Norwegian University of
Science and Technology

Reinforcement Learning For Control of Robotic Manipulators

Specialisation Project Fall 2018

Arild Vermedal

Norwegian University of Science and Technology
Department of Engineering Cybernetics (ITK)

Supervisor: Associate Professor Anastasios Lekkas, ITK

January 15, 2019

Preface

This report is written as part of my specialization project in TTK4550 at the Norwegian University of Science and Technology (NTNU) during the fall of 2018. The topic studied is at the behest of the Department of Engineering Cybernetics at NTNU.

I would like to give a special thanks to my supervisor Associate Professor Anastasios Lekkas at the Department of Engineering Cybernetics for his thorough and valuable guidance throughout the semester.

Arild Vermedal
January 15, 2019

Abstract

Artificial intelligence have progressed immensely the past decade and is starting to be used for control of complex systems. This project concerns itself with the application of reinforcement learning to the task of controlling robotic manipulators. A history of the control of robotic manipulators is presented, followed by an introduction to increasingly advanced reinforcement learning theory and methods. The methods are applied to simple textbook problems to show their performance. Lastly the project uses the recent Proximal Policy Optimization method to control a simulated robotic manipulator performing various tasks, showing both how it can quickly learn a simple task, and requires more than a direct application for more complex tasks.

Contents

Abbreviations

Computed Torque Method	CTM
Deep Deterministic Policy Gradient	DDPG
Deep Q-Learning	DQL
Guided Policy Search	GPS
Linear-Quadratic Controller	LQR
Makrov Decision Process	MDP
Model Predictive Control	MPC
Neural Network	NN
Nonlinear Model Predictive Controller	NMPC
Normalized Advantage Functions	NAF
Policy Iteration	PI
Proximal Policy Optimization	PPO
Rectified Linear Units	ReLu
Reinforcement Learning	RL
Robotic Manipulator	RM
State-Action-Reward-State-Action	SARSA
Trust Region Policy Optimization	TRPO
Value Iteration	VI

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Artificial intelligence as a tool for control of physical systems is a large topic in research, and new methods and techniques are developed continuously. This project focuses on the application of reinforcement learning for control of robotic manipulators, to improve performance over classical control and enable the execution of new categories of tasks. A brief introduction to the history of robotic manipulators and the theory behind reinforcement learning is presented, before progressively more modern methods are applied to progressively more complex problems. The goal of this project is to go through the fundamentals of reinforcement learning and explore the capability of state-of-the-art reinforcement learning for the purpose of controlling a robotic manipulator.

Section 1.1 will succinctly describe the development of the robotic manipulator throughout the past decades, before Section 1.2 introduces reinforcement learning in the context of controlling a robotic manipulator.

1.1 History of Robotic Manipulator Control

A robotic manipulator (RM) is a subset of the category of industrial robots. RM may have multiple interpretations and configurations, but will here be defined as an open kinematic chain of rigid bodies interconnected by joints with a specialized tool at the end, as described by Siciliano (2009). RMs have been in development for a long time, and it can be unclear exactly when something we would today call an RM came to be. The precursor to the early RM were the teleoperated systems, where a slave machine copied the movements performed by a human-operated master device, later including force feedback to the operator (Spong, 1989).

A brief overview of the development of the control of RMs through the 60s is given by Paul (1981), starting with George Devols' robot arm. This device have little claim to the term *robot* as it is understood today. Instead it was guided through a series of motions, which it stored and was then able to repeatably play back. Nevertheless, this simple repeated motions machine was useful for similarly simple tasks, and the machine could be reprogrammed to perform a new task when needed.

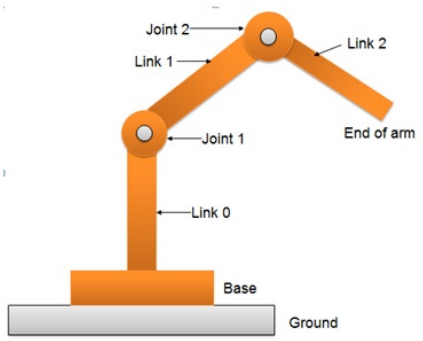

Figure 1.1: The basic shape of an RM. Picture: nptel

Only a year later, in 1961 at MIT, another RM was created, using touch sensors to detect when it had run into something, which would activate a preprogrammed response. The technology remained dependent on predetermined, humanly added knowledge of the environment. It did however not rely on absolute movements, and was perhaps the first step towards programming goals and letting the machine determine when they were accomplished. It was also in the 60s that the first real steps in computer vision were taken, and the position and orientation of known, simple objects could be determined from images in real time (Wichman, 1967).

1.1.1 1970–80

In the 1970s, more advanced methods for control were being researched, while most industrial robots in use remained either teleoperated by humans, or copying a set of movements with simple, slow control methods to guide them (Dubowsky and DesForges, 1979). One of the focuses of research were methods of specifying the end of the movement in Cartesian coordinates, and have them translated into joint positions and realized. Multiple approaches were used, such as solving complicated polynomials for planning and using a simple PID-like controller to adjust the exact inputs (Paul, 1972). In comparison, past methods would linearize the system and neglect external loads (Dubowsky and DesForges, 1979). Adaptive methods that would not require exact knowledge of the parameters of the RM were being developed. They could estimate both their parameters and the mass properties of an external load (Paul, 1981). Still, they were based on linearization of the system and simplifying the load to a simple mass attached to the end. Landau (1974) compiles a survey of the state of the art research on model reference adaptive techniques, summarizing that adaptive methods remain limited by the available computing power and detail of the modelled dynamics of the system.

1.1.2 1980–90

The works of Uchiyama (1989) details the state of the art of control of robot arms by that time, split into two main categories: motion control and force control. Motion control is based on

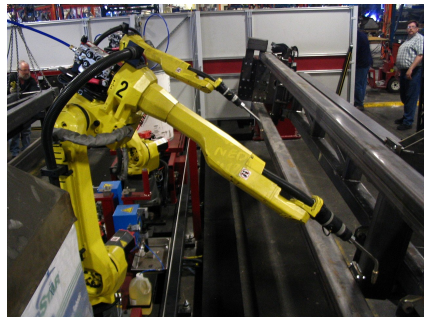

Figure 1.2: A set of six-axis robots used for welding. Picture Wikipedia (b)

the control of the absolute position of the end-effector, split again into model-based control and feedback control. Model-based control was a recent innovation for practical use, as it was only possible with a fast and powerful computer. These methods also required a detailed model of the system. The payout for these high requirements were faster and more tightly-controlled motions than feedback methods could achieve. Feedback control is the evolution of the control methods of the 70s and includes schemes like adaptive control methods. Feedback control generally makes use of less detailed models, incorporating only the basic setup of the system while disregarding the need to know the exact parameters. The drawback were slower movements and less ability to quickly answer to new movement commands. In contrast to motion control methods, force control methods were based on more direct interaction with the environment. The use of force and torque sensors would detect collisions and exertions of force such as acceleration. These methods rely on an accurate description of the system and the task to be performed when used in a computer-operated system (Whitney, 1987). Still, most industrial robots depended on human expertise to perform the planning and coordination, which was then stored for later playback (Lewis, 2004). Force control was a useful tool for creating teleoperated robots (Uchiyama, 1989). Two important pieces of research were the Computed Torque Method (CTM) (Luh et al., 1980) and the discovery that "the conventional PD feedback servo-loop assures the globally asymptotic stability for point-to-point position control (setpoint control) if the gravity force is carefully compensated.",as stated by Arimoto (1994) (Takegaki and Arimoto, 1981).

1.1.3 1990–2000

Model Predictive Control (MPC) was a major innovation of model-based control that became very popular for a significant number of processes, but less so for the control of RMs (Allgöwer et al., 1999). The Nonlinear Model Predictive Controller (NMPC) however, was found to be suitable for point to point control and showed high efficiency of movement at the cost of significant computation costs. It required detailed knowledge of the system, and manual tuning of costs (reward function equivalent) (Poignet and Gautier, 2000). Force control continued through two main paths: impedance control and hybrid control. Yoshikawa (2000) surveys the state of the art of force control by that time. There is little conceptual difference to the methods of the previous decade, though advances in the theory allowed for a greater number of possible applications.

While there have been advances in the control of RMs outside the field of AI since 2000, they are less interesting with regards to understanding the past of RM control. Methods such as CTM, MPC, NMPC and linear-quadratic controller (LQR) remain at the forefront of robot control. The main improvements are in more powerful computers and efficient calculations (Ajwad et al., 2014), (Sciavicco and Siciliano, 2012).

1.2 Robotic Manipulator Control Using Reinforcement Learning: State of the Art

Reinforcement learning (RL) is a subset of machine learning focused on an agent exploring an environment and determining a policy for performing actions dependent on the state that maximizes some give reward. RL can in some ways be framed as an evolution of the adaptive optimal control methods mentioned earlier (Sutton et al., 1992). In terms of control theory, the policy to be optimized is the controller, the environment is the plant, the reward is the cost function, the state is the and the action is the control vector. One of the most significant differences between classical control and RL methods are the latter's ability to function without a priori information of the system model. RL methods can be completely model free, or they can incorporate a model estimate either supplied by the creator and adapted by the algorithm or completely self taught (Li, 2017). The general version of an RL method as a function of the state s and action a taken to reach that state. The policy π is the mapping of states to actions with the purpose of optimizing a sum of rewards $R(s, a)$. The goal of any RL method is to revise this policy to reach a "good" or, when feasible, optimal policy. Thus, the sum of rewards is maximized.

Kober et al. (2013). RL is distinguished from other forms of AI learning in the exploration of the environment. An RL based agent is not told what action it should take, or should have taken, but instead is given only a reward based on the action taken. Thus, there is no information gained about the reward from unexplored options, and the addition of uncertainty means that even previously taken actions are not necessarily fully known. Actions may also have long reaching consequences not immediately obvious.

