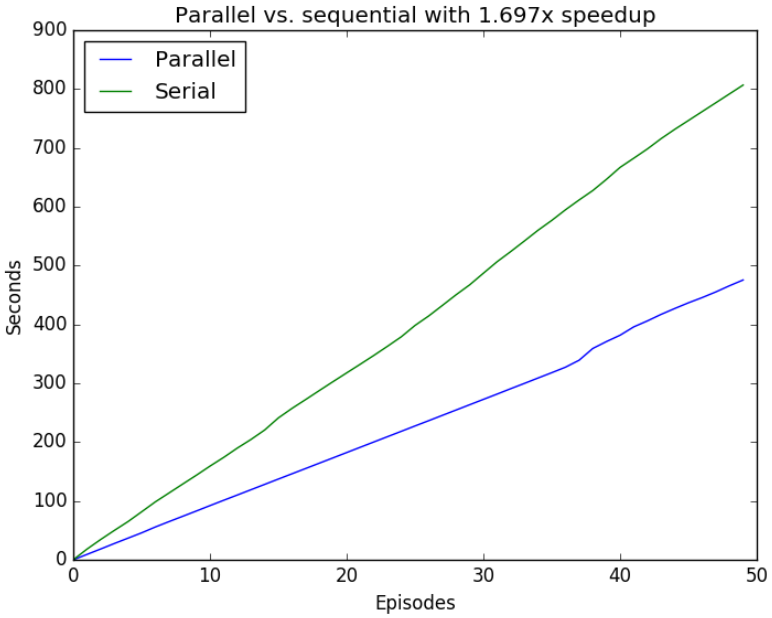


Figure 5.14: Parallelism speedup on Reacher2D environment with 300 rollouts. One vs two processors.

Using this knowledge we implemented a parallel procedure for sampling trajectories by instantiating multiple simulation environments and multiple copies of the policy network. In Figure 5.14 we depict the speedup of two processors compared to a single processor. It was generated on computer with 2 core Intel Core i5 with 2,4GHz. We have not achieved expected linear speed up which could be due to multiple reasons. In the one vs. two processors we trained on personal laptop whose processors were used by several other applications. This could take some of its computational power. Another part of computational time was taken by merging trajectories from different processes together.

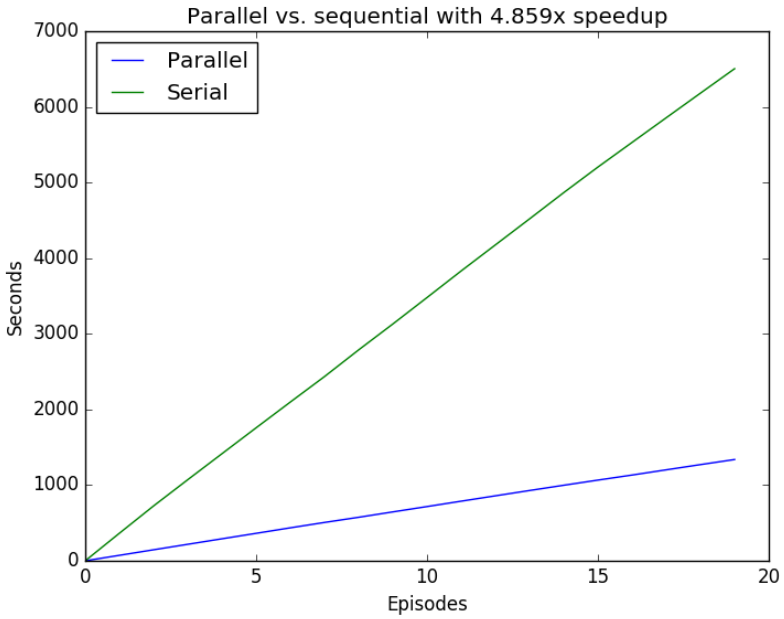


Figure 5.15: Sequential vs five processors.

Figure 5.16: Parallelism speedup on Grasper3D environment with 1000 rollouts per episode. One vs five processors.

In Figure 5.15 we compare one with five parallel processes. While generating it we used 6 core AMD Phenom II X6 1090T with 3.2GHz. The experiment with 5 cores seems much better. It is probably caused by the fact that we have 6 core processor and the last core can be used for computations required by other processes.

