Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Tharald Jørgen Stray

Application of deep reinforcement learning for control problems

Master's thesis in Cybernetics and Robotics
Supervisor: Ole Morten Aamo
January 2019

NTNU
Norwegian University of
Science and Technology

Tharald Jørgen Stray

Application of deep reinforcement learning for control problems

Master's thesis in Cybernetics and Robotics
Supervisor: Ole Morten Aamo
January 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

This thesis is the result of completing the course "TTK4900 - Engineering Cybernetics, Master's Thesis" at the Norwegian University of Science and Technology. The work was carried out during the autumn semester of 2018, and is the final project in the Cybernetics and Robotics study programme.

The thesis idea was proposed by my supervisor, Professor Ole Morten Aamo. Inspired by recent achievements of deep reinforcement learning methods in games such as Go and Dota 2, the basic goal was to investigate if similar methods could be applied to control problems. References to the OpenAI Baselines open-source implementations of DRL algorithms [9], and the AlphaGo Zero algorithm [32] were included, and combined with the suggested tasks established the basis for the thesis.

The candidate algorithm of choice, DDPG, was presented by Lillicrap et al. [19]. As suggested by the thesis proposal, an open-source implementation was used, forked from the Spinning Up [2] repository (as specified in section 2.3 and 3.4). The proposed environments utilize the interface outlined by OpenAI Gym [5]. All code has been written using the Python programming language, and its broad ecosystem has been utilized in many aspects of the implementation: the SciPy package is used for ODE solving and the NumPy package is used for array objects and random number generators. GitHub repositories were used f or version control.

Professor Ole Morten Aamo has provided valuable help and suggestions throughout the thesis work.

The reader is assumed to have a technical background, with some basic knowledge of control theory, numerical optimization, object-oriented programming and software architecture.

Acknowledgment

I would like to thank my supervisor, Professor Ole Morten Aamo for all the help and guidance during the work with this thesis. A warm "thank you" also goes out to all my friends and family for the support they have provided.

Summary

In cybernetics, the control approach usually relies on, in some way or other, explicitly designing and implementing a controller based on some analysis of the system. The deep reinforcement learning control (DRL) approach is different. No part of a DRL agent's behaviour policy has been explicitly implemented, rather, the policy has been *learned*, through repeatedly interacting with its environment. This is one of the underlying motivations for machine learning. The solutions to some tasks are hard to program explicitly. Because of the continued increase in computational power available, problems in machine learning that were previously regarded as too computationally expensive have now become almost trivial to solve.

Motivated by recent successes in the field of DRL, this thesis investigates the possibility of applying the same concepts underpinning DeepMinds's AlphaZero, to control problems. A simulation framework based on the OpenAI Gym interface is presented, and some example scenarios are implemented. Different open-source implementations of DRL alorithms are explored and discussed.

Sammendrag

Innenfor kybernetikk er tilnærmingen til prosesskontroll ofte basert på eksplisitt design og implementasjon av en regulator, basert på analyser av systemet. I dyp forsterkende læring er tilnærmingen svært forskjellig. Ingen deler av agentens oppførsel er eksplisitt implementert, men *lært*, gjennom gjentatt interaksjon med omgivelsene sine. Dette er en av de underliggende motivasjonene bak maskinlæring, løsningen på noen problemer er svært vanskelige å programmere eksplisitt. På grunn av økende tilgang til prosessorkraft, har problemer innenfor maskinlæring som man tidligere trodde var uløselige, nå blitt nærmest trivielle.

Motivert av nylige suksesser innenfor dyp forsterkende læring, undersøker denne oppgaven muligheten for å anvende de samme konseptene som ligger bak DeepMinds AlphaZero, for å løse reguleringsoppgaver. Et simuleringsrammeverk basert på grensesnittet til OpenAI Gym blir presentert, og noen scenarioeksempler blir utledet. Forskjellige implementasjoner av DRL-algoritmer med åpen kildekode blir utforsket og diskutert.

Table of Contents

Preface 1

Acknowledgment 3

Summary i

Sammendrag i

Table of Contents iv

Abbreviations v

1 Introduction 1
- 1.1 Background 1
- 1.2 Goals 2
- 1.3 Outline 3

2 Theoretical background 5
- 2.1 Machine learning 5
  - 2.1.1 Supervised learning 6
  - 2.1.2 Artificial neural networks 7
  - 2.1.3 Deep learning 8
- 2.2 Reinforcement learning 11
  - 2.2.1 Problem formulation 12
  - 2.2.2 Value function 14

2.2.3 Exploration vs. exploitation . . . 15
2.2.4 Model-free vs. model-based RL . . . 16
2.2.5 Temporal difference learning . . . 18
2.2.6 Policy optimization . . . 21
2.2.7 Deep deterministic policy gradient . . . 25
2.3 DRL algorithm implementations . . . 31
2.3.1 OpenAI Baselines . . . 32
2.3.2 Stable Baselines . . . 32
2.3.3 OpenAI Spinning Up . . . 33
2.3.4 RLlib . . . 34

3 Implementation **35**
3.1 Systems . . . 35
3.1.1 First order system . . . 35
3.1.2 Second order system . . . 36
3.2 PID controller . . . 37
3.2.1 Tuning . . . 38
3.3 Environments . . . 38
3.3.1 Generalization and dynamics randomization . . . 39
3.3.2 Observation contents . . . 39
3.3.3 Reward function . . . 40
3.3.4 Simulation of system dynamics . . . 41
3.3.5 OpenAI's Gym interface . . . 41
3.3.6 Environment classes . . . 43
3.4 DDPG algorithm . . . 49
3.4.1 Neural networks . . . 50
3.4.2 Hyperparameters . . . 51

4 Experiments and results **53**
4.1 Training . . . 53
4.1.1 Network architectures . . . 53
4.1.2 Hyperparameters . . . 54
4.1.3 Performance . . . 55

5 Conclusion **59**
5.1 Further work . . . 60

Bibliography **61**

Abbreviations

AI	Artificial intelligence
ANN	Artificial neural network
API	Application programming interface
DDPG	Deep deterministic policy gradient
DNN	Deep neural network
DRL	Deep reinforcement learning
MDP	Markov decision process
ML	Machine learning
RL	Reinforcement learning

Chapter 1

Introduction

The goal of reinforcement learning, a subfield of machine learning, can be summarized as follows: an agent attempts to learn a policy which will allow it to control a dynamic environment to maximize some objective function. Although a simplification, this formulation is constructed mainly to catch the attention of anyone with experience in the field of control theory, as this should seem very similar to what their own field of study attempts to accomplish. However, the two fields have different approaches to the problem of controlling dynamic systems, with control theory being mainly model-driven, and machine learning being mainly data-driven. This thesis will explore the theory behind the relatively new field of deep reinforcement learning, with a focus on how it can be used in problems related to control theory.

1.1 Background

The field of artificial intelligence (AI) has seen a surge of interest lately, both in academia and in the industry. This is mostly due to recent successes in machine learning, which is a subfield of AI, in many different areas such as computer vision, speech recognition, and more. Another astonishing achievement has taken place in a subfield of machine learning called reinforcement learning (RL), namely the superhuman performance in the games Chess and Go.

Deep Q-Networks (DQN) were first presented by Mnih et al. [22] in 2013, and then further explained in-depth in a paper published in Nature [23] in 2015. This is considered a breakthrough, because for the first time, reinforcement learning was successfully combined with deep neural networks at scale, and achieving high performance in advanced control tasks (Atari games), in some cases even beating human experts. DQN ignited the current wave of interest and development in the field of deep reinforcement learning, by combating the stability and convergence issues of using DNNs in RL methods [16]. Building on concepts introduced by DQN, DeepMind's AlphaZero [32] beat the best human player in the game of Go, a feat which most AI experts thought was years away, while crushing all the previous best chess computers using the same general learning algorithm.

1.2 Goals

The underlying motivation for this master's thesis was to explore if some of the approaches and concepts described above could be adopted to solve control problems. To investigate this, some subtasks were given:

1. Conduct a literature study into the realm of deep reinforcement learning, and identify a suitable algorithm
2. Implement a simulator for different scenarios, such as "setpoint change", "input disturbance", and/or "output disturbance"
3. For reference, implement a controller with reasonable tuning
4. Train the algorithm until acceptable performance is obtained
5. If time permits, implement more processes

The final result consists of different parts. The theory section is aimed at anyone with a control theory background interesting in learning more about the field of RL, especially DRL, and includes a short review of different open-source algorithm implementations and solutions. A general simulation framework for constructing RL environments out of processes and different scenarios, with an interface based on the unified OpenAI Gym interface, is presented. Some approaches to using DRL in control problems are presented. A PID controller implementation which can be used with the simulator is included. The results of some experiments with an open-source DRL algorithm and the simulator are discussed.