The current field of RL is wide and varied, with some approaches being more applicable for RM than others. Kober et al. (2013) divides these methods into two groups: value-functions approaches, split into dynamic programming methods, Monte Carlo methods and temporal difference methods, and policy search. A significant amount of past research have been based on value-function approaches. These are however less suitable for RM due to factors such as a continuous state and action space and high dimensionality. Some approaches leverages the best of both methods with what is called actor-critic methods, where the value function, the critic, is used only for policy updates, and not for action selection. A brief look at value-function methods includes State–Action–Reward–State–Action (SARSA), the Q-learning method, its deep reinforcement learning variant deep Q-learning (DQL) (Mnih et al., 2015) and an adaption for continuous action space, Normalized Advantage Functions (NAF) (Gu et al., 2016). Recent policy search methods include guided policy search (GPS) (Levine and Koltun, 2013), deep deterministic policy gradient (DDPG) (Lillicrap et al., 2015) and trust region policy optimization (TRPO) (Schulman et al., 2015). Some methods are based on the stability analysis of Lyapunov functions to create methods safer for physical systems (Chow et al., 2018). Many methods for RL are "deep" methods, methods with one or more hidden layers between the

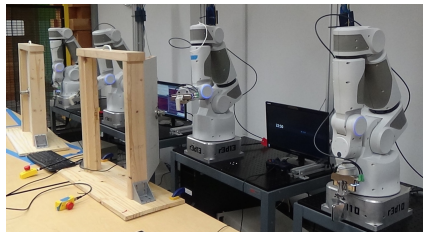

Figure 1.3: A series of RMs learning to open a door, sharing their experience for a faster learning curve. Picture: Yahya et al.

input and the output, such as a neural network. Li (2017) describes this and attempts to explain why that is the case: "Why has deep learning been helping reinforcement learning make so many and so enormous achievements? Representation learning with deep learning enables automatic feature engineering and end to-end learning through gradient descent, so that reliance on domain knowledge is significantly reduced or even removed. Feature engineering used to be done manually and is usually time consuming, over-specified, and incomplete. Deep, distributed representations exploit the hierarchical composition of factors in data to combat the exponential challenges of the curse of dimensionality. Generality, expressiveness and flexibility of deep neural networks make some tasks easier or possible".

A key motivation for this work is the question of why RL is necessary. To say that RL is better than classical control, or will replace it in the foreseeable future would be questionable. There are many problems and applications where the control methods of today are reliable, cheap and effective. In a known environment the current industry standards certainly perform great work. Both robotic control and AI control have greatly evolved over the previous decades, allowing new challenges to be addressed in both fields. The challenge RL seeks to overcome, that today's classical control cannot, is that of flexibility, of adaptability. Many tasks require a human hand simply because there are no current methods that can tackle the amount of varying situations that can be encountered. With the last years' explosion of available data and new techniques, RL is now in a place where these problems can be faced, and hopefully solved. With this in mind, RL is headed towards becoming a useful tool for control of RMs, and more and more combined methods employing both RL and classical control are likely to surface in research in the near future.

Chapter 2

Theory

This chapter provides an introduction to the theory behind RL and a select few methods. Section 2.1 covers the fundamental theory RL is based on, and several introductory methods incorporating various key aspects. Section 2.2 details more modern techniques and the workings of a modern method making use of then.

2.1 The Basics of Reinforcement Learning

RL is a wide topic, and this project will only concern itself with giving an introductory explanation. Subsection 2.1.1 will cover the basic theory behind all of RL, and Subsection 2.1.2 describes several early methods demonstrating elements of the theory.

2.1.1 Fundamental Theory

Reinforcement learning deals with solving problems of sequential decisions with limited feedback. Contrasted with supervised learning, where the agent is told the correct action it should have taken, and unsupervised learning, where the agent receives no feedback at all, RL agents receive feedback in the form of a scalar reward. The goal of the agent is to maximize the cumulative reward. The agent does this by calculating or learning a policy of what action to take in each possible state. To go further, some notation is required.

A Markov decision process (MDP) is a fundamental tool in RL. Any problem to be solved is almost always stated as an MDP. An MDP consists of states, actions, transitions and rewards. The set of states S is a finite set of all the possible states of the system. A state characterizes all parameters of the problem that are relevant for the agent. The set of actions A is the finite set of all possible actions. Some actions may not be possible in certain states. The transition function $T(s, a, s')$ describes the chance for ending up in a new state s' after preforming action a in state s. It is

a mapping from $S \times A \times S \rightarrow [0, 1]$. Lastly the reward function R denotes the reward of an outcome. It may be dependent only on the state reached, $R : S \rightarrow \mathbb{R}$, or the path taken to reach it, $R : S \times A \times S \rightarrow \mathbb{R}$. The notation is usually simplified for a problem to only specify the parameters that affects the outcome. MDPs also have the Markov Property, which states that the next state is only dependent on the current state. That is, the past is irrelevant; only the current state and the action taken affect the outcome of each state transition.

As stated in Section 1.2, the goal of the RL agent is to maximize the received reward. With the notation of the MDP, we can formulated that reward as $\Sigma_{i=1}^{h} r_i$. Usually we want to discount the value of future rewards with a factor $\gamma \in [0, 1]$ and use $\Sigma_{i=1}^{\inf} \gamma^{i-1} r_i$. Lastly, the average reward variant $\frac{1}{h}\Sigma_{i=1}^{h} r_i$ may also be used. Given the MDP $\{S, A, T, R\}$ the agent attempts to learn a policy $\pi : S \rightarrow A$. There also exists a stochastic variant $\pi(s, a) \rightarrow [0, 1]$. Often, the problem the RL agents are exploring is a stochastic problem, and the exact outcome of an action is not certain. In this case it is useful to consider the expected reward. In the deterministic case the expected reward is the exact reward, so the same notation may be used for both. Since we want to maximize the cumulative reward, we must also take into account the expected future rewards. With this we can assign each state s an expected value if we follow a policy π, defining the value function $V^{\pi}(s)$. Here, the discounted infinite horizon is used as the model for cumulative rewards. Equations (2.1) and the similar state-action value function (also known as the Q-function), Equations (2.2), which additionally specifies the first action taken before following the policy afterwards, describes this expected value.

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{i=0}^{\inf} \gamma^i r_{i+1}\right] \quad \forall s \in S. \tag{2.1}$$

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{i=0}^{\inf} \gamma^i r_{i+1}\right] \quad \forall s \in S, \forall a \in A. \tag{2.2}$$

Equation (2.3), the Bellman equation, gives a recursive definition the value function, creating a set of nonlinear equations:

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(a), s')\Big(R(s, \pi(a), s') + \gamma V^{\pi}(s')\Big) \quad \forall s \in S. \tag{2.3}$$

The Bellman equation describes the value when following any policy. An optimal policy π^* will fulfill $V^{\pi^*}(s) \geq V^{\pi}(s) \quad \forall s \in S, \forall \pi$. For simplicity V^{π^*} is denoted V^*. The Bellman optimality equation (2.4) gives an expression for this optimal value function, which can also be phrased in terms of the Q-function as equation (2.5).

$$V^*(s) = \max_a \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma V^*(s')\Big) \quad \forall s \in S, \tag{2.4}$$

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma \max_{a'} Q^*(s', a')\Big) \quad \forall s \in S, \forall a \in A. \tag{2.5}$$

The optimal policy may be extracted from the value function or the Q-function using (2.6) or (2.7) respectively.

$$\pi(s) = \underset{a'}{\operatorname{argmax}} \sum_{s' \in S} T(s, a, s') \Big(R(s, a, s') + \gamma V^(s') \Big) \quad \forall s \in S, . \tag{2.6}$$

$$\pi(s) = \underset{a'}{\operatorname{argmax}}\, Q(s, a) \quad \forall s \in S, . \tag{2.7}$$

2.1.2 Introductory Methods in RL

The following subsections will describe a selection of basic methods for solving textbook RL problems.. At first, perfect knowledge is assumed in Policy Iteration (PI) and Value Iteration (VI), then exploration is added with Q-learning, and continuous states with REINFORCE.

2.1.2.1 Policy and Value Iteration

The calculation of the optimal value function, and so the optimal policy, is a set of non-linear equations. This is complicated and time consuming to solve analytically. Instead, it can be shown that iteratively computing the value function using the Bellman Equation as an update rule, as done in equation (2.8), will converge to the optimal value function. Because convergence happens in the infinite horizon, the update is usually stopped when the difference between the old value and the new value is below some margin σ.

$$V_{k+1}(s) = \sum_{s' \in S} T(s, a, s') \Big(R(s, a, s') + \gamma V_k(s') \Big). \tag{2.8}$$

PI starts with some arbitrary policy π_0, and calculated the value function V^{π_0} using the method described above (policy evaluation). Then it uses this value function to compute an improved policy π_1 (policy improvement), which in turn is used to calculated V^{π_1} and so on. This continues until there is no change in policy from one iteration and the next. This policy must then be the optimal policy, as the corresponding value function fulfills the Bellman optimality equation. Algorithm 1 is the pseudocode for performing this calculation in two alternating functions until termination.

Algorithm 1 Policy Iteration

Initialization
$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathbb{R}$ arbitrarily for all $s \in S$

1. Policy evaluation:
while $\Delta > \sigma$ **do**
  $\Delta \longleftarrow 0$
  for $s \in S$ **do**
    $v \longleftarrow V(s)$
    $V(s) \longleftarrow \sum_{s'}[T(s, \pi(s), s')\Big(R(s, \pi(s), s') + \gamma V(s')\Big)]$
    $\Delta \longleftarrow max(\Delta, |v - V(s)|)$
  end
end
2. Policy Improvement:
while *not stable* **do**
  stable = True
  for $s \in S$ **do**
    $\pi_{old} = \pi(s)$
    $\pi(s) \longleftarrow \underset{a}{\operatorname{argmax}} \sum_{s'}[T(s, a, s')\Big(R(s, \pi(s), a) + \gamma V(s')\Big)]$
    if $\pi_{old} \neq \pi(s)$ **then**
      stable = False
    end
  end
  if *not stable* **then**
    preform Policy evaluation()
  end
end
Pseudocode from Wiering and Otterlo (2012)

The drawback of PI is that the policy evaluation is repeated multiple times. An alternate method of calculating the optimal policy is to instead combine the action of policy evaluation and policy improvement to 1 step and repeat that until the value function converges. Then the optimal policy can be extracted from the value function at the end. Pseudocode for the VI method is shown in Algorithm 2.