5.3.5 Summary

In this chapter we presented various environments for reaching and grasping. In table 5.1 we show DoF configurations for each environment. We showed that TRPO was able to converge into intended behaviour in each of them

Table 5.1: Hinge joint DoF constraints in each environment

		Reacher 2D	Reacher 3D	Wall Reacher	Grasper 3D
Arm	axis x	unlimited	unlimited	unlimited	unlimited
	axis y	-	unlimited	unlimited	unlimited
	axis z	-	-	-	-
Forearm	axis x	unlimited	unlimited	unlimited	unlimited
	axis y	-	unlimited	unlimited	unlimited
	axis z	-	-	-	unlimited
Grasper	left	-	-	-	(-40, 40)
	right	-	-	-	(-40, 40)

and found good policy. We thought our simulated robotic arm how to reach for objects and also how to grasp them.

We described multiple improvements for TRPO algorithm based on reusing an information from previous episodes. One of the methods was based on reusing previous search direction while the other reused previous trajectories for better estimation of the curvature. Both approaches led to convergence improvement over the standard TRPO algorithm in all three environments. We showed that it is possible to implement TRPO in a parallel manner which led to almost linear speedup with the number of cores.

Chapter 6

Implementation

In this chapter we briefly discuss how we chose the tool to work with. First we consider a simulator choice and then we go through some numerical computation frameworks. Then we present parallel implementation of TRPO algorithm.

6.1 Simulator

The first design decision was the choice of the simulator. Our requirements were that it have to be fast to allow quick training and development. As the first choice we were trying the *iCub* simulator where we are given control of child like robot (Tikhanoff et al., 2008). After some time playing with *iCub* we found that it contains lot of bugs and it often breaks down if there is an error in the physical simulation and therefore we decided not to use it.

The second platform we experimented with was *VREP* which is an acronym

```xml
<mujoco model="reacher">
    <compiler angle="radian" inertiafromgeom="true"/>
    <default>
        <joint armature="1" damping="1" limited="true"/>
        <geom contype="0" friction="1 0.1 0.1" rgba="0.7 0.7 0 1"/>
    </default>
    <option gravity="0 0 -9.81" integrator="RK4" timestep="0.01"/>
    <worldbody>
        <!-- Arena -->
        <geom conaffinity="0" contype="0" name="ground" pos="0 0 0" rgba="0.9 0.9 0.9 1" size="1 1 10" type="plane"/>
        <geom conaffinity="0" fromto="-.3 -.3 .01 .3 -.3 .01" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
        <geom conaffinity="0" fromto=" .3 -.3 .01 .3  .3 .01" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
        <geom conaffinity="0" fromto="-.3  .3 .01 .3  .3 .01" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
        <geom conaffinity="0" fromto="-.3 -.3 .01 -.3 .3 .01" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
        <!-- Arm -->
        <geom conaffinity="0" contype="0" fromto="0 0 0 0 0 0.02" rgba="0.9 0.4 0.6 1" size=".011" type="cylinder"/>
        <body name="body0" pos="0 0 .01">
            <geom fromto="0 0 0 0.1 0 0" name="link0" rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>
            <joint axis="0 0 1" limited="false" name="joint0" pos="0 0 0" type="hinge"/>
            <body name="body1" pos="0.1 0 0">
                <joint axis="0 0 1" limited="true" name="joint1" pos="0 0 0" range="-3.0 3.0" type="hinge"/>
                <geom fromto="0 0 0 0.1 0 0" name="link1" rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>
                <body name="fingertip" pos="0.11 0 0">
                    <geom contype="0" name="fingertip" pos="0 0 0" rgba="0.0 0.8 0.6 1" size=".01" type="sphere"/>
                </body>
            </body>
        </body>
        <!-- Target -->
        <body name="target" pos=".1 -.1 .01">
            <joint armature="0" axis="1 0 0" damping="0" limited="true" name="target_x" pos="0 0 0" range="-.27 .27"
                   ref=".1" stiffness="0" type="slide"/>
            <joint armature="0" axis="0 1 0" damping="0" limited="true" name="target_y" pos="0 0 0" range="-.27 .27"
                   ref="-.1" stiffness="0" type="slide"/>
            <geom conaffinity="0" contype="0" name="target" pos="0 0 0" rgba="0.9 0.2 0.2 1" size=".009" type="sphere"/>
        </body>
    </worldbody>
    <actuator>
        <motor ctrllimited="true" ctrlrange="-1.0 1.0" gear="200.0" joint="joint0"/>
        <motor ctrllimited="true" ctrlrange="-1.0 1.0" gear="200.0" joint="joint1"/>
    </actuator>
</mujoco>
```

Figure 6.1: 2D reacher environment in xml.

for "virtual robotic experimentation platform". It was quite easy to create or import new models and test them. However the main problem with this platform was its speed. After testing a few algorithms we have found that training under *VREP* environment takes too much time and we were also unable to start multiple sessions at once for parallelisation purposes.