Five years of studying control theory has provided me with knowledge in areas which are central in reinforcement learning, such as linear algebra and numerical optimization. However, reinforcement learning is a vast field of study, and during my time at NTNU, I have only had one course which introduced the concept briefly. Consequently, a good portion of the time spent working with this thesis was devoted to the literature study. The different sources of information, combined with the open-source implementations of different algorithms [9][2] and the framework of OpenAI Gym [5] provide excellent tools for not only learning about deep RL theory, but also learning how to use it in practice. Using these tools has allowed me to test and play around with different high performing implementations of cutting edge algorithms while reading the theoretical background and papers, which has been an invaluable addition. Some parts of this endeavour are suitable as content of a thesis such as this, others are not. However, I have learned more during this thesis work than any other semester at NTNU, and I hope it has resulted in a good thesis as well.

> "My work consists of two parts: of the one which is here, and of everything which I have not written. And precisely this second part is the important one." - Ludwig Wittgenstein [21]

1.3 Outline

The thesis is structured as follows:

- Chapter 1. Introduction: Presents the background and goals of the thesis.
- Chapter 2. Theoretical background: Some underlying theoretical concepts are explained and summarized, with a focus on deep reinforcement learning.
- Chapter 3. Implementation: The implementations of the thesis are detailed and discussed.
- Chapter 4. Experiments and results: Some experiments and their results are presented and discussed.
- Chapter 5. Conclusion: A short conclusion and some recommendations for further work.

Chapter 2

Theoretical background

This chapter will provide some theoretical background for the algorithms and methods studied in this thesis. First, some general concepts of machine learning will be presented, and the rest will mostly focus on reinforcement learning specifically. Some control theory concepts will also be touched upon briefly. Most of the material in this chapter is distilled from various books, papers and articles from the web. Worth mentioning specifically are the books "Artificial Intelligence: A Modern Approach" by Russel and Norvig [28], "Deep Reinforcement Learning" by Li [16], and "Reinforcement Learning: An Introduction" by Sutton and Barto [33], the UCL reinforcement learning course by Silver [30], OpenAI's educational resource "Spinning Up" [2], and the papers "Human-level control through deep reinforcement learning" by Mnih et al. [23], "Deterministic policy gradient algorithms" by [31], and "Continuous control with deep reinforcement learning" by Lillicrap et al. [19].

2.1 Machine learning

Machine learning is a subfield of artificial intelligence which focuses on how a computer can *learn* (ie. improve its performance on a task), by applying various statistical methods on data. Machine learning can be roughly divided into three different categories:

- Supervised learning: in supervised learning, an agent is given a set of labelled data, examples of output/input pairs, and attempts to learn the function which has produced these pairs. In other words, it tries to generalize a rule by looking at (usually a lot of) examples.

- Unsupervised learning: unsupervised learning is fairly similar to supervised learning, except the input data examples are not labelled. The task of the agent is to find the a structure or pattern in the input data by itself, with no explicit feedback given.

- Reinforcement learning: in reinforcement learning, an agent is placed in an environment to fend for itself. At each time step, is receives a percept, and chooses an action. Then, it learns from a series of reinforcements, in the form of rewards or punishments.

Unsupervised learning is not very relevant to this thesis, and will not be discussed further. However, both supervised learning and reinforcement learning will be described in more detail.

2.1.1 Supervised learning

The task of supervised learning can be stated like this: Given a training set of N input/output pairs:

$$(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$$

where each y_j has been generated by a function $y = f(x)$, find a function $h(x)$ which approximates $f(x)$.

Note that both x and y can be of any type, such as text, a number, or an image. The function $h(x)$ is called a hypothesis. Learning is a search through the space of possible hypotheses for one which will generalize well from the examples. The performance of the hypothesis is usually measured by giving it inputs it has not seen before, and testing if it is able to predict the output, or label. This can be done by putting aside some of the example data before training, these examples are called a test set.

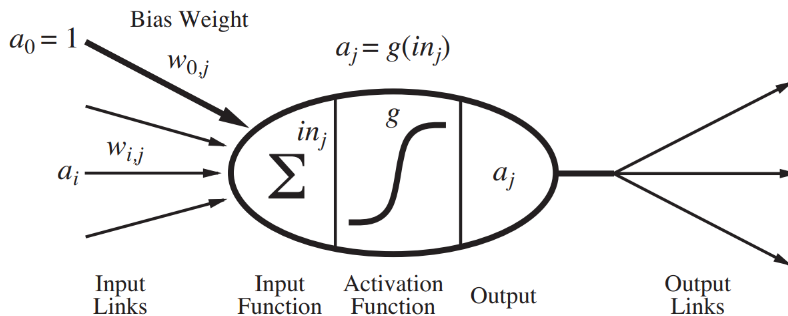


Figure 2.1: Example of a simple model for a neuron. From Russel and Norvig [28].

2.1.2 Artificial neural networks

Artificial neural networks (ANNs) were created as an attempt to mimic how the animal brain functions. They turned out to be a oversimplification, but remain very useful as function approximators. ANNs are called *networks*, because they contain multiple neurons (i.e. nodes) which are connected together. Each node has a number of inputs, and an output (which can be connected to multiple other nodes). When a linear combination of the inputs exceeds some value, the node produces an output. Mathematically, each node j calculates a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i, \tag{2.1}$$

where the index i ranges over all nodes in the previous layer connected to it. Then, an activation function g is applied, producing the output:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{2.2}$$

These neurons are arranged in different *layers*, which can be divided into three broad categories: input layers, hidden layer(s), and output layers. Every neural network has one input layer, where the input data is fed to the network, and one output layer, which produces the resulting output. Networks can have any number of hidden layers, including zero. In a *feedforward* network, nodes in each layer are connected to nodes in the next layer, see figure 2.2. If there are loops in the ANN, it is a *recurrent* network, but most of the ANNs discussed in this thesis

are feedforward networks.

Each node implements a linear classifier, but the network as a whole can approximate nonlinear functions as well. Cybenko [7] showed that two hidden layers can represent any function, and later proved that a single hidden layer is enough to represent any continuous function [8], an early version of the universal approximation theorem. This is done by varying the different weights of the nodes in the network, collectively called the parameters, and denoted as $\boldsymbol{\theta}$. Then, the problem is finding a combination of weights which will produce a function closest to the target function the network is to approximate.

Gradient descent

In the context of neural networks, *training* refers to solving the problem described above. The weights of the neural network are updated incrementally to increase its accuracy and performance on the task. There are multiple ways of doing this, but by far the most popular is a numerical optimization algorithm known as gradient descent (also known as steepest descent). Gradient descent uses the gradient of the function to be minimized (in this case the loss function of the neural network) to iteratively search for the minimum. Given a loss function $Loss(\boldsymbol{\theta})$, the parameter updates are,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} Loss(\boldsymbol{\theta}), \tag{2.3}$$

where α is the step size, or *learning rate*. It can either be a constant, or decrease over time to ensure convergence. This means that each weight is updated by,

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\boldsymbol{\theta}). \tag{2.4}$$

A popular way of performing gradient descent consists of randomly selecting training examples, and taking a step after each one. This is known as stochastic gradient descent (SGD).

2.1.3 Deep learning

Deep learning is a class of machine learning algorithms which employ multiple layers of nonlinear processing units in order to learn. Most modern deep learning models are based on the artifical neural networks discussed above, and this chapter

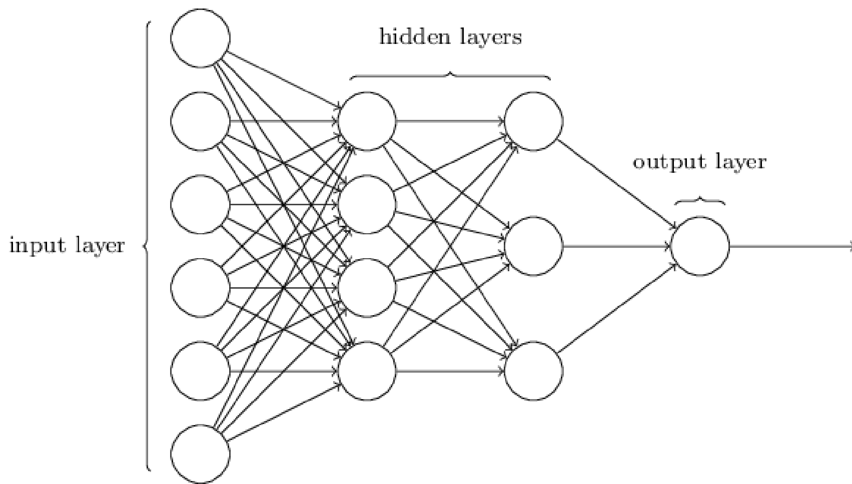


Figure 2.2: Simple example showing a deep neural network with four layers. From Nielsen [25].

will focus on these architectures, often referred to as deep neural networks (DNNs, see figure 2.2). DNNs have become quite popular lately, and are responsible for some of the most impressive abilities of modern machine learning systems.