Algorithm 2 Value Iteration

Initialization
$V(s) \in \mathbb{R}$ arbitrarily for all $s \in S$ except $V(terminal) = 0$

while $\Delta > \sigma$ **do**
  $\Delta \longleftarrow 0$
  for $s \in S$ **do**
    $v \longleftarrow V(s)$
    $V(s) \longleftarrow \max_a \sum_{s'} [T(s,a,s')\Big(R(s,a,s') + \gamma V(s')\Big)]$
    $\Delta \longleftarrow max(\Delta, |v - V(s)|)$
  end
end
for $s \in S$ **do**
  $\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s', r|s, \pi(s))[r + \gamma V(s')]$
end
Pseudocode from Wiering and Otterlo (2012)

2.1.2.2 Q-learning

PI and VI both lack a key part of the concept of RL: exploration. The requirement of knowing the full model is one that RL attempts to circumvent, by taking actions and using the feedback received to understand the system. Some methods use this to model the transition function, while others, like Q-learning, use it in a model-free way to directly choose actions. Unlike PI and VI, Q-learning interacts with the environment while learning the policy. Exploring methods are split in two categories: on-policy and off-policy. On-policy methods incorporate exploration and exploitation as one policy, while off-policy methods can explore with a different strategy than what it consider optimal for exploitation. Q-learning have a lot in common with on-policy methods, but is considered off-policy as it updates the Q function based on the greedy policy, but may take a random, non-greedy action. That is, it does not exactly follow the policy it is creating.

Q-learning agents updates the value of a state-action pair (s, a) based on the result of taking action a while in state s, similar to the update rule of PI and VI, but bound by having to actually perform the action before learning the reward of the outcome. It will in some way chose an action, and update the Q function based on update rule in (2.9). A learning rate α decides how much importance should be given to the new information compared to the previously learned value. In a deterministic environment $\alpha = 1$ will cause the Q-function to quickly reach the true value. Q-learning can be shown to converge as long as the method of choosing the action leads to every state being visited an infinite amount of times. The most common method of choosing the action is ϵ-greedy, which will choose the greedy option based on the already learned Q-function, with a chance $\epsilon \in [0, 1]$ of choosing a random action. The chance often starts high, and is then lowered over time as the Q-function approaches the true values.

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)\Big). \tag{2.9}$$

Algorithm 3 Q-learning

Require discount factor γ, learning rate α Initialize Q arbitrarily
for *each episode* **do**
  s is initialized as the starting state
  while *s is not a goal state* **do**
    choose an action $a \in A(s)$ based on Q and an exploration strategy
    perform action a
    observe the new state s' and received reward r
    $Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a' \in A(s')} Q_k(s',a') - Q(s,a)\Big)$
    $s := s'$
  end
end
Pseudocode from Wiering and Otterlo (2012)

2.1.2.3 REINFORCE

All of the described methods have so far been discrete, tabular value function methods. These form an important base for the topic of RL and the understanding of the state value functions and state-action value functions. However, their discrete nature makes them less applicable to the task of controlling an RM. This brings us to REINFORCE, a policy gradient method. REINFORCE is a stochastic, continuous method based on creating a parameterized policy directly, without the use of a value function. The policy has a parameter vector called $\boldsymbol{\theta}$ giving the notation $\pi(a|s,\boldsymbol{\theta}) = Pr(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$ for the probability of selecting action a at time t while in state s if the parameter is $\boldsymbol{\theta}$. The second new notation is $J(\boldsymbol{\theta})$, denoting a scalar performance measure used to update the policy parameter vector. Policy gradient methods are called so because they seek to use the gradient of the performance measure to update the policy parameter vector through gradient accent. The general update rule is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{2.10}$$

where $\widehat{\Delta J(\boldsymbol{\theta}_t)}$ is an estimate whose expectation approximates the gradient of $\boldsymbol{\theta}$.

In particular, the update rule of REINFORCE is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla ln\pi(A_t|S_t,\boldsymbol{\theta}), \tag{2.11}$$

where G is the return given by

$$G \leftarrow \sum_{k=t+a}^{T} \gamma^{k-t-1} R_k, \tag{2.12}$$

and t iterates through each step of the episode, from 0 to T-1. This update is performed at the end of each episode, meaning the policy parameter and thus the policy remains unchanged during

each episode. REINFORCE has good convergence properties in theory, but often ends up with high variance.

2.2 Modern RL

In the last decade the field of RL has experienced a huge growth, with new supporting techniques and methods. This section will cover a portion of this relevant to the control of an RM.

2.2.1 Deep Learning and Neural Networks

A neural network (NN) is at its most simple a function approximator, taking some input and transforming it to some output. The least complex form of an NN is a perceptron, developed in the 50s (Rosenblatt, 1958). A perceptron takes a number of inputs x_1 to x_n each multiplied with a corresponding weight w_1 to w_2 as well as a scalar bias b to produce a binary output. An NN is a network of perceptron stacked in parallel, then layered so that each stack of perceptrons take input from the previous layer (or the function input for the first layer) and provides its results as inputs from the next input (or as the function output from the last layer). The layers between the input layer and the output layers are referred to as hidden layers. A modern NN may use several different non-binary activation methods for deciding the output of a perceptron such as Sigmoid, Softmax or Rectified linear units (ReLu). In common most activation methods is the limitation that the output is a scalar $\in [0, 1]$ or $\in [-1, 1]$. These have different effects on learning rate and are fit for different applications. Sometimes a mix of different functions can be effective, and the choice of activation function is a mayor part of designing a NN. One key benefit of these common activation methods are that they introduce non-linearity, allowing the NN to represent non-linear functions.

When training an NN, the parts of the system that changes are the weights. This is done through backpropagation with the use of a loss function. A loss function is a measure of how far off target the output was for any given input. This is a whole topic on its own, and a large part of each RL method using an NN is the design of this function. The important part is that as this function gives some form of measure of how close an output is to being correct it can be used improve the NN. This is done using the gradient of the loss function and updating each weight based on this gradient through backpropagation. It uses the chain rule to update each layer in turn, starting with the weights between the last hidden layer and the output, *propagating backwards* through each layer.

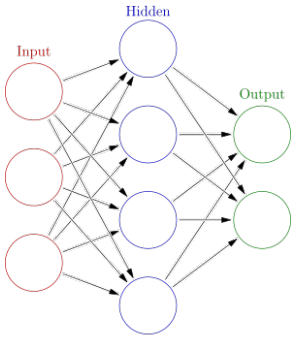

Figure 2.1: An NN with one hidden layer taking three scalar inputs, providing two scalar outputs. Picture: Wikipedia (a)

2.2.2 Trust Region

Trust region is a term for the area around the current search point where the approximation, generally the current function approximation, is trusted to be correct. This is used when updating an approximation to avoid moving too far in any direction, as the uncertainty of the approximation outside the trust region means that while it looks like an improvement, it could end up being worse. When an update would case a value to move outside the trust region, it will instead be stopped at the boundary.

2.2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) proposed in Schulman et al. (2017) has become hugely popular for its good convergence and high performance. It is an on-policy method suited for complex continuous state- and action-spaces; exactly what is needed to control an RM. PPO is a recent method, published just over 1 year ago, and already widely well regarded in research for its potential applications. PPO is often implemented as an actor-critic method, defining an value function to be trained together with the policy function. This is used to create a loss function $L^{VF} = (V_\theta(s_t) - V_t^{target})^2$. When evaluating a new policy it is compared to the old policy, so $R_t(\theta)$ defines the ratio between the new policy and the old so that a measure for the improvement L^{CPI} is defined as

$$L^{CPI}(\theta) = \hat{\mathrm{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t}\hat{\mathrm{A}}_t\right] = \hat{\mathrm{E}}\left[r_t(\theta)\hat{\mathrm{A}}_t\right], \tag{2.13}$$

where $\hat{\mathrm{A}}$ is the estimated advantage function. Maximizing this is stated to lead to excessively large policy updates, so the method proposes several different ways of dealing with this issue. One version is to use trust regions to set restrictions for the new policy, another to adapt the strict restrictions to instead be a penalty, but the main, simplest and most promising trick is to use clipping on this function, but only when the change is too positive, creating

$$L^{CLIP}(\theta) = \hat{\mathrm{E}}\left[\min(r_t(\theta), \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{\mathrm{A}}_t\right)], \tag{2.14}$$

with ϵ being a hyperparameter for the method. The complete function for optimizing the policy then becomes

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathrm{E}}\left[L^{CPI}(\theta) - c_1 L_t^{VF} + c_2 S[\pi_\theta](s_t)\right], \tag{2.15}$$

where c_1 and c_2 are hyperparameters and S an entropy bonus.

The method runs through a number of iterations where N actors each interacts with an environment for T timesteps, creating $N * T$ datapoints. These are then used to optimize the policy, before a new batch runs using the new policy, repeating until the agent is finished or simulation stopped. The process is shown in Algorithm 4.

Algorithm 4 PPO, Actor-Critic Style

for *iteration=1,2,...* **do**
  for *actor=1,2,...,N* **do**
    Run policy $\pi_{\theta old}$
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end
  Optimize surrogate L wrt θ, With K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end

Pseudocode from Schulman et al. (2017)

The result is a method that avoids overly optimistic adjustments in order to avoid the inaccuracies introduced be moving further away from the old policy, and thus the parameters under which the current data was gathered.

Chapter 3

Implementation

This chapter will go through the specifics of how each method was implemented, what tools where used and which decisions were made.

3.1 Tools

Each method was implemented using Python version 3.6.1 using PyCharm community edition as IDE. The Scipy and Numpy libraries were used for calculations on vectors and matrices, and TensorFlow and Keras were used for the NNs. Open AI: gym were used for it's implementations of many classical problems in RL, and the RM used for the manipulator problems. The environment provides the number of states, the number of actions and a list of each possible outcome of action a taken from state s. Additionally the implementation offers the ability to take actions and receive feedback about the outcome.

3.2 FrozenLake

The FrozenLake problem is a classic introductory problem in reinforcement learning. An agent has to navigate a slippery lake, avoiding falling into a hole and reaching the goal. The problem is episodic, stochastic, stationary and discrete. The lake is modeled as a 4x4 grid, for a total of 16 possible states. Some states are holes, giving a negative rewards and ending the episode, and one state is the goal state, giving a positive reward and ending the episode. All other states give zero rewards. In each non-terminal state there are four available actions, moving up, down, right or left. When taking an action, the agent has a 33% chance of moving as intended, and 33% to move to either side instead. It will never move in the opposite direction of the movement chosen. Moving into a wall leaves the agent in the previous state. As will be seen later, this slippery surface leads to some unintuitive policies. Perfect information about the transition function is available, allowing

methods such as PI and VI to solve the problem. The agent can also be denied this information, and be given only the result of each action as feedback.