To guarantee the high speed we needed to reach for implementations on lower level and find suitable physical engine simulator. We decided to test *MuJoCo* which was used in the most research papers we have mentioned in Chapter 4 and also in Schulman et al. (2015). The MuJoCo was already used in multiple environments in RL framework called *Gym*, we decided to give apply it to our problem. We used MuJoCo and its python bindings to extend *Gym* library with multiple environments for grasping and reaching.

Moreover, MuJoCo proved to be the fastest of all other choices and therefore we decided to exploit it in our experiments.

Specifying environment in MuJoCo is as easy as writing a simple and human readable XML, e.g. Code 6.1 which is specification of 2D reacher environment which is depicted in Figure 5.1b.

6.2 Mathematical computation library

Theano, *Tensorflow* and *Torch* are currently the most used and the most developed libraries for numerical computation. They all resemble at similar level of abstraction defining tensor (multidimensional array) structures and operations on them. These libraries use highly optimized C/CUDA/-Fortran code to reduce computational time as much as possible. At the start of this work *Torch* only offered Lua bindings, while our language of choice was python. Hence, we have not considered using Torch. Note that in 2017 *PyTorch* seems to be gaining big popularity in research community.

Both *Theano* and *Tensorflow* defined Tensor as base data structure on which they offer numerous operations from which computational graphs are built. Each operation have its derivation in the automatic differentiation module, which in use with the chain rule allows computing gradients for each node in such graphs. While *Theano* was older and much more developed, we decided to stick with *Tensorflow* due to its nicer API and better documentation.

There also exist higher level libraries which abstract common operations

such as creation of neural networks layers or whole networks such as *Keras*, *TFLearn* or *TFSlim* with *Tensorflow* backend or *Lasagne*, *Blocks* and *Keras* with *Theano* backend. We decided to use only lower level API since we were already familiar with it.

6.3 Parallel TRPO

A GIL (global interpreter lock) is an interpreter level lock which prevents multiple threads from running at the same time within the same interpreter. Each thread that wants to execute an action must first wait for GIL to be released by other threads. Hence, using python multithreading module would not really help our algorithm to run any faster. However, python still offers a way to make our algorithm parallel by using quite expressive multiprocessing API for creating new processes and communicating between them using pipes and queues. In our implementations we extended Process class and used Queues for sending jobs to processes and another queue for receiving results from workers.

6.3.1 Architecture

To speed up the most time consuming part of our algorithm, computing trajectories, we decided to create multiple workers where each worker reads from job queue where the main process put requests to receive new trajectory. After workers asynchronously sample rollouts, they write them to result queue where the main process collects them and performs the whole policy update. After having computed the new parameters $\boldsymbol{\theta}$ for the policy, the main pro-

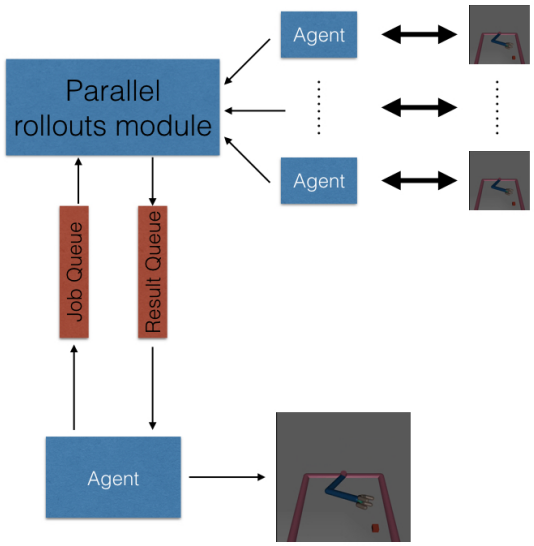


Figure 6.2: Parallel architecture design.

cess distributes them to each worker, so sampling can continue under the new policy. We repeat the process until we are satisfied with a policy and terminate the program. The whole process is depicted in Figure 6.2.