Deep neural networks contain multiple hidden layers between the input and output layers, and while there is no definition for how many hidden layers are "required", ANNs with two or more hidden layers are usually considered to be DNNs. Most "shallow" machine learning techniques require carefully engineered *features* (individual measurable characteristics found in the training data) to perform well. In other words, humans have to decide what parts of the data to give to the algorithm as input, to ensure that it learns effectively. One of the biggest advantages of deep learning, is that feature engineering is not required. A deep neural network can take raw data as input, and then learn which features are the most relevant on its own. This is due to the multi-layered architecture of the DNN, each layer can create abstractions which are then fed to the next layer. In this way, each layer can build more and more complex features. As an example, it can be useful to think of the case of computer vision: the early layers might learn that lines are useful features to detect, later layers might combine these to form edges and corners, and even later layers might build even more complicated features.

Because these networks are very large, they also require a lot of training data to be effective, compared to other methods, and this is one of the drawbacks of

deep learning. Naturally, this causes training to be very computationally expensive. However, deep learning methods can keep improving with more training data where other methods reach a saturation threshold (when more training does not improve performance). Moreover, some theoretical results suggest that deep networks are more powerful than shallow networks by nature [26].

Back-propagation

As described in chapter 2.1.2, training neural networks is usually done using SGD, and this is also the case for deep neural networks. However, with the addition of hidden layers, a complication worth noting arises. While the error $\boldsymbol{y} - \boldsymbol{h_\theta}$ (where $\boldsymbol{y}$ is the correct output and $\boldsymbol{h_\theta}$ is the output of the neural network hypothesis) at the output layer is easily obtained, the error in the hidden layers are not as straight-forward to find. However, it turns out that the error in the output layer can be propagated backwards to the hidden layers, and used to calculate the weight updates. The weight-update rule for an output node k is given as,

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k, \tag{2.5}$$

where

$$\Delta_k = Error_k \times g'(in_k), \tag{2.6}$$

where $Error_k$ is the error in output node k, g is the activation function, and in_k is the weighted sum as shown in equation 2.1. Then, we want to propagate this modified error Δ_k backwards to all the nodes connected to output node k. Intuitively, the idea is that each hidden node j is responsible for some part of this error, proportional to the strength of the connection between node j and k. The error is propagated in the following way,

$$\Delta_j = g'(in_j) \times \sum_k w_{j,k} \Delta_k, \tag{2.7}$$

which makes the weight-update rule for the hidden layers identical to the output layer,

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j. \tag{2.8}$$

For a more mathematically detailed explanation and derivation of back-propagation, see Russel and Norvig [28], chapter 18.7.4.

2.2 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning focusing on how agents can learn how to act in a certain environment so as to maximize some cumulative reward. The agent is often initialized with a blank slate, *tabula rasa*, with no information about what it should do or how the environment works, and then learns by interacting with it repeatedly. RL can be divided into two problems:

- **Prediction**, where RL is used to learn the *value function* (explained in chapter 2.2.2) for a given policy. This is also known as *policy evaluation*.
- **Control**, where RL is used to learn a policy which maximizes the reward, which might include prediction. For simplicity, this is what "reinforcement learning" will refer to in this thesis unless otherwise specified.

The task of the RL agent is to figure out which actions to perform in the different states it can encounter, and it does this by observing how the environment responds to different actions, both in terms of state change, and reward received. In some approaches, the agents attempts to learn the dynamics of the environment in order to find the best policy, while other methods simply attempts to learn the value of each state and/or action directly. As is usual in machine learning methods, the agent learns how to improve its performance iteratively. When deep neural networks are used as function approximators in the RL agent design, it is called *deep* reinforcement learning (DRL).

The reward is the ultimate measurement of how good or bad a state or action is, but the RL agent can take into account the possibility of potential future rewards in potential future states when estimating the value of an action or state. One thing to note about rewards, is that they can be intermittent, and therefore, temporally delayed. Take the game of chess as an example: the agent might perform hundreds of interactions with the environment (moves) before receiving the reinforcement at the end of the game. This creates a problem for the agent: how does it know which of the moves were important to win the game? This is known as the credit assignment problem.

In control literature, reinforcement learning is often called *approximate dynamic programming*, or *neuro-dynamic programming*.

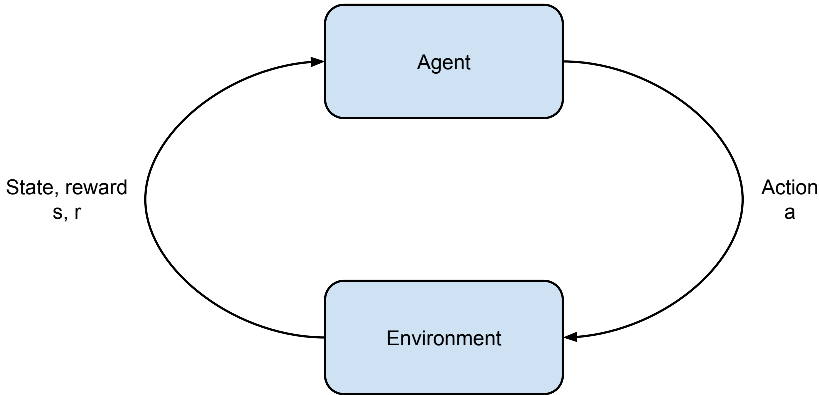


Figure 2.3: Illustration showing how the agent interacts with the environment.

2.2.1 Problem formulation

One characteristic which separates RL from other machine learning paradigms, is that time is an important factor. The RL agent interacts with the environment sequentially over time, and at each time step t, the following happens: the agent receives a state s_t from the state space S, then selects an action a_t from the action space A, according to a policy, which is a rule used by the agent to decide what actions to take, i.e. a mapping from the state s_t to actions a_t. The policy might be deterministic, in which case it is usually denoted as $\mu(s_t)$, or stochastic, in which case it is usually denoted as $\pi(a_t|s_t)$. Then, the agent receives a percept, which contains both a scalar reward r_t, and the next state, s_{t+1} (see figure 2.3). If the environment is episodic, the process continues until it reaches a terminal state, and can be repeated multiple times. The return value is the accumulated reward, often discounted by a *discount factor* $\gamma \in [0, 1]$,

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t, \tag{2.9}$$

where τ is a *trajectory*, which is a sequence of states and actions in the environment,

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.10}$$

The goal of the agent is to optimize its policy so as to maximize the expectation

of the long term return *in each state*, that is, maximize

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{2.11}$$

in each time step t. In the case of deep RL, there are *parameterized* policies, which are policies whose outputs depend on a set of parameters (e.g. the weights and biases of a neural network, as discussed in chapter 2.1.2), which can be adjusted using some optimization algorithm. These parameters are often denoted by $\boldsymbol{\theta}$, and the associated policy by $\pi_{\boldsymbol{\theta}}$. The optimization aims at finding the optimal policy, denoted by π^*, but this is often infeasible, so an approximation is acceptable.

Markov decision process

The Markov decision process (MDP) provides a mathematical framework for modeling the environment which the RL agent is interacting with, as described above. An MDP consists of the following:

- A set of states, S, which contains all possible states of the environment
- A set of actions, $A(s)$, which contains all possible actions in each state
- A transition model, $P(s'|s,a)$, which denotes the probability of reaching state s' when performing action a in state s
- A reward function $R(s)$, which is the reward perceived in state s

The state transitions of the environment are assumed to obey the *Markov property*, which means the probability of reaching state s' only depends on s, and not on the history of any earlier states.

If the complete state of the environment is available to the agent through the state, the environment is *fully observable*. If only a partial observation is available, the environment is *partially observable*.

Different environments allow different kinds of actions, and the set of all valid actions in a given environment is called the *action space*. In chess, for example, the action space of a given state could be the set of legal moves available. This is an example of a discrete action space, where a finite number of moves are available to the RL agent. Other environments can have continuous actions spaces, such

as real-valued inputs or parameters. The different environments explored later in this thesis have continuous action spaces.

2.2.2 Value function

The value function $V_\pi(s)$ is the expected return value of state s when following policy π,

$$V_\pi(s) = \mathbb{E}[R_t | s_t = s] \tag{2.12}$$

In other words, the value function estimates how good a state is. The action value function, also known as the Q-function, $Q_\pi(s, a)$, gives the expected return for performing action a in state s, and then act according to policy π:

$$Q_\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a] \tag{2.13}$$

In both of these value functions, R_t is the return value as defined in equation 2.11. The optimal value function,

$$V^*(s) = \max_\pi V_\pi(s) = \max_\pi \mathbb{E}[R_t | s_t = s], \tag{2.14}$$

gives the expected return when starting in state s, and acting according to the optimal policy π^* afterwards. The optimal action-value function,

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a], \tag{2.15}$$

gives the expected return when starting in state s, performing action a, and then acting according to the optimal policy π^* afterwards. There is an important connection between the optimal action-value function and the optimal action: the optimal policy in s will select the action which maximizes the expected return when starting in s. Therefore, if $Q^*(s, a)$ is known, the optimal action $a^*(s)$ can be obtained directly,

$$a^*(s) = \arg\max_a Q^*(s, a). \tag{2.16}$$

Bellman equation

A Bellman equation is a necessary condition for optimality in dynamic programming (DP). Dynamic programming is an optimization method which simplifies a complicated problem by recursively breaking it down into simpler sub-problems,

which can then be solved. Russel and Norvig [28] gives a formulation applied to the utility of a state,

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.17}$$

In the RL and MDP context, the underlying concept of the Bellman equation can be described intuitively as follows: the value of a state is the immediate reward perceived in that state, plus the value of the next state. Both the value function and Q-function described above obey the Bellman equation,

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)(r + \gamma V_\pi(s')) \tag{2.18}$$

$$Q_\pi = \sum_{s'} P(s'|s,a)[r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')] \tag{2.19}$$

Advantage function

Sometimes, it is not necessary to know how good an action is in the absolute sense, and knowing how much better it is than other actions on average is satisfactory. This can be done using an *advantage function* $A_\pi(s,a)$, which describes how good choosing a specific action a is, compared to simply following the policy π,

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{2.20}$$

This is very useful in policy optimization methods, which will be discussed later, in section 2.2.6.