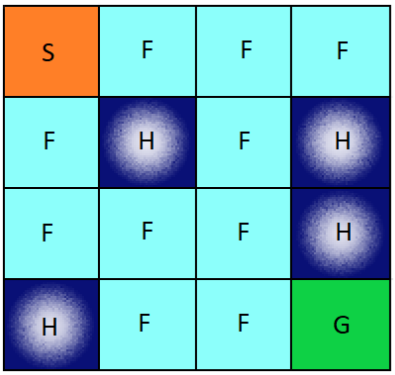

Figure 3.1: The 4x4 grid of the frozen lake problem. Picture: Analytics India Magazine

3.2.1 Policy and Value Iteration

There are three main ways to affect the policy iteration algorithm: the initial policy π_0, the initial value function $V_0(s)$ and the discount factor γ. The effect of γ is how much value is placed on future reward. There are three values that are interesting to look at the effects of: 0, <0,1> and 1. The effect of the initial policy and the initial value function is the direction of the initial search, and thus the number of iterations needed. This can be clearly seen by considering the case where the initial policy and initial value function is the optimal policy and optimal value function respectively. Policy iteration is likewise parameterized, excepting the lack of policy iterations and thus setting of an initial policy.

3.2.2 Q-learning

As the Q-learning algorithm interacts with the environment it requires a few extra configurations. The agent will go through 25 000 episodes for each configuration. An episode is defined to end when the agent lands in a hole, reaches the goal, or reaches 1000 steps. It explores with epsilon greedy, starting with $\epsilon_0 = 0.5$ updating at each step so that $\epsilon_{t+1} = \epsilon_t * 0.999$. Note that ϵ is set to 0.5 at the start of the algorithm, not at the start of each episode.

The algorithm will then be run with different values for the learning rate ($\alpha = 1, \alpha = 0.9, \alpha = 0.1, \alpha = 0.01$) and discount factor ($\gamma = 1, \gamma = 0.9$). The performance is measured as a running average of the rewards from each episode.

3.3 CartPole

The cartpole problem is a classic continues problem where the algorithm attempts to keep a pole upright on top of a cart, by moving the cart left or right. Different versions use discrete actions (LEFT, RIGHT) or continuous from -1 to 1. The state shown to the agent is the poles current angle and angular velocity, and the carts current position and velocity. Each episode is failed if the cart moves to far from the center or the pole leans too far to either side. The agent gets a reward of 1 per timestep alive, as well as a penalty for moving away from the center of the stage, for a total $r = 1 - abs(x) * 0.01$ given at each timestep. . The episode also ends if 500 points is reached.

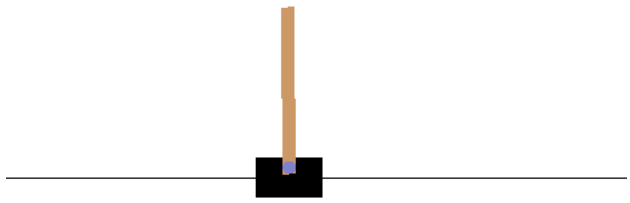

Figure 3.2: The CartPole environment rendered by Gym

3.3.1 REINFORCE

Like the Q-learning algorithm, REINFORCE learns through interaction. It interacts with the cartpole environment, attempting to keep the pole upright for 500 steps. The hyperparameters used are set to learning rate $\alpha = 0.001$ and discount factor $\gamma = 0.99$. The policy function is implemented through a deep NN with 2 or 3 hidden layers. All hidden layers have the same number of nodes, 10, 24 or 100. The NN is trained using Categorical Crossentropy (CC) as loss function and "adam" as optimizer. CC is reduced to $H(p, q) = A * log(\pi(s, a))$ as it is used. The implementation is based on the implementation by Woongwon et al. (2017).

3.3.2 Proximal Policy Optimization

The implementation of PPO used is based on an Keras implementation by OctThe16th (2018). The policy estimator is created using an NN with 2, 3 or 4 hidden layers with equal numbers of nodes. The number of nodes used varies between 10 and 200. The implementation uses a separate but identical value estimator. The hyperparameters used for all runs are presented in Table 3.1.

Learning rate	0.0001
Discount factor	0.99
Entropy coefficient	0.001
Cliprange	0.2
Epochs	10
Batch size	64
Number of steps per update	256

Table 3.1: Hyperparameters for PPO

3.4 Manipulator

Open AI GYM contains a robotic manipulator simulated using the physics engine Mujoco. There are multiple environments using this implementation, including FetchReach and FetchPush. All environments uses a state containing information about each joint of the manipulator, both position and angular velocity. Additionally, there are a few states related to the goal, dependent on the exact environment. PPO as described in Subsection 3.3.2 is used as the agent for all manipulator simulations.

For FetchReach this is the XYZ coordinates for the goal position. The action is a four continuous values denoting the XYZ coordinates to reach for the next movement and the final value controls the parallel gripper at the end. The feedback is given in one of two variations. Sparse rewards give -1 each step, and a 0 when the goal is reached, ending the episode. Dense rewards give negatively proportional to the euclidean distance between the end manipulator and the goal. The goal is considered reached if the end manipulator is within a certain distance from the goal. The episode can also end after 50 steps if the goal is not reached by that time, or if the agents sets the manipulator outside the operational space.

FetchPush is a problem where the goal is to move a block to a set location on the ground. The manipulator may simply push the block without picking it up. FetchPickAndPlace is a more advanced version, where the goal is suspended in the air, forcing the manipulator to grab the block to achieve the goal. The non-completion endings are as for FetchReach, while the dense and sparse rewards are instead based on the distance between the block and the goal. This means that the agents initial actions have no impact on the reward, and it must first reach the block before it will receive any feedback.

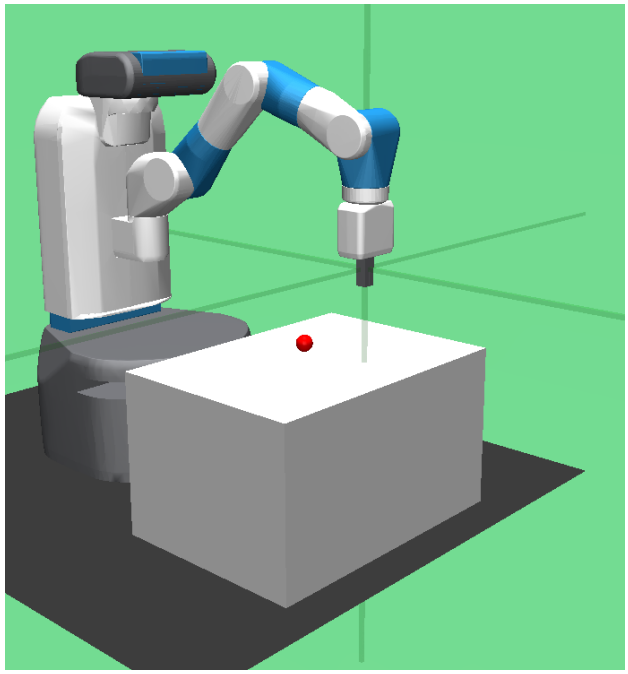

Figure 3.3: The FetchReach environment rendered by Gym. The red dot represent the target area.

Chapter 4

Results

This chapter will cover the output and results of each implemented method. Section 4.1 contains the results from the methods solving the FrozenLake problem, Section 4.2 the methods solving the CartPole problem, and Section 4.3 the results from the various RM problems.

4.1 FrozenLake

The FrozenLake problem was solved using PI, Subsection 4.1.1 , VI, Subsection 4.1.2, and Q-learning, Subsection 4.1.3.

4.1.1 Policy Iteration

Using all zeroes as the initial value function and the initial policy, $\gamma = 1$ and $\sigma = 10^{-8}$ it takes a total of 7 iterations to reach the optimal policy. The value function and the policy it generates is shown in Tables 4.1–4.8 The final policy and the value of each state is show in Table 4.8. The holes are shown as red cells, and the goal as green. Repeating the algorithm with $\gamma = 0.9$ results in a similar policy as seen in Table 4.9, with each state having a lower value. The algorithm also stopped one iteration earlier. Other values of $\gamma \in <0, 1>$ are similar. Using $\gamma = 0$ updates the one non-terminal state adjacent to the goal with the value 0.333, but the value does not spread and no usable policy is created. From table 4.10 it can be read that as $\gamma = 1$ and the value function of the starting state (0,0) is 0.824 the expected reward using this policy from this position is 0.824. As this is always the starting position, and this is the optimal policy, it can be interpreted as a 82.4% chance of reaching the goal when following an optimal policy.

←	←	←	←
←	←	←	←
←	←	←	←
←	←	←	←

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1

Table 4.1: Policy iteration, iteration zero

←	←	←	←
←	←	←	←
←	←	←	←
←	←	↓	←

0	0	0.038	0.019
0	0	0.077	0
0	0	0.192	0
0	0	0.500	1

Table 4.2: Policy iteration, iteration one

←	↓	→	↑
←	←	←	←
←	↓	←	←
←	↓	→	←

0	0.061	0.184	0.184
0	0	0.184	0
0	0.237	0.368	0
0	0.342	0.684	1

Table 4.3: Policy iteration, iteration two

↓	→	→	↑
←	←	←	←
↓	↓	←	←
←	→	→	←

0.144	0.114	0.227	0.227
0.174	0	0.227	0
0.205	0.409	0.455	0
0	0.568	0.727	1

Table 4.4: Policy iteration, iteration three

←	↑	→	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.750	0.542	0.333	0.333
0.750	0	0.333	0
0.750	0.750	0.667	0
0	0.833	0.917	1

Table 4.5: Policy iteration, iteration four

←	↑	←	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.780	0.658	0.536	0.536
0.780	0	0.415	0
0.780	0.780	0.707	0
0	0.854	0.927	1

Table 4.6: Policy iteration, iteration five

←	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.823	0.823	0.823	0.823
0.823	0	0.529	0
0.823	0.823	0.765	0
0	0.882	0.941	1

Table 4.7: Policy iteration, iteration six. Final policy have been reached

←	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.823	0.823	0.823	0.823
0.823	0	0.529	0
0.823	0.823	0.765	0
0	0.882	0.941	1

Table 4.8: Policy iteration, iteration seven. Final value function has been reached, same policy is generated and algorithm stops

←	↑	←	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.069	0.061	0.074	0.056
0.092	0	0.112	0
0.145	0.247	0.300	0
0	0.380	0.639	1

Table 4.9: Policy iteration with $\gamma = 0.9$, final policy and value function.