Chapter 7

Conclusion

In this work we described an approach for solving robotic reaching and grasping by using state-of-the-art reinforcement learning (RL) algorithm called trust region policy optimization (TRPO). We described basic theoretical foundations of reinforcement learning and the necessary theory for policy gradient algorithms. Then we explained policy gradient improvements such as natural policy gradients and our method of choice - TRPO. We presented several state-of-the-art RL algorithms. We showed that TRPO can be successfully applied to a simulated robotic environments such as 2D reaching and 3D reaching and grasping. In the 3D reacher environment we also added wall which made the task harder. We showed that with suitable reward TRPO is capable of finding a good policy for target reaching while avoiding collisions. We proposed two improvements for TRPO algorithm. The first one reused previous trajectories for better estimation of the parameter space curvature while the second one reused previous gradient information. Both of these upgrades improved convergence rate and decreased a variance

of TRPO in 2D, 3D reaching and 3D grasping environments. Experimental results showed that our parallel TRPO implementation was able to achieve almost linear speedup with up to 5 cores, but we were not able to test its limit due to hardware constraints. Last but not least, we described implementation choices related to tools, libraries and the simulator. We showed how to build a simple environment in MuJoCo physical simulator and we also described implementation details for TRPO parallelisation.

There definitely could be done more development in current framework by applying TRPO to visual based reaching and grasping and further research can be conducted on why the application of proposed methods for convergence improvement did not work together.

Bibliography

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural Computation*, 10(2), 251–276.

Bohg, J., Morales, A., Asfour, T., & Kragic, D. (2014). Data-driven grasp synthesis - A survey. *IEEE Transactions on Robotics*, 30(2), 289–309.

Castellini, C., Orabona, F., Metta, G., & Sandini, G. (2007). Internal models of reaching and grasping. *Advanced Robotics*, 21(13), 1545–1564.

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*.

Gu, S., Lillicrap, T. P., Sutskever, I., & Levine, S. (2016). Continuous deep Q-learning with model-based acceleration. In M.-F. Balcan and K. Q. Weinberger, editors, *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2829–2838.

Haykin, S. (2009). *Neural Networks and Learning Machines*. Neural networks and learning machines. Prentice Hall, 3 edition.

Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep net-

work training by reducing internal covariate shift. *Computing Research Repository*, abs/1502.03167.

Joseph Redmon, A. A. (2015). Real-time grasp detection using convolutional neural networks. In *IEEE International Conference on Robotics and Automation, (ICRA 2015)*, pages 1316–1322. IEEE.

Kakade, S. (2001). A natural policy gradient. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14 (NIPS)*, pages 1531–1538. MIT Press.

Kakade, S. & Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 267–274, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Kalakrishnan, R. et al. (2011). Learning force control policies for compliant manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Kraft, D., Detry, R., Pugeault, N., Başeski, E., Guerin, F., Piater, J., & Krüger, N. (2010). Development of object and grasping knowledge by robot exploration. *IEEE Transactions on Autonomous Mental Development*, 2(4), 368–383.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114.

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

Levine, S. & Koltun, V. (2013). Guided policy search. In *International Conference on Machine Learning (ICML 2013)*.

Levine, S., Pastor, P., Krizhevsky, A., & Quillen, D. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *Computing Research Repository*, abs/1603.02199.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *Computing Research Repository*, abs/1509.02971.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3), 293–321.

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, pages 735–742.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *Computing Research Repository*, abs/1602.01783.

Pascanu, R. & Bengio, Y. (2014). Revisiting natural gradient for deep networks. In *International Conference on Learning Representations*.

Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Comput.*, 6(1), 147–160.

Saxena, A., Driemeyer, J., & Ng, A. Y. (2008). Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2), 157–173.

Schulman, J., Moritz, P., et al. (2015). Trust region policy optimization. In *International Conference on Machine Learning (ICML)*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. A. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, (ICML 2014)*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 387–395.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Tikhanoff, V., Cangelosi, A., Fitzpatrick, P., Metta, G., Natale, L., & Nori, F. (2008). An open-source simulator for cognitive robotics research: The prototype of the icub humanoid robot simulator. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems, (PerMIS 2008)*, pages 57–61.

van Hasselt, H. & Wiering, M. A. (2009). Using continuous action spaces to solve discrete problems. *International Joint Conference on Neural Networks*, 00, 1149–1156.

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep reinforcement learning with double Q-learning. *Computing Research Repository*, abs/1509.06461.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1995–2003.

Zdechovan, L. (2012). *Modelovanie uchopovania objektov pomocou neurónových sití v robotickom simulátore iCub*. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava.

Zhang, F., Leitner, J., Milford, M., Upcroft, B., & Corke, P. (2015). Towards vision-based deep reinforcement learning for robotic motion control. In *Australasian Conference on Robotics and Automation 2015*, Canberra, A.C.T.

List of Figures