2.2.3 Exploration vs. exploitation

An important problem in reinforcement learning is the tradeoff between exploration and exploitation. To achieve high rewards, the agent has to choose actions it has tried before and know to be good, but to discover these actions, it has to choose actions that have not been tried. Exploration refers to the RL agent exploring the environment to collect more information, and exploitation means simply following the current best policy to gain as much reward as possible (the

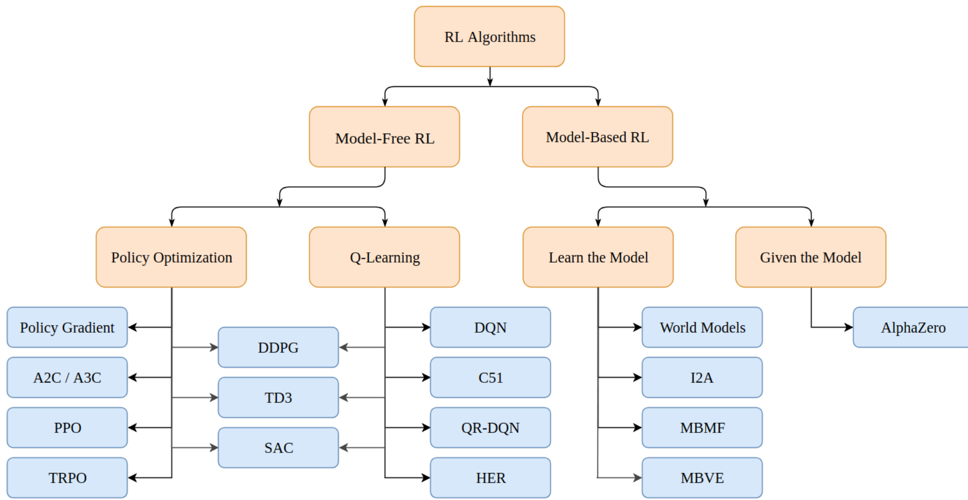


Figure 2.4: A non-exhaustive taxonomy of modern RL algorithms. Figure by Achiam [2].

latter is known as a *greedy* agent). In other words, short-term rewards have to sacrificed to be able to find a good policy in the long run.

Silver [30] proposes some solutions, such as naive exploration, optimistic initialization, optimism in the face of uncertainty, probability matching and information state search.

One of the most popular methods used is a version of naive exploration called ϵ-greedy, which is fairly straight-forward: Let $\epsilon \in (0, 1)$. The agent selects a random action with a probability of ϵ, and a greedy action $a = \arg\max_a Q_\pi(s, a)$ according to the current best policy π with a probability of $(1 - \epsilon)$.

2.2.4 Model-free vs. model-based RL

A very important distinction to make with RL algorithms, is whether its approach is *model-based* or *model-free*.

In model-based algorithms, the agent either has access to a complete model of the environment, or attempts to learn it through interaction. In this context, a model of the environment refers to a function which predicts state transitions and rewards. A major advantage of having a model is that it allows the agent to plan

ahead, and see what would happen for a range of different actions it can perform in its current state, and then comparing these outcomes when deciding which action to take. This can lead to a substantial improvement in *sample efficiency*, compared to algorithms that do not use a model. One example of this is AlphaZero [32].

Unfortunately, a ground-truth model of the environment is rarely available in most RL problems. If a model-based approach is to be used without a perfect model available, the agent has to learn the model itself from experience, which raises some challenges. The biggest challenge is usually that the agent, by design, exploits any bias found in the learned model, which might cause poor performance in the actual environment.

While model-free algorithms forego the potential gains in sample efficiency, they tend to be easier to implement and tune. At this time, model-free methods are more popular and have been more extensively developed and tested than model-based methods [2]. This thesis focuses mainly on the model-free approaches, but some examples of model-based RL are discussed below. Later subsections will discuss model-free methods.

Learning in model-based RL

The most basic approach is to not learn a policy at all, and employ pure planning methods such as model predictive control (MPC), widely used in control theory. In MPC, an optimal plan with respect to the model is computed over some finite time-horizon, and then discarded in the next time step, and computed again. MPC has been combined with learned models to improve sample efficiency in locomotion tasks [24].

A slightly more advanced approach than pure planning involves simultaneously following and learning a policy, by employing some planning algorithm, such as Monte Carlo Tree Search (MCTS) [15], α-β Search and greedy search. Potential actions are generated by sampling its current policy, $\pi_\theta(a|s)$, and comparing different plans. This is called *expert iteration*, because the planning algorithm is able to choose actions which are better than what the policy suggests, making it an "expert" relative to the policy alone. The policy is then updated to produce actions similar to the planning algorithm. Examples of using this approach include AlphaZero [32] and ExIt [4].

A third approach called data augmentation for model-free methods, uses a model-free RL algorithm to learn either a policy or an action-value function, but uses either augmented or purely fictional experiences for training the agent. One example of augmenting real experiences with constructed ones is Model-Based Value Estimation [10], while Recurrent World Models [11] trains an agent entirely inside of a self-generated world, and transfers the learned policy back into the actual environment.

Yet another approach is embedding planning loops into policies. The planning part is integrated directly into the policy, making complete plans available as additional context for the policy while learning the policy itself with a model-free algorithm. The idea is that the policy can learn to choose whether or not to use the plans, and how. This combats the problem of model bias, because the policy can learn to ignore the plan in states where the model is unhelpful. Imagination-Augmented Agents (I2A) [37] is an example of one such architecture.

For a more in-depth discussion of model-based RL and planning, see Sutton and Barto [33], chapter 8. For a more in-depth summary of different implementations, see Li [16], chapter 6.

2.2.5 Temporal difference learning

Temporal difference (TD) learning is a very central topic in reinforcement learning. TD learning methods are model-free, and can learn directly from experiences without a model of the environment. These methods can also employ *bootstrapping*, which means estimates are updated in part by using other learned estimates, without waiting for the actual outcome. TD learning often refers to the prediction problem with an update rule for the value function given as,

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s)), \tag{2.21}$$

where α is the learning rate, and γ is the discount factor. Note the part inside of the parenthesis, this is known as the *TD error*,

$$\delta_t = r + \gamma V(s') - V(s), \tag{2.22}$$

and arises in various forms in many areas of reinforcement learning.

The TD learning method for prediction is used in two different methods for doing

TD control, which are discussed below. The main difference is that one is *on-policy*, and the other is *off-policy*.

SARSA

SARSA gets its name from the quintuple representing a transition from one state-action pair to the next: (s, a, r, s', a'). It works by taking the principle of TD prediction, and applying it to learning an action-value function $Q(s, a)$, instead of a value function. It is an on-policy method, because it estimates $Q_\pi(s, a)$ *for the current policy* π, and simultaneously update the π greedily with respect to the estimated Q_π. The action-value update rule is given as

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)), \tag{2.23}$$

and is used after every transition to a nonterminal state s. If a state s' is found to be terminal (usually available to the agent via its percept), $Q(s', a')$ is set to zero. It can be shown that SARSA converges to an optimal action-value function and policy when all state-action pairs are visited an infinite number of times, and the policy converges to be purely greedy [33].