4.1.2 Value Iteration

Using all zeroes as the initial value function and the initial policy and $\gamma = 1$, it takes a total of 991 iteration for the value function to converge within the threshold $\sigma = 10^{-8}$. The final policy and the value of each state is show in Table 4.10. Using $\gamma = 0.9$ the iterations required is reduced to 55, 10 for 0.5 and 5 for 0.2, all producing the same policy. When reduced even further however, the policy changes, shown in Table 4.11 for $\gamma = 0.1$. At this point the value does not spread all the way to the top-left state before the change from one iteration to the next is bellow σ. With $\gamma = 0$ the effect is exactly the same as for PI, stopping after a single iteration.

←	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.824	0.824	0.824	0.824
0.824	0	0.529	0
0.824	0.824	0.765	0
0	0.882	0.941	1

Table 4.10: Value iteration using $\gamma = 1$, taking 991 iterations to reach convergence

←	↓	→	↑
←	←	←	←
↓	↓	←	←
←	→	↓	←

0	0	0.000	0.000
0	0	0.000	0
0.000	0.000	0.000	0
0	0.000	0.033	1

Table 4.11: Value iteration using $\gamma = 0.1$, taking 4 iteration to reach convergence. Values shown as 0.000 are very small non-zero numbers.

4.1.3 Q-learning

The algorithm was repeated with different parameterizations of α and γ, with the results being calculated as the moving average reward across 25 000 episodes each. The results are shown in Figure 4.1. They were chosen for their performance as measured by the moving average reward.

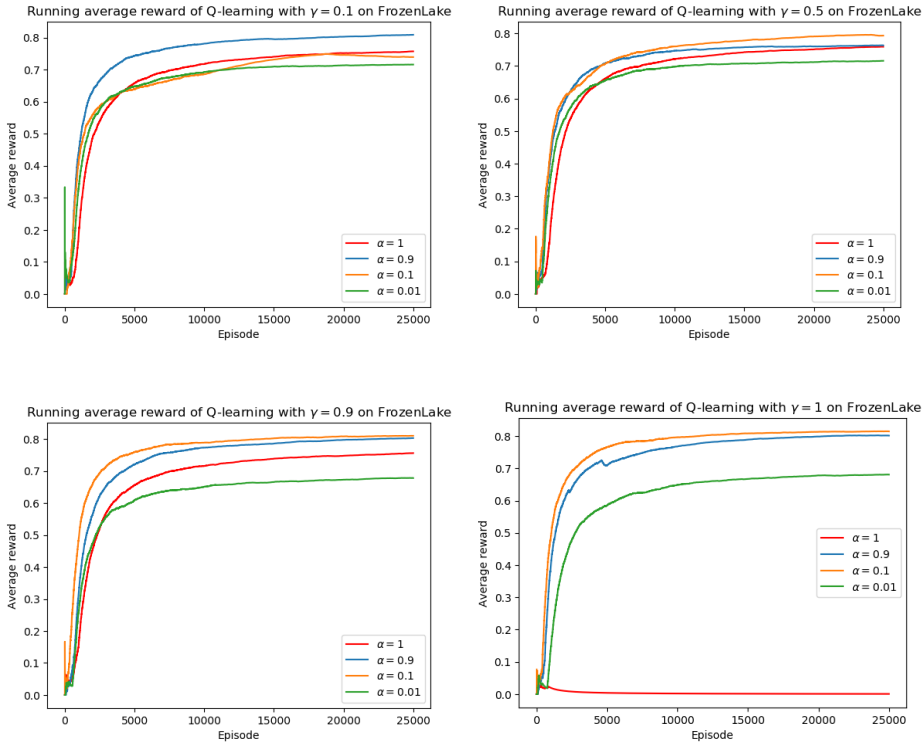

Figure 4.1: Results after 25000 episodes of FrozenLake using Q-learning with different parameters

Additionally the policy and extracted value function for the various configurations are in Tables 4.12–4.17. Most variations of the Q-learning algorithm approaches something near the maximum average score of 0.824, but there is a clear difference between parameters. The runs with $\alpha = 0.01$ performs the worst for three out of four values of γ, and second worst in the last. Similarly the runs with $\alpha = 1$ performs second worst in two runs, and in the run with $\gamma = 1$ fails completely. $\alpha = 0.9$ and $\alpha 0.1$ both gives the optimal policy when $\gamma = 0.9$. The left side of Table 4.14 and Table 4.15 shows this policy.

↑	↑	↑	↑
↓	←	←	←
→	↓	←	←
←	→	↑	←

0.947	0.947	0.947	0.947
0.794	0	0.989	0
0.309	0.975	0.105	0
0	0.916	1.000	1

Table 4.12: Policy after running 25000 episodes with Q-learning, $\alpha = 0.9$ and $\gamma = 1$. Note that this policy will be stuck in the upper row if employed as is.

↑	↑	↑	↑
←	←	←	←
↑	↓	←	←
←	→	←	←

0.829	0.829	0.829	0.829
0.736	0	0.331	0
0.593	0.535	0.460	0
0	0.528	0.512	1

Table 4.13: Policy after running 25000 episodes with Q-learning, $\alpha = 0.1$ and $\gamma = 1$. Note that this policy will be stuck in the upper row if employed as is.

←	↑	←	↑
←	←	→	←
↑	↓	←	←
←	→	↓	←

0.047	0.126	0.023	0.000
0.197	0	0.016	0
0.605	0.663	0.882	0
0	0.493	0.940	1

Table 4.14: Policy after running 25000 episodes with Q-learning, $\alpha = 0.9$ and $\gamma = 0.9$

→	↑	←	↑
←	←	←	←
↑	↓	←	←
←	→	↓	←

0.062	0.061	0.111	0.025
0.100	0	0.195	0
0.157	0.276	0.309	0
0	0.367	0.750	1

Table 4.15: Policy after running 25000 episodes with Q-learning, $\alpha = 0.1$ and $\gamma = 0.9$

↓	↑	←	↓
←	←	←	←
↑	→	↓	←
←	→	↑	←

0.000	0.000	0.001	0.000
0.000	0	0.001	0
0.000	0.000	0.008	0
0	0.000	0.001	1

Table 4.16: Policy after running 25000 episodes with Q-learning, $\alpha = 0.9$ and $\gamma = 0.1$

→	→	←	←
↓	←	↓	←
→	→	↓	←
←	↓	→	←

0.000	0.000	0.000	0.000
0.000	0	0.000	0
0.000	0.001	0.009	0
0	0.010	0.347	1

Table 4.17: Policy after running 25000 episodes with Q-learning, $\alpha = 0.1$ and $\gamma = 0.1$

4.2 CartPole

The CartPole problem was solved using first REINFORCE, then for comparison, PPO.

4.2.1 REINFORCE

Figure 4.2 to Figure 4.7 shows the reward per episode across 4000 episodes each. There were no large variations in computation speed, as the simulation took far greater time in comparison. Thus, while there were major variations in speed, they were dependent on the length of the episode, which is equal to the score, and negligibly correlated with the complexity of the NN. The agent using an NN with three hidden layers of 24 nodes, Figure 4.6, and both agents using 100 nodes per layer, Figures 4.4 and 4.7, suffers a complete non-recovering failure, though they do manage to solve at least one or more episodes first. The two layered 24 node network also failed in a second run seen in figure 4.8. The remaining networks also suffer from repeated intermittent complete failures for some number of episodes, before rapidly recovering.

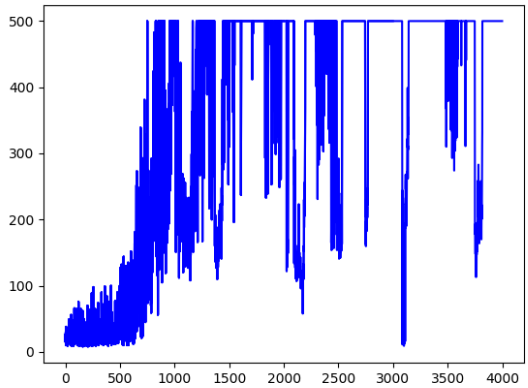

Figure 4.2: Reward per episode for REINFORCE with a NN with two hidden layers of ten nodes applied to the CartPole problem.

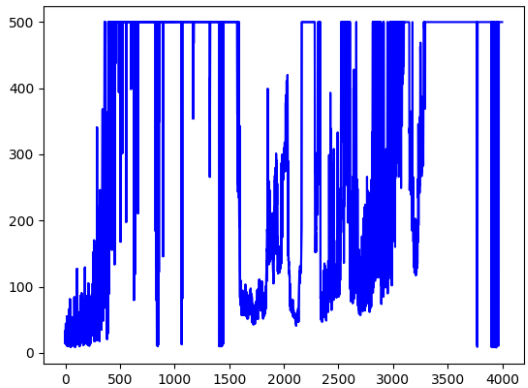

Figure 4.3: Reward per episode for REINFORCE with a NN with two hidden layers of 24 nodes applied to the CartPole problem.

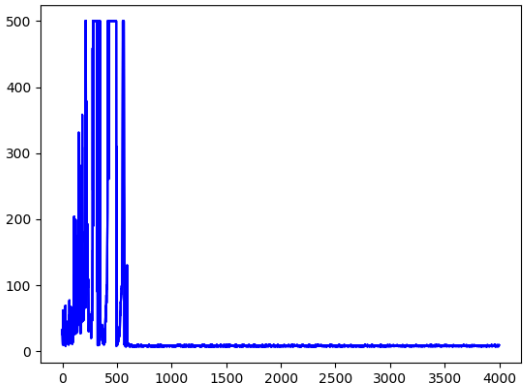

Figure 4.4: Reward per episode for REINFORCE with a NN with two hidden layers of 100 nodes applied to the CartPole problem.

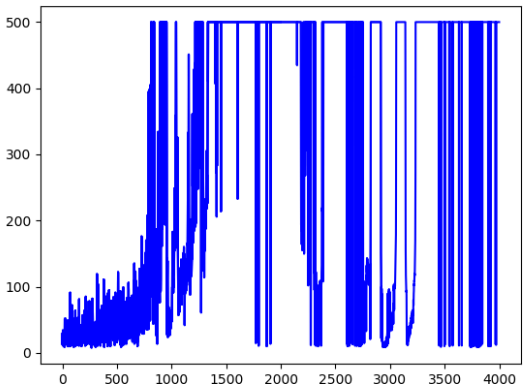

Figure 4.5: Reward per episode for REINFORCE with a NN with three hidden layers of ten nodes applied to the CartPole problem.

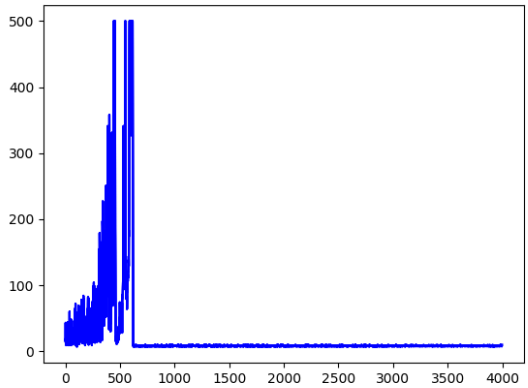

Figure 4.6: Reward per episode for REINFORCE with a NN with three hidden layers of 24 nodes applied to the CartPole problem.

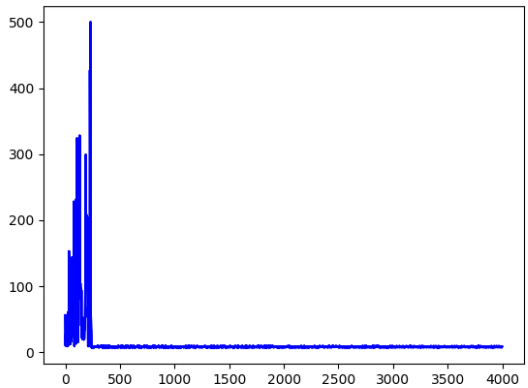

Figure 4.7: Reward per episode for REINFORCE with a NN with three hidden layers of 100 nodes applied to the CartPole problem.

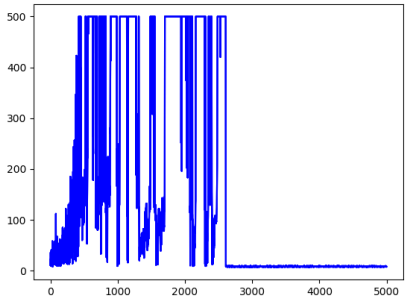

Figure 4.8: A second plot of reward per episode for REINFORCE with a NN with two hidden layers of ten nodes applied to the CartPole problem.

4.2.2 Proximal Policy Optimization

PPO ran a total of 6 times, varying the NN each time. The different configurations as well as their computation times for 4000 episodes of CartPole can be seen in Table 4.18. The reward is plotted against the episode number for each of the runs and shown in Figure 4.9 and Figure 4.10. Additionally the actor and critic loss for runs using three hidden layers are shown in Figure 4.11 and Figure 4.12 respectively. As with REINFORCE there are intermittent drops in performance, but they are universally quicker to recover. Additionally, for the agents using three hidden layers the drops are far shorter, still maintaining more than 250 points per episode in most cases.

One notable result is in the agent using three hidden layers with 24 nodes each, in Figure 4.10 near the 4000th episode there is a large drop that does not fully recover by the end of the run. In Figure 4.11 and Figure 4.12, there is a clear reaction shortly afterwards, corresponding to when the performance starts to recover. By far the most consistent performance is achieved by the agent using three hidden layers of 10 nodes each, though it is slow to reach stability. In general simpler networks learns faster while more complex networks are more stable once trained.

Layers	Nodes per layer	Time
2	10	16m 17s
2	24	17m 30s
2	100	20m 11s
3	10	19m 24s
3	24	21m 34s
3	100	25m 47s

Table 4.18: Computation time for different runs of PPO using neural networks with the specified number of layers and nodes per layer

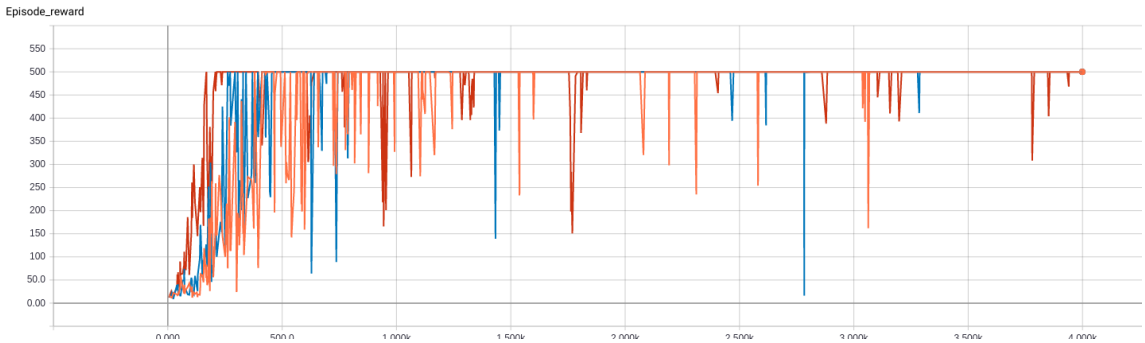

Figure 4.9: Plot of reward against episode number for PPO solving CartPole. The three graphs are all using two hidden layers. Red is 100 nodes per layer, Blue is 24 and Orange is 10.

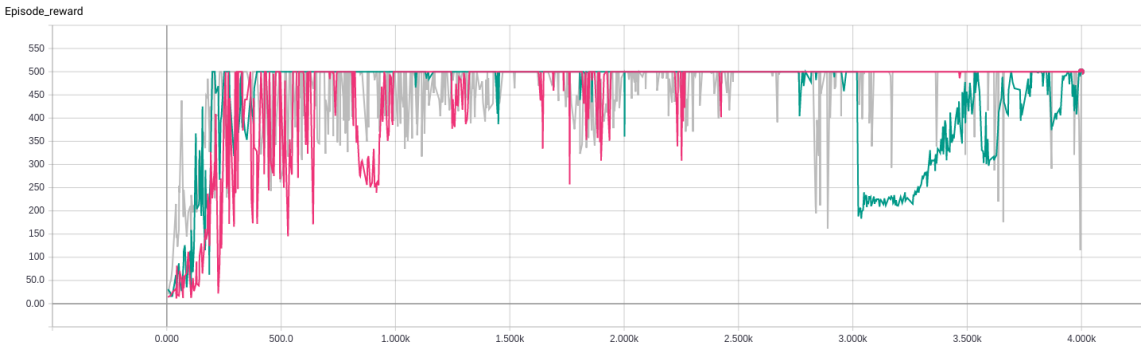

Figure 4.10: Plot of reward against episode number for PPO solving CartPole. The three graphs are all using three hidden layers. Grey is 100 nodes per layer, green is 24 and pink is 10.

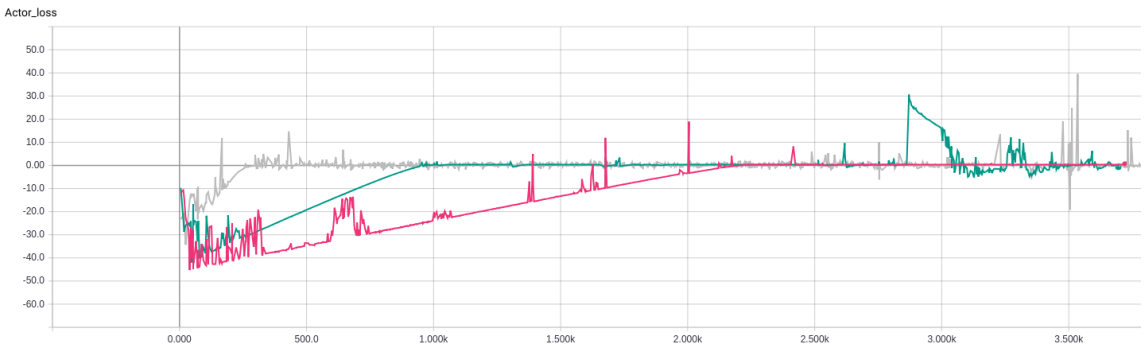

Figure 4.11: Plot of actor loss against batch number for PPO solving CartPole. The three graphs are all using three hidden layers. Grey is 100 nodes per layer, green is 24 and pink is 10.

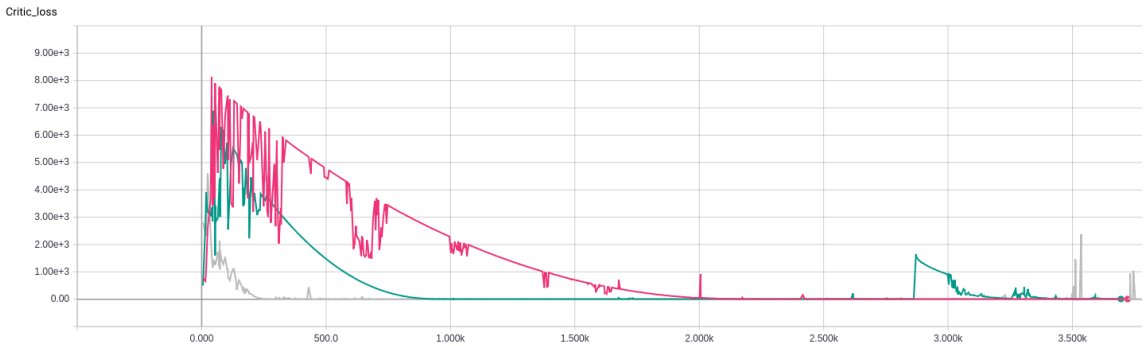

Figure 4.12: Plot of critic loss against batch number for PPO solving CartPole. The two graphs are both using two hidden layers. Pink is 24 nodes per layer, blue is 128.

4.3 Manipulator

Figure 4.13 through Figure 4.15 shows the results from the FetchReach problem, using two layers. The graphs are smoothed for readability; the original values can be seen faded in the background. There is high variance between individual episodes due to the random distance from the initial position to the target. Compared to the graphs generated when solving the CartPole problem, Figure 4.13 and Figure 4.14 shows noticeable less smooth developments, indicating there are many adjustments made after each batch attempting to model the far more complex system. It also takes far more episodes to create a functional policy. The more complex NN performs far better than the simpler 24 nodes per layer NN.

Figure 4.16 shows the results from the FetchPush problem, where the agent should push a block to specified location (not necessarily pick up), and Figure 4.17 shows the result from the FetchPickAndPlace problem, where the agent is to pick up a block, and move it to a location suspended in the air. Both figures shows result equivalent to that of taking random actions at each step.