Q-learning

In 1989, Watkins [36] presented one of the early breakthroughs in reinforcement learning, namely the off-policy TD control algorithm Q-learning. Its update rule is given as

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)). \tag{2.24}$$

The algorithm is regarded as off-policy, because the learned action-value function directly approximates the optimal action-value function Q^*, independent of the actual policy π the agent follows. In practice, this means that the update can use transition data from any point during training, regardless of how the agent was behaving when the data was obtained. The policy still determines which action-state pairs are visited and updated, but the only requirement for convergence to the optimal policy is that all pairs continue to be updated [33]. The actual policy is learned via the connection between Q^* and π^*, as the actions chosen by the

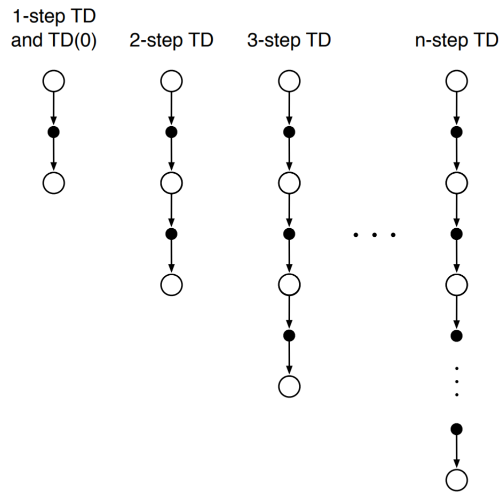


Figure 2.5: Illustration of different TD methods. White circles are states, black circles represent actions. Figure (slightly modified) from Sutton and Barto [33].

agent are given by

$$a(s) = \pi(s) = \arg\max_a Q(s, a). \tag{2.25}$$

Multistep bootstrapping

The algorithms described above only use a single look-ahead step when calculating the return, and are sometimes referred to as TD(0), SARSA(0) and Q(0). They also have variants with multistep returns, which utilize additional steps in their estimations. The n-step update, the state-value function is updated towards the n-step return, defined as

$$r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(st + n)[16]. \tag{2.26}$$

Bootstrapping methods are usually fast to learn, and enable online, continual learning. For a more in-depth discussion of multistep bootstrapping, see Sutton and Barto [33], chapter 7.

2.2.6 Policy optimization

The value-based methods discussed thus far optimize value functions first, and then use them to derive the policies. An alternative to this is *policy optimization*. These methods represent a policy explicitly as $\pi_{\boldsymbol{\theta}}(a|s)$, and optimize their parameters $\boldsymbol{\theta}$ either directly, by using gradient ascent on the objective function $J(\pi_{\boldsymbol{\theta}})$, or indirectly, by maximizing some local approximation of $J(\pi_{\boldsymbol{\theta}})$. Compared to value-based methods, these methods usually have better convergence properties, are effective in high-dimensional or continuous action spaces, and can learn both stochastic and deterministic policies [16]. On the other hand, policy-based methods sometimes converge to local optimum, can be inefficient to evaluate, and encounter high variance [30].

Note that $J(\boldsymbol{\theta}) = J(\pi_{\boldsymbol{\theta}})$ is sometimes used for simplicity when $\boldsymbol{\theta}$ denotes the parameters of the function approximation of the policy π. Using gradient ascent, the parameters are updated in the following way:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t). \tag{2.27}$$

The gradient of the objective function, $\nabla_{\boldsymbol{\theta}}$, is called the policy gradient, and methods which optimize this way are collectively called *policy gradient methods*. Using this update rule, the problem now becomes finding an expression of the policy gradient which can be computed numerically. The objective function itself is the expected long term return when following some trajectory dictated by the policy:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[R(\tau)]. \tag{2.28}$$

For a differentiable policy $\pi_{\boldsymbol{\theta}}(a|s)$, the gradient of the policy can be calculated analytically:

$$\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) = \pi_{\boldsymbol{\theta}}(a|s) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} = \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) \tag{2.29}$$

Using this, and the policy gradient theorem [34], we get

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi_{\boldsymbol{\theta}}}(s,a)]. \tag{2.30}$$

The algorithm REINFORCE [38] is a simple example of using this method. It utilizes the return R_t as an unbiased sample of the action-value function $Q(s_t, a_t)$,

which results in the policy gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) R_t \tag{2.31}$$

Using this gradient, each parameter update will be proportional to the return value R_t, multiplied with a vector $\nabla \pi_{\boldsymbol{\theta}}(a_t|s_t)$, which is the gradient of the probability of choosing the action which was chosen, divided by the probability of choosing that action, $\pi_{\boldsymbol{\theta}}(a_t|s_t)$. Intuitively, the gradient vector will be the direction in parameter space which will lead to the largest increase of the probability of choosing action a_t in state s_t, which means the update will lead to an increase in the parameter vector in this direction which is proportional to the return value, causing the policy to favor actions with high return values. The division causes this increase to also be inversely proportional to the current probability of choosing this actions, which will normalize it to avoid giving frequent actions an advantage over infrequent actions, regardless of return value.

By generalizing the policy gradient theorem slightly, a baseline $b(s_t)$ can be included for comparison of the action value, by subtracting it from the action-value function. This is to reduce the variance of the gradient estimate. Using the baseline, the policy gradient becomes

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)(Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) - b(s_t)) \tag{2.32}$$

The baseline can be any function, as long as it does not depend on the action a. Policy optimization often involves learning an approximation of the the on-policy value function $V_{\pi_{\boldsymbol{\theta}}}(s)$, which is then used in the update of the policy $\pi_{\boldsymbol{\theta}}$. Pseudocode of the REINFORCE algorithm with a baselines is shown in algorithm 1.

Note that if the value function is used as a baseline, we end up with the advantage function:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)(Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) - V_{\pi_{\boldsymbol{\theta}}}(s_t)) \tag{2.33}$$

$$= \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) A_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \tag{2.34}$$

Empirically, using this baseline results in faster and more stable learning [2].

Algorithm 1 REINFORCE (with baseline)

Input: policy $\pi(s|\boldsymbol{\theta}_\pi)$, state-value function $v(s|\boldsymbol{\theta}_v)$
Parameters: step sizes, $\alpha_\pi > 0$, $\alpha_v > 0$
Initialize policy parameters $\boldsymbol{\theta}_\pi$ and state-value parameters $\boldsymbol{\theta}_v$
for episode = 0, M-1 **do**
  Generate an episode $(s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T)$, following $\pi_{\boldsymbol{\theta}_\pi}$
  for t = 0, T-1 **do**
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G_t - v(s_t|\boldsymbol{\theta}_v)$
    $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \alpha_v \delta \nabla_{\boldsymbol{\theta}_v} v(s_t|\boldsymbol{\theta}_v)$
    $\boldsymbol{\theta}_\pi \leftarrow \boldsymbol{\theta}_\pi + \alpha_\pi \gamma^t \delta \nabla_{\boldsymbol{\theta}_\pi} \log \pi(s_t|\boldsymbol{\theta}_\pi)$
  end for
end for

Actor-critic

Pure policy gradient methods tend to learn slowly due to estimates with high variance, and are inconvenient to implement for online problems, however, the TD methods discussed in chapter 2.2.5 can be used to combat these problems. Actor-critic methods combine these two approaches to learn both a policy and a state value function simultaneously, and use the value function for bootstrapping. Note that learning a value function for use as a baseline is not an example of actor-critic, because the value function is not used for bootstrapping. The policy is the *actor*, and controls how the agent behaves, while the learned value function is the *critic*, and measures how good an action is, i.e. criticizes the actions chosen by the actor.

The critic value function can be learning using some TD method as described above. The actor policy is learned by using the policy gradient, with the estimated value function being used in the calculation of the TD error, replacing the return:

$$\begin{aligned} \boldsymbol{\theta}^\pi_{t+1} &= \boldsymbol{\theta}^\pi_t + \alpha\big(r_{t+1} + \gamma v(s_{t+1}|\boldsymbol{\theta}^v) - v(s|\boldsymbol{\theta}^v)\big)\nabla_{\boldsymbol{\theta}^\pi} \log \pi(s|\boldsymbol{\theta}^\pi_t) && (2.35) \\ &= \boldsymbol{\theta}^\pi_t + \alpha\delta\nabla_{\boldsymbol{\theta}^\pi} \log \pi(s|\boldsymbol{\theta}^\pi_t) && (2.36) \end{aligned}$$

Note that the notation $\boldsymbol{\theta}^\pi$ is the same as $\boldsymbol{\theta}_\pi$. An illustration of the actor-critic method is shown in figure 2.6, and pseudocode for a simple one-step actor-critic algorithm is shown in algorithm 2.

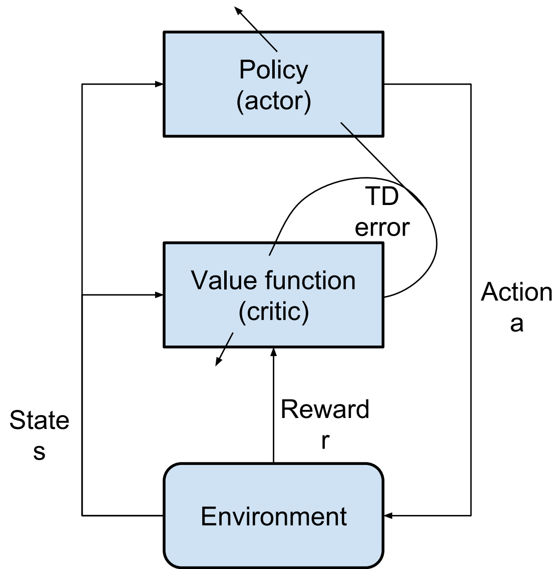


Figure 2.6: Illustration showing how the actor and critic interact with each other and the environment.