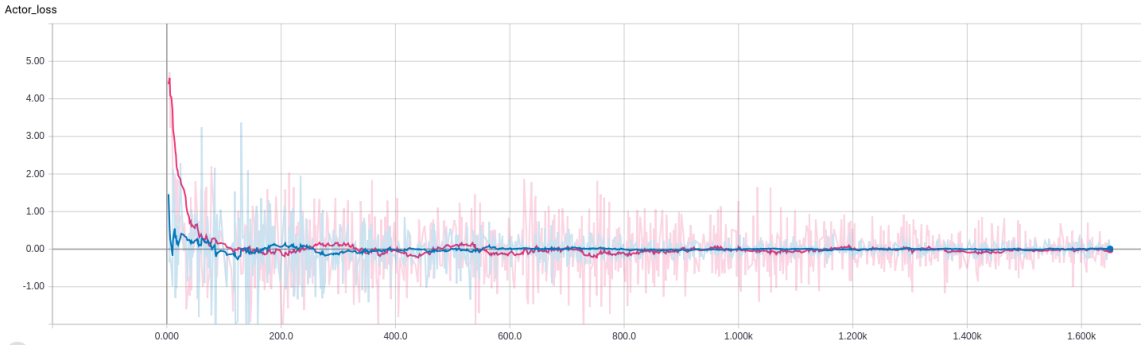

Figure 4.13: Plot of actor loss against batch number for PPO solving FetchReach. The two graphs are both using two hidden layers. Pink is 24 nodes per layer, blue is 128.

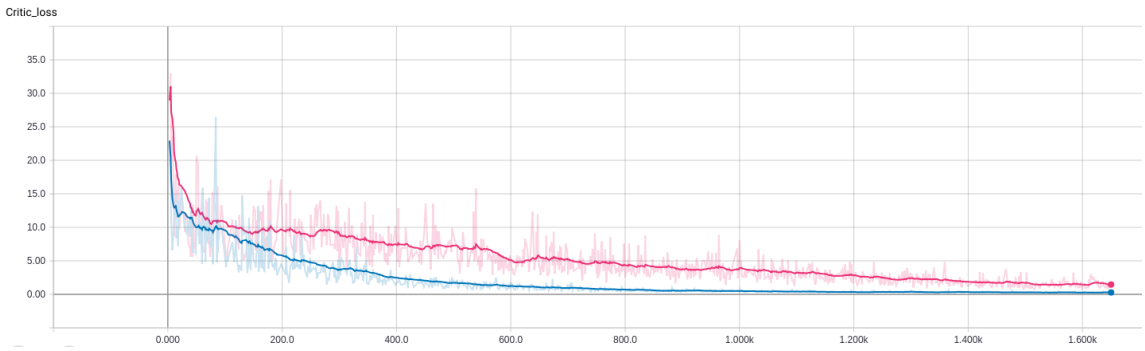

Figure 4.14: Plot of critic loss against batch number for PPO solving FetchReach. The three graphs are all using two hidden layers. Red is 100 nodes per layer, Blue is 24 and Orange is 10.

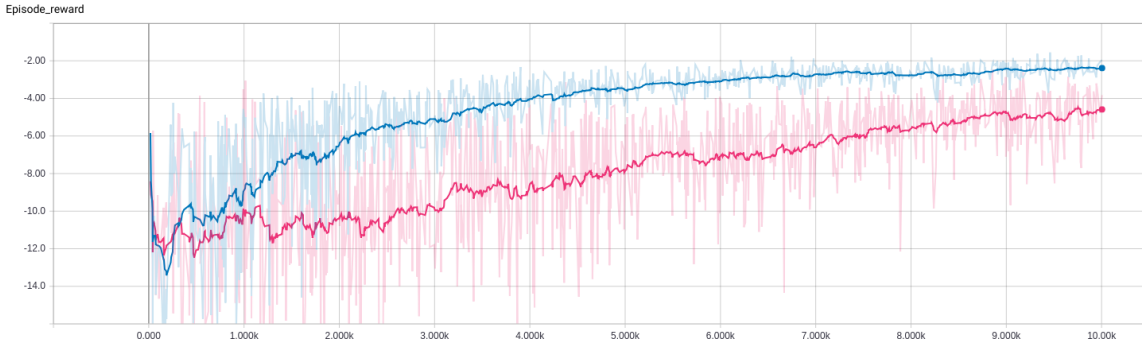

Figure 4.15: Plot of reward against episode number for PPO solving FetchReach. The two graphs are both using two hidden layers. Pink is 24 nodes per layer, blue is 128.

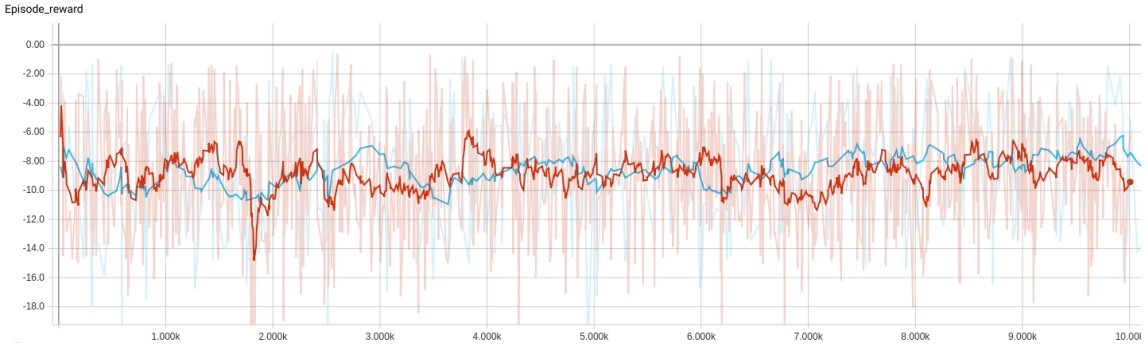

Figure 4.16: Plot of reward against episode number for PPO solving FetchPush. The two graphs are both using three hidden layers. Red is 128 nodes per layer, blue is 256.

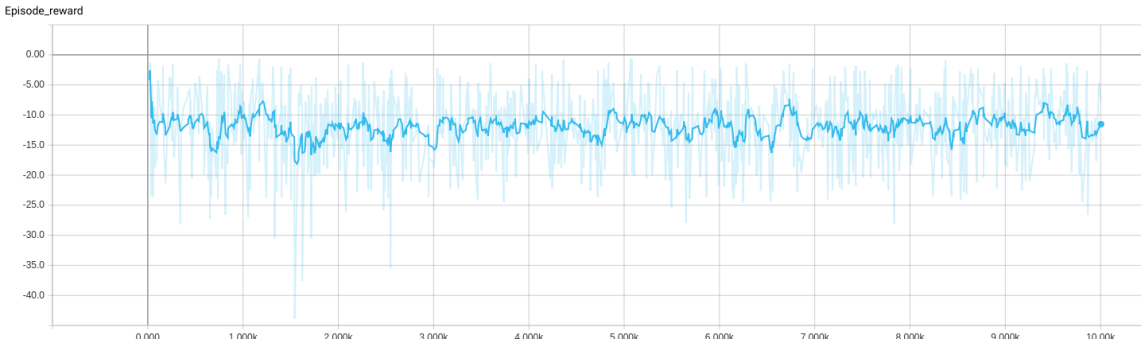

Figure 4.17: Plot of reward against episode number for PPO solving FetchPickAndPlace, using three hidden layers of 128 nodes each.

Chapter 5

Discussions

5.1 Methods in Reinforcement Learning

VI and PI are two very similar methods used for solving the same category problems, MDPs where each element of the problem is known. The results from Sections 4.1.1 and 4.1.2 show that the same policy is reached for the same values of σ and γ. However, the resulting value functions differ slightly between the methods, even while using the same $V^0(s)$ and γ, as PI does not guarantee converging on the true value of each state. From Table 4.10, it can be concluded that as the value function of the starting state (0,0) is 0.824, there is an 82.4% chance of reaching the goal when following an optimal policy. VI requires fever calculations to reach convergence, but each of those calculations for each of those steps calculates the nonlinear max function, whereas PI computes linear operations for the majority of the operations, only using the max operator during the infrequent policy improvement iterations. Each method demonstrate the iterative working of an RL agent solving a problem.

For solving the same FrozenLake problem, Q-learning is massively slower and less effective, as it is denied the perfect information of the environment and must rely on exploration. Tables 4.14–4.17 shows that most configurations resulted in the agent learning a policy quite close to the optimal policy. From Figure 4.1 it can be seen that Q-learning agents with less extreme values for α perform similarly well in most cases, indicating that the optimal parameter values lies somewhere in between. As they switch places dependent on the value of γ, there is likely no one value that would lead to the best performance in all cases. There is also a clear correlation between parameter pairs that learn quickly and converges on the highest values.

REINFORCE manages to solve the task of holding the pole upright for the full length of an episode across all configurations, but is also highly unstable. Multiple agents ends up failing consistently after a few hundred episodes, and never recover. Notably they are employing the NNs with the most nodes. It is clear that this would be unsuitable for implementation in any system where failure can cause damages.

PPO, when applied to the same CartPole problem achieved far more promising results. There is a clear progression of learning, and a mostly consistent policy is quickly achieved, often within one- to two-hundred episodes. The same intermittent drops in performance are present with less intensity and frequency, and the collapses without recovery do not happen. After a longer than usual series of low-performance by the agent employing an NN of three hidden layers of 24 nodes, there is a reaction in the actor and critic loss calculations. Interestingly, the actor loss is positive in this instance, unlike the earlier changes, which where for all runs an approximately smooth graph going from some negative number linearly up to zero. The reason for the linear effect is due to the clipping function that prevents any too large change is policy that is perceived to cause a better result. From Table 4.18 there is a trend that larger, more complex NNs have a noticeable impact on computation time, even when accounting for the differing length of episodes.

5.2 Manipulator Control

For the other manipulator problems PPO were applied too, the agents were in all runs unable to get any feedback from the exploration, as it never stumbled upon anything that led it closer to the goal. This could potentially be solved through an extremely long training session, until it stumbles upon a reward, but this would still not guarantee that it will learn anything useful. A recommendation from the creators of the problem is to use Hindsight Experience Replay (HER), (Andrychowicz et al., 2017). This is used to let the agent pretend that whatever it just did, was the actual goal, and learn how those action, if that was the actual goal, would give a reward. Through careful use, this may allow the agent to infer the workings of the system and what actions are lead to a reward. Another, simpler, technique would be to let the robot start closer to a beneficial position for every other episode during early training, drastically increasing the chance that it will stumble upon a reward. A third possibility would be to separate the problem into multiple parts, locating and moving the manipulator to the block, grasping the block and finally moving the block.