Algorithm 2 Actor-critic

Input: policy $\pi(s|\boldsymbol{\theta}_\pi)$, state-value function $v(s|\boldsymbol{\theta}_v)$
Parameters: step sizes, $\alpha_\pi > 0$, $\alpha_v > 0$
Initialize policy parameters $\boldsymbol{\theta}_\pi$ and state-value parameters $\boldsymbol{\theta}_v$
for episode = 0, M-1 **do**
  Initialize first state s_0
  for t = 0, T-1 **do**
    Perform action a_t according to policy $\pi(s_t|\boldsymbol{\theta}_\pi)$, receive percept s_{t+1}, r_t
    $\delta \leftarrow r_t + \gamma v(s_{t+1}|\boldsymbol{\theta}_v) - v(s_t|\boldsymbol{\theta}_v)$
    $\boldsymbol{\theta}_v \leftarrow \boldsymbol{\theta}_v + \alpha_v \delta \nabla_{\boldsymbol{\theta}_v} v(s_t|\boldsymbol{\theta}_v)$
    $\boldsymbol{\theta}_\pi \leftarrow \boldsymbol{\theta}_\pi + \alpha_\pi \gamma^t \delta \nabla_{\boldsymbol{\theta}_\pi} \log \pi(s_t|\boldsymbol{\theta}_\pi)$
  end for
end for

2.2.7 Deep deterministic policy gradient

Deep deterministic policy gradient (DDPG) is a model-free, actor-critic algorithm with continuous action spaces, presented by Lillicrap et al. [19] in 2015. It is the main algorithm chosen for study in this thesis, mainly due to being adapted specifically for environments with continuous action spaces, which most physical control tasks have, and because of its high performance. DDPG is an extension of two other algorithms, Deep Q-Networks (DQN) [22, 23] and Deterministic Policy Gradient (DPG) [31]. Specifically, it utilizes the experience replay and target network techniques from DQN, and uses actor-critic with a deterministic policy as in DPG. DDPG concurrently learns an action-value function and a policy, by using off-policy data and the Bellman equation to learn the action-value function, and then uses the action-value function to learn the policy. The different techniques and methods will be discussed in-depth below. Pseudocode for DDPG is shown in algorithm 3.

The Bellman equation for the action-value function can be written as,

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim E}\left[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1}\sim\pi}[Q_\pi(s_{t+1}, a_{t+1})]\right], \tag{2.37}$$

where $s_{t+1} \sim E$ means that the transition is sampled from the environment E, and $a_{t+1} \sim \pi$ means that an action is sampled from the policy π. If the policy is stochastic, it is usually denoted μ, and the inner expectation of the Bellman equation can be avoided,

$$Q_\mu(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim E}[r(s_t, a_t) + \gamma Q_\mu(s_{t+1}, \mu(s_{t+1}))] \tag{2.38}$$

Because this expectation only depends on the environment, Q_μ can be learned *off-policy*, by using transitions generated by a different stochastic policy β. Using the greedy policy from Q-learning, $\mu(s) = \arg\max_a Q(s, a)$, and representing the Q-function as a function approximator parameterized by $\boldsymbol{\theta}_Q$, the mean-squared Bellman error (MSBE) can be used as a loss function. That is, optimization is done by minimizing

$$L(\boldsymbol{\theta}_Q) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_t\sim E}\left[(Q(s_t, a_t)|\boldsymbol{\theta}_Q) - y_t)^2\right], \tag{2.39}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\boldsymbol{\theta}_Q), \tag{2.40}$$

and ρ^β is the discounted state visitation distribution for the policy β. y_t is often called the *target value*.

Using deep neural networks as function approximators, DQN [22, 23] is capable of solving problems with high-dimensional observation spaces (i.e. state spaces), but can only handle low-dimensional and discrete action spaces. This is due to the way DQN chooses actions, by maximizing the action-value function (Q-function), which would require an iterative optimization for each time step for a continuous action space. One solution is to discretize the action space of the environment, however, this leads to some problems. First of all, the curse of dimensionality comes into play, especially when considering systems with multiple degrees of freedom, or where fine control is required. The number of different actions to choose from can quickly increase to a number where using DQN directly is infeasible. Additionally, discretization of continuous action spaces might lead to loss of valuable information about the action domain structure.

DDPG solves this issue by using an actor-critic approach based on the DPG algorithm [31]. The actor is a parameterized approximation of a deterministic policy, $\mu(s|\boldsymbol{\theta}_\mu)$, and the critic is a parameterized approximation of the action-value function, $Q(s,a|\boldsymbol{\theta}_Q)$, and they are both represented by deep neural networks. The critic, $Q(s,a)$ is learned using the Bellman equation as in Q-learning (see above), while the actor is learned by using the policy gradient. Silver et al. [31] showed that for a deterministic policy, the policy gradient is simply the expected gradient of the action-value function:

$$\nabla_{\boldsymbol{\theta}_\mu} J \approx \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_{\boldsymbol{\theta}_\mu} Q(s_t,\mu(s_t|\boldsymbol{\theta}_\mu)|\boldsymbol{\theta}_Q)\right] \tag{2.41}$$

$$= \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_a Q(s_t,\mu(s_t)|\boldsymbol{\theta}_Q)\nabla_{\boldsymbol{\theta}_\mu}\mu(s_t|\boldsymbol{\theta}_\mu)\right] \tag{2.42}$$

Recall from chapter 2.2.6 that in the stochastic case, the policy gradient integrates over both state and actions spaces. The deterministic policy gradient, however, only integrates over state space, and can therefore be estimated much more efficiently than the stochastic policy gradient.

Replay buffers

Most optimization algorithms used for training neural networks assume that the samples used are independently and identically distributed. In reinforcement learning, where the samples are generated from sequentially interacting with the

environment, this assumption does not hold. Additionally, to take advantage of hardware optimizations, it is essential to learn in mini-batches, rather than online [19].

One way to deal with this issue, is to use a experience replay buffer, which was introduced by Lin [20], and used in DQN [22]. Following some policy, different transition tuples $e_t = (s_t, a_t, r_t, s_{t+1})$ are generated, and saved in a cache $\mathcal{R}_t = \{e_1, ..., e_t\}$. This set contains a finite amount of previous experiences, and at each time step, both the actor and critic are updated using a uniformly sampled mini-batch of tuples from this buffer, yielding uncorrelated samples for training. The implementation from the original DDPG paper [19] used a replay buffer size of 10^6. When the buffer is full, old samples are discarded to make room for new ones. Note that this means that the updates might use old transitions generated by an outdated policy. However, this is not a problem, because DDPG is *off-policy*. This is due to the nature of Q-learning, the optimal action-value function $Q^*(s, a)$ should satisfy the Bellman equation for all possible transitions, and all transitions are therefore useful for training, regardless of how good the action chosen was.

Another advantage of using replay buffers, is that each step of experience e_t can be used in multiple weight updates, which allows for greater data efficiency. Furthermore, learning on-policy means that the current policy parameters determine the next transition, which produces the next training sample used for updating the parameters. This might cause unwanted feedback loops and lead to the agent getting stuck in local minimum, or even divergence [23].

Target networks

Another trick used in DQN to achieve stable learning with the deep neural networks, is the use of target networks. The critic network $Q(s, a|\boldsymbol{\theta}_Q)$ is being updated while also being used in the target value (see equation 2.40) of the MSBE loss, which means the parameter update depends on the parameters $\boldsymbol{\theta}$ which are being updated. This causes the Q update to be prone to divergence, and makes learning unstable. To avoid this, copies of both the actor and critic networks are created, and denoted $\mu'(s|\boldsymbol{\theta}_{\mu'})$ and $Q'(s, a|\boldsymbol{\theta}_{Q'})$. They are used for calculating the target values, hence their names. The idea is that the weights of the target networks are initialized as copies of the weights of the actor and critic networks, but updated more slowly.

In DQN, the main network is cloned to create the target network every C updates, and then used for C updates, where C is a fixed size number. In DDPG, a slightly different approach is used. The target networks are updated as often as the main networks, but with "soft" updates:

$$\boldsymbol{\theta}_{Q'} \leftarrow \tau\boldsymbol{\theta}_Q + (1-\tau)\boldsymbol{\theta}_{Q'} \tag{2.43}$$

$$\boldsymbol{\theta}_{\mu'} \leftarrow \tau\boldsymbol{\theta}_\mu + (1-\tau)\boldsymbol{\theta}_{\mu'} \tag{2.44}$$

where $\tau \in (0,1)$ is a hyperparameter, usually with a small value (e.g. 0.001). This causes the target networks to change slowly, which slows learning, but greatly improves learning stability [19].

Exploration

As discussed earlier, the tradeoff between exploration and exploitation is an important problem in reinforcement learning, especially in continuous action spaces. Because DDPG is off-policy, the problem of exploration can be completely seperated from the learning algorithm itself. In order to make the DDPG agent explore the environment, noise sampled from a noise process $\mathcal{N}$ is added to the actor policy when selecting an action a during training:

$$\pi(s_t) = \mu(s_t|\boldsymbol{\theta}_\mu) + \mathcal{N} \tag{2.45}$$

This is called action noise. Different noise processes can be used, the original DDPG paper [19] suggests an Ornstein-Uhlenbeck process, which is time correlated. More recent results suggest that uncorrelated, mean-zero Gaussian noise works well, and is often used because it is simpler [2]. When testing the agent, the noise is simply removed from the actor policy, causing it to exploit the information it has gathered as much as possible.

Parameter noise

Another way to ensure exploration, is to introduce noise to the parameters of the neural network representing the policy, instead of adding it to the action output (see figure 2.7). This has been shown to cause more consistent, effective exploration and faster learning [27]. One important difference between action noise and parameter noise, is episode consistency. Consider equation 2.45: the

Algorithm 3 DDPG algorithm, from Lillicrap et al. [19]

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

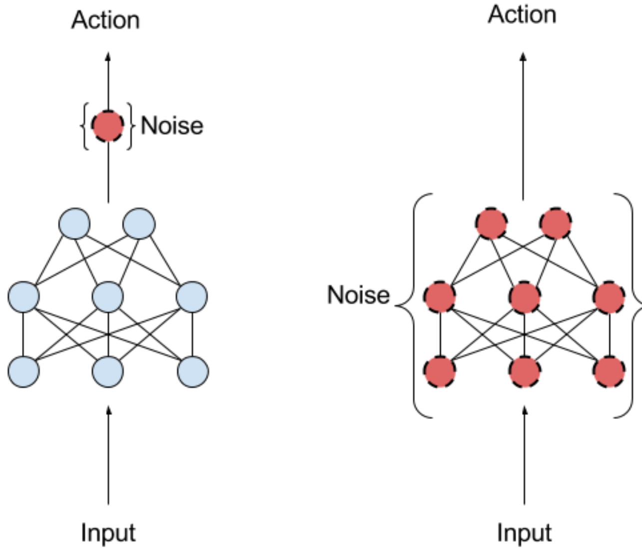


Figure 2.7: Illustration of action noise (left) and parameter noise (right). Figure from the OpenAI blog post on parameter noise[1].

policy μ might be deterministic, but combined with the noise process, it becomes a stochastic policy π, which might choose different actions in the same state s_t during the same episode. In other words, action noise is independent of the state. With parameter noise, perturbations are added to the parameters at the beginning of each episode, which ensures consistency in the choice of action in any fixed state s_t.

Choosing a scale σ for parameter noise is not straightforward, mainly due to the complexity of neural networks. Plappert et al. [27] propose a solution to this problem, by making the noise scale adaptive. Some distance measure $d(\mu, \mu')$ is defined, where μ' and μ is the policy with and without parameter noise, respectively. Some threshold δ is set for this distance measure, and the parameter noise is then adaptively increased or decreased based on the distance measure, as such:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d(\mu, \mu') \leq \delta, \\ \alpha^{-1}\sigma_k & \text{otherwise} \end{cases} \tag{2.46}$$

where α is a scaling factor, usually slightly bigger than 1.

[1] https://blog.openai.com/better-exploration-with-parameter-noise/

For off-policy algorithms such as DDPG, application of parameter noise is as simple as using action noise, because all generated transitions are useful. The noise is simply used during training to cause exploration, and the data is used to train a noise-free network afterwards. For a more detailed description of adaptive parameter noise, see Plappert et al. [27].

2.3 DRL algorithm implementations

There are many different open-source repositories available which implement different DRL algorithms and primitives. A considerable part of working with this thesis was spent exploring, testing and comparing different implementations of DRL algorithms, especially the DDPG algorithm. The different implementations were considered with a focus on four main aspects:

- Performance: Different measures such as convergence speed, i.e. how many steps are usually required before a good solution is found, and training time, which is how much time is needed to perform the steps, and the quality of the trained policy, usually dictated by a good exploration policy.

- Functionality: What kind of additional functionality the frameworks provide in addition to the algorithm itself, such as saving and restoring the agents properly, logging of relevant information, etc.

- Code quality: Well written, structured, easy to understand code with useful comments and function/variable names.

- Documentation: Comprehensive and structured documentation which complements the code, with good examples and explanations of functionality.

Because the example environments are rather simple and basic, the performance difference is minimal in terms of solution quality. Combined with the fact that training is fairly well optimized in all of the implementations, the overall performance difference is rather negligible. Therefore, the latter three qualities were weighted more heavily when deciding which repository to use in the final implementation. Below is a brief summary of the different repositories that were tested.

2.3.1 OpenAI Baselines

Baselines [9] is a repository with implementations of multiple reinforcement learning algorithms, created by the company OpenAI. It is under active development, which means breaking changes may happen in future updates.

Performance. The algorithm implementations are high-performing, well optimized, and support parallelization. Baselines implement adaptive parameter noise (see section 2.2.7), which has been shown to improve learning [27].

Functionality. Baselines is a bare-bones repository which only provides the basic necessities for running the algorithms. Most of the algorithms have functionality for saving and restoring, however, the DDPG implementation lacks these, and they are not straightforward to implement. Some logging and plotting functionality exists.

Code quality. The code base is fairly well structured, but rather complex and can be hard to read. It does not follow the Python PEP8 code style [35]. The different algorithms are also implemented differently, and do not have a common interface.

Documentation. Baselines comes with no documentation, except for some basic examples in the `README` file. Some algorithm subdirectories contain more detailed examples and references.

2.3.2 Stable Baselines

Stable Baselines [12] is a fork of OpenAI Baselines which aims to fix many of the issues with documentation and functionality that Baselines has.

Performance. Similar performance to OpenAI Baselines, however, because it is a fork, it lacks some of the newest tricks implemented in the Baselines repository. It is under active development, so this may change. Some algorithms support multi-processing, but the DDPG implementation does not. Stable Baselines also implements adaptive parameter noise [27].

Functionality. All algorithms use a common interface, which implements functionality for saving and restoring agents. Additional functionality for logging and plotting exists.

Code quality. Stable Baselines is a major refactoring of the original Baselines repository. The structure is changed, all algorithm implementations follow the same unified structure and use the same common interface. The code has been improved, and follows the Python PEP8 code style [35]. It is easier to understand and has useful comments to some degree, but also contains more functionality, and some parts are complex and hard to follow.

Documentation. Stable Baselines also adds documentation, which is well structured and easy to use. It includes good examples of most of the functionality.

2.3.3 OpenAI Spinning Up

Spinning Up [2] is an educational resource, which is also created by OpenAI. It contains a repository of short and simple algorithm implementations. Spinning Up is a great resource for anyone looking to learn more about deep reinforcement learning.

Performance. The algorithm implementations have decent performance, but prioritize simplicity, which means they lack certain tricks found in other repositories, such as adaptive parameter noise. Some of the algorithms support parallelization, but the DDPG implementation does not.

Functionality. Includes functionality for saving and restoration, with extended tools for logging and plotting. Also comes with a small utility for running experiments called ExperimentGrid.

Code quality. Spinning Up has the highest code quality of all the repositories gathered here, mostly due to its focus on simplicity. The code is easy to understand, well commented, and all the algorithm implementations are standalone.

Documentation. The Spinning Up documentation is excellent, it is well structured, has good examples, and complements the code nicely. It also doubles as a learning resource for DRL concepts, and contains descriptions and explanations of all the algorithms and some of their underlying theory.

2.3.4 RLlib

RLlib [17] is a library for reinforcement learning which contains both a set of algorithm implementations, and primitives for creating new ones. It is built on top of the distributed execution framework Ray.

Performance. Potentially high throughput with Ray. RLlib does not implement adaptive parameter noise for DDPG.

Functionality. Includes functionality for saving and restoration, and logs information during training. No tools for plotting included. RLlib has primitives which can be used to develop custom RL algorithms, and includes a REST interface in addition to the regular Python interface. Ray also comes with a hyperparameter search tool called Tune [18], which is great for testing algorithms.

Code quality. Decent code quality comparable to OpenAI Baselines. The code structure can be hard to follow, but is manageable. Algorithms use shared primitives and inheritance, which avoid code duplication. Decent readability and comments.

Documentation. Decent, but rather convoluted documentation. Basic functionality is explained, but lacks good examples and in-depth descriptions of functions and parameters. The structure could be clearer.

Chapter 3

Implementation

3.1 Systems

To explore whether DRL could be used for typical control problems and measure their performance, two generic examples of systems with simple dynamics, one of first and one of second order, were chosen rather arbitrarily. This section details the relevant information about these systems.

3.1.1 First order system

The first system is a simple, standard first order system with $K_p = 3$ and $\tau_p = 2$. The Laplace domain transfer function is given as

$$\frac{Y(s)}{U(s)} = \frac{Kp}{\tau_p s + 1} \tag{3.1}$$

The state space model of the system is

$$\dot{x} = Ax + Bu \tag{3.2}$$

$$y = Cx + Du \tag{3.3}$$

$$A = -\frac{1}{\tau_p}, B = -\frac{1}{\tau_p}, C = 1, D = 0 \tag{3.4}$$

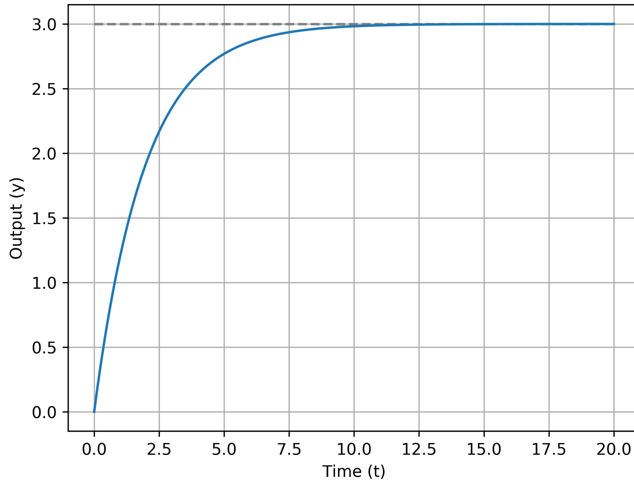


Figure 3.1: Step response of the first order system.

The system step response can be seen in figure 3.1.

3.1.2 Second order system

The second system is a standard second order system. The transfer function is

$$\frac{Y(s)}{U(s)} = \frac{K_p}{\tau_s^2 s^2 + 2\zeta\tau_s s + 1} e^{-\theta_p s} \tag{3.5}$$

The state space model is

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{\tau_s^2} & -\frac{2\zeta}{\tau_s} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{K_p}{\tau_s^2} \end{bmatrix} u(t - \tau_p) \tag{3.6}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u \tag{3.7}$$

With $\zeta = 0.25 < 1$, it is underdamped. The gain is $K_p = 3$ and time constant $\tau_p = 2$. No time delay, $\theta_p = 0$. The step response can be seen in figure 3.2.

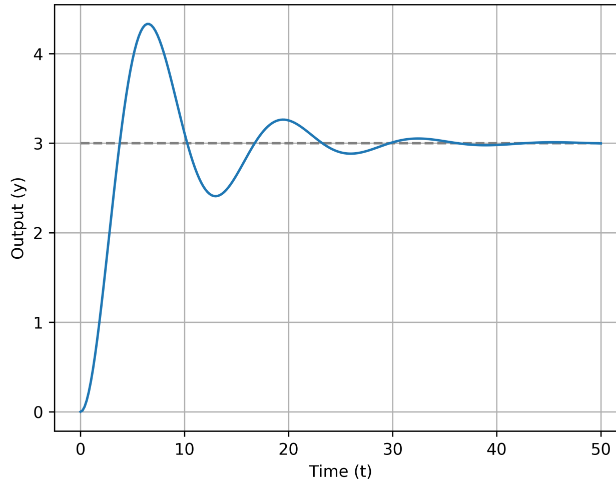


Figure 3.2: Step response of the second order system.

3.2 PID controller

A proportional-integral-derivative (PID) controller was implemented as part of this thesis, both for comparison purposes, and for use in the PID tuning environments which will be explained later. There are open-source implementations of PID controllers available, but most of them implement real-time simulation, which uses actual time differences to calculate Δt. In DRL environments, producing a very large amount of environment interaction experience is required to learn properly, which means using time delays in simulations is out of the question. Therefore, to be able to simulate dynamics with time, but without using actual time delays, a discrete PID controller class with variable Δt (`dt`) was implemented.

PID controllers calculate error values based on a desired setpoint value, and a measurement of the actual output:

$$e(t) = SP(t) - y(t) \tag{3.8}$$

Based on this error value, three different terms are calculated, and combined to produce the system input,

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \tag{3.9}$$

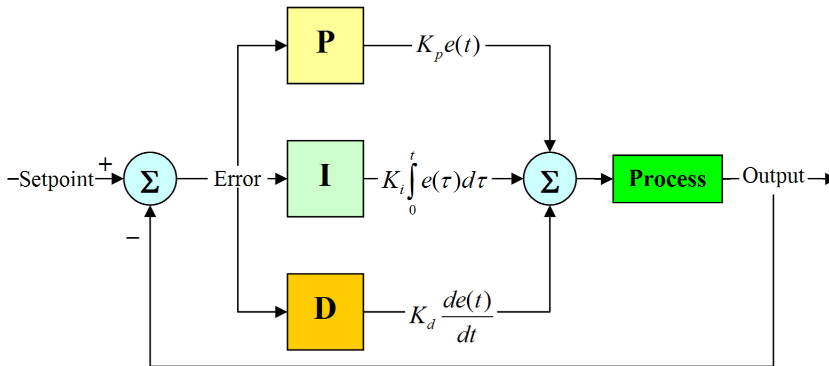


Figure 3.3: Illustration of a PID controller, from Wikipedia[1].

The interface is reasonably straightforward, Δt, setpoint value, output and integral limits, and term coefficients can be set either through the constructor, relevant methods, or directly by accessing the attributes. Output limits and integral limits (also known as windup guard) are optional.

3.2.1 Tuning

Tuning was done using the Ziegler-Nichols method [39], then performing adjustments until a reasonable performance was achieved. For the `SetpointChange-v0` environment, using an average setpoint change, the coefficients are

$$K_p = 6.4, K_i = 0.4, K_d = 3.8 \tag{3.10}$$

3.3 Environments

As discussed in the theory section, the two basic parts of any RL problem are the agent, which is the algorithm implementation, and the environment the agent interacts with. The behaviour of an DRL agent is not programmed explicitly, but implicitly, by carefully designing the environments in which they are trained. The DRL algorithm itself is usually very general, and only impose restrictions on problem details such as whether the action space and observation space are discrete

[1] `https://en.wikipedia.org/wiki/File:PID-feedback-loop-v1.png`

or continuous. We can say that the algorithm implementation dictates *how* the agent learns, but the environment dictates *what* the agent learns. Specifically, the implementation of the system simulation and the reward function shape the resulting agent policy.

3.3.1 Generalization and dynamics randomization

One important feature of any agent is the level of generalization. Agents can be trained to perform a specific task, e.g. control a specific plant or process, in which case it can be appropriate to train the agent using some model of the specific plant. If, however, the agent should be able to control a variety of different processes, it will have to learn to generalize from different models during training.

This can be achieved by introducing randomness into the environment, which can be done in different ways, depending on the desired level of generalization. For example, a general-purpose agent capable of controlling a variety of different systems can be trained by randomizing the system dynamics of the environment. In practice, this can be done by having the simulator initialize the system variables with some degree of randomness, producing a system with different variables with each reinitialization. If an agent is to control a system with a known disturbance, a constant model of the disturbance can be used in the simulation. If the system should be able to handle different types of disturbances, the disturbance process can be initialized with variable dynamics to reflect this. In other words, the relevant aspects of the environment are randomized so as to influence the level of generalization achieved by the agent. This is known as dynamics randomization. Two examples of randomization of a variable are random sampling and adding noise.

All of the environments utilize some form of dynamics randomization. The initial state is always initialized with some small random deviation, and other environment-specific variables are also randomized.

3.3.2 Observation contents

With each interaction with an environment, the agent receives an observation which describes the current state of the environment. The agent behaviour is dictated by a policy which maps these states to actions, so what information is made available to the agent through the observation is of vital importance for its

performance. Providing useful state information and transformations can help an algorithm learn better policies faster. However, one of the reasons to use deep neural networks as function approximators is that they are able to learn useful abstractions from the input by itself, and one of the goals of DRL in general is to design agents which can learn from raw input data, without the need for feature engineering. The environment observations are limited to basic state values and the error (see the implementations in section 3.3.6 for more details).

3.3.3 Reward function

The reward function is the measure the DRL agent uses to gauge its own performance. The policy is optimized so as to maximize the accumulated returns from this function, which means that the design of the reward function essentially dictates the behaviour of the agent. This can be used both to reward desirable behaviour, and to punish undesirable behaviour. Standard control objectives such as minimizing or limiting the rise time and overshoot, rejecting disturbance and constraining input and input change can be accomplished by including relevant terms in the reward function. Rise time can be minimized by for example using the error (see below), overshoot can be limited by punishing the state value being higher than the setpoint, and input and input change can be added as costs directly to achieve preferential behaviour. The basic reward function in the constructed environments is simply

$$r(s_t, a_t) = -|e(s_{t+1})|, \tag{3.11}$$

where $e(s_t)$ is the error at time t, given by

$$e(s_t) = SP_t - y_t. \tag{3.12}$$

where SP_t is the target value, or setpoint, and y_t is the system output, both at time t. The square of the error can also be used. Some of the constructed environments add a small cost on changing the input to avoid large oscillations, which can be undesirable in many physical systems. A reward function combining the above objectives might look like this:

$$r(s_t, a_t) = -e(s_{t+1})^2 - w_u u(t) - w_{\Delta u}|\Delta u(t)|, \tag{3.13}$$

where $w_u, w_{\Delta t} \in [0, 1]$ are weight values, and $\Delta u_t = u_