For all runs on the same problem, the only variation in configuration of the agent have been in the size of the NN. Further attempts could see improvements in performance on the FetchReach problem by adjusting other parameters, such as the activation function of the nodes and the hyperparameters of the PPO agent. These adjustments are unlikely to affect the success on the problems where the agent never found an action that changed the reward received, but may improve performance when such an action is discovered.

Chapter 6

Conclusions

This project have introduced, implemented and looked at the results for various methods of control through RL. From the simple tabular methods PI and VI and the temporal difference method Q-learning solving the FrozenLake problem, to the more complex REINFORCE and PPO solving the CartPole problem and finally the PPO method employed for control of a simulated robotic manipulator.

PI and VI are very simple methods that demonstrate the basics of RL. They require complete knowledge of the system, including the transition function and the reward function, which makes them unsuitable for most real world control applications. They are limited to discrete state-space and action-space, further invalidating their usability. Q-learning expands on the theory of RL by introducing exploration and exploitation, but remains bound by discrete state- and action-space. The method's exploration strategy does not avoid failures, putting any real world application controlled using this method at risk. REINFORCE is the first method explored in this project that fulfill the basic requirements for controlling an RM. It works for continuous state- and action-spaces and is an on-policy method. It introduces the policy estimator, in form of an NN circumventing the issue of storing values for each of the possible states. It manages to reach the maximal score for the CartPole problem, but is notably unstable. Training is highly dependent on the parameters used, for which there are no clear optimal values.

These methods where explored as tools for introducing the development and methods of RL, leading up to PPO. PPO showed clear advantages over REINFORCE when applied to the CartPole problem, as well as being fully capable of solving the FetchReach problem. This showed an ability to handle the large state-space required, and to infer the dynamics of the system. For more complex tasks, however, where the reward functions feedback was dependent on reaching some specific part of the state-space before giving any change in feedback, a naive application of the agent was not enough to solve the problem. This showed that the design of the reward function is a critical part of a RL problem alongside the choice and training of the agent. Additionally, training aids such as controlling the initial state, sub goals and otherwise aiding the agent in discovering the solution may be required for complicated tasks where the true goal is hidden behind some preparatory action. Future work will involve designing a reward function that will allow the agent to solve more complex tasks, and

further investigation of the effect of changing various hyperparameters of the PPO agent. In my master thesis building upon the research of this project I will connect the PPO agent to a simulation of a more complex task, the turning of a valve. Initial research will begin with the valve grasped, and the reward being directly linked to the angle of the valve. Even further development will include locating and grasping the valve from a randomized initial position in the vicinity, with the aid of a visual module for image recognition.

Bibliography

Ajwad, S., Ullah, M., Khelifa, B., and Iqbal, J. (2014). A comprehensive state-of-the-art on control of industrial articulated robots. *Journal of Balkan Tribological Association*, 20(4):499–521.

Allgöwer, F., Badgwell, T. A., Qin, J. S., Rawlings, J. B., and Wright, S. J. (1999). Nonlinear predictive control and moving horizon estimation—an introductory overview. In *Advances in control*, pages 391–449. Springer.

Analytics India Magazine. Frozen-lake. `https://www.analyticsindiamag.com/wp-content/uploads/2018/03/Frozen-Lake.png`. Accessed: 2018-09-07.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058.

Arimoto, S. (1994). State-of-the-art and future research directions of robot control. *IFAC Proceedings Volumes*, 27(14):3–14.

Chow, Y., Nachum, O., Duenez-Guzman, E., and Ghavamzadeh, M. (2018). A lyapunov-based approach to safe reinforcement learning. *arXiv preprint arXiv:1805.07708*.

Dubowsky, S. and DesForges, D. (1979). The application of model-referenced adaptive control to robotic manipulators. *Journal of dynamic systems, measurement, and control*, 101(3):193–200.

Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838.

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274.

Landau, I. (1974). A survey of model reference adaptive techniques-theory and applications. *Automatica*, 10(4):353–379.

Levine, S. and Koltun, V. (2013). Guided policy search. In *International Conference on Machine Learning*, pages 1–9.

Lewis, F. L. (2004). Robot manipulator control : theory and practice.

Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Luh, J. Y., Walker, M. W., and Paul, R. P. (1980). On-line computational scheme for mechanical manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 102(2):69–76.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

nptel. course on mechatronics and manufactoring automation. `https://nptel.ac.in/courses/112103174/`. Accessed: 2018-08-30.

OctThe16th (2018). PPO-Keras. `https://github.com/OctThe16th/PPO-Keras`.

Paul, R. (1972). Modelling, trajectory calculation and servoing of a computer controlled arm. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

Paul, R. P. (1981). *Robot manipulators: mathematics, programming, and control: the computer control of robot manipulators*. Richard Paul.

Poignet, P. and Gautier, M. (2000). Nonlinear model predictive control of a robot manipulator. In *Advanced Motion Control, 2000. Proceedings. 6th International Workshop on*, pages 401–406. IEEE.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization. *arXiv preprint arXiv:1502.05477*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sciavicco, L. and Siciliano, B. (2012). *Modelling and control of robot manipulators*. Springer Science & Business Media.

Siciliano, B. (2009). Robotics : Modelling, planning and control.

Spong, M. W. (1989). Robot dynamics and control.

Sutton, R. S., Barto, A. G., and Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12(2):19–22.

Takegaki, M. and Arimoto, S. (1981). A new feedback method for dynamic control of manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 103(2):119–125.

Uchiyama, M. (1989). Control of robot arms. *JSME international journal. Ser. 3, Vibration, control engineering, engineering for industry*, 32(1):1–9.

Whitney, D. E. (1987). Historical perspective and state of the art in robot force control. *The International Journal of Robotics Research*, 6(1):3–14.

Wichman, W. M. (1967). Use of optical feedback in the computer control of an arm. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE.

Wiering, M. and Otterlo, M. (2012). Reinforcement learning : State-of-the-art.

Wikipedia. Colored neural network. ://en.wikipedia.org/wiki/File:Colored$_neural_network.svg$.

Wikipedia. A set of six-axis robots used for welding. https://en.wikipedia.org/wiki/File:FANUC_6-axis_welding_robots.jpg. Accessed: 2018-09-07.

Woongwon, Youngmoo, Hyeokreal, Uiryeong, and Keon. (2017). Reinforcement learning. https://github.com/rlcode/reinforcement-learning.

Yahya, A., Li, A., Kalakrishnan, M., Chebotar, Y., and Levine, S. Collective robot reinforcement learning. https://www.youtube.com/watch?time_continue=7&v=QZvu8M02BeE. Accessed: 2018-09-07.

Yoshikawa, T. (2000). Force control of robot manipulators. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 220–226. IEEE.

Appendices

Appendix A

Code

A.1 Policy Iteration

```
1  import numpy as np
2  import my_logger as log
3
4  def policy_evaluation(env, policy, gamma=1, theta=1e-8):
5      V = np.zeros(env.nS)
6      iterations = 0
7
8      delta = theta+1
9      while delta >= theta:
10         iterations += 1
11         delta = 0
12         for s in range(env.nS):
13             Vs = 0
14             for prob, next_state, reward, done in env.P[s][policy[
                   s]]:
15                 Vs += prob * (reward + gamma * V[next_state])
16             delta = max(delta, np.abs(V[s]-Vs))
17             V[s] = Vs
18     return V, iterations
19
20 def policy_iteration(env, initial_policy = None, gamma=1, theta=1e
       -5):
21     policy = np.zeros(env.nS)
22     if initial_policy is not None:
23         policy = initial_policy
24
```

```
    iterations = 0
    evaluations = []

    f = log.log_init("PI.txt")
    while True:
        iterations += 1
        V, i = policy_evaluation(env, policy, gamma, theta)
        evaluations.append(i)
        log.log(f, policy, V)

        convergence = True
        for s in range(env.nS):
            a_vals = np.zeros(env.nA)
            for a in range(env.nA):
                for prob, next_state, reward, done in env.P[s][a]:
                    a_vals[a] += prob * (reward + gamma * V[
                        next_state])

            if policy[s] != np.argmax(a_vals):
                policy[s] = np.argmax(a_vals)
                convergence = False

        if convergence:
            log.log(f, policy, V)
            log.log_close(f)
            break;

    return policy, V, iterations, evaluations
```

A.2 Value Iteration

```
import numpy as np


def value_iteration(env, gamma = 1, theta = 1e-8):
    V = np.zeros(env.nS)
    delta = theta+1
    iterations = 0
    while delta > theta*(1-gamma)/gamma:
        iterations += 1
        delta = 0
        for s in range(env.nS):
            a_vals = np.zeros(env.nA)
            for a in range(env.nA):
```

```
                for prob, next_state, reward, done in env.P[s][a]:
                    a_vals[a] += prob*(reward + gamma*V[next_state
                        ])

            v_new = gamma*np.max(a_vals)
            delta = max(abs(v_new - V[s]), delta)
            V[s] = v_new

    pi = np.zeros(env.nS)
    for s in range(env.nS):
        a_vals = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[s][a]:
                a_vals[a] += prob * (reward + gamma * V[next_state
                    ])
        pi[s] = np.argmax(a_vals)

    return pi, V, iterations
```

A.3 Q-learning

```
import numpy as np
import time
from os import system

epislon = 0.5
def choose_greedy(q,nEps, greed):
    global epislon
    epislon *= 0.9999
    if np.random.uniform() < max(epislon,0) and not greed:
        return(np.random.randint(0,4))
    return np.argmax(q)

def q_learning(env, episodes = 100, rate=0.5, discount=0.9):
    q = np.random.rand(env.nS, env.nA)
  #Set the value of any terminal action to 0.
    for s in range(env.nS):
        for a in range(env.nA):
            for outcome in env.P[s][a]:
                if outcome[3]:
                    q[s,a] = 0
    nEps = 0
    while nEps < episodes:
        nEps += 1
```

```
24          env.reset()
25          done = False
26
27          while not done:
28              s = env.s
29              a = choose_greedy(q[s],nEps, False)
30              s_new, r, done, _ = env.step(a)
31              q[s, a] = q[s, a] + rate*(r + discount*q[s_new,
                    choose_greedy(q[s_new], nEps, True)]-q[s, a])
32      return q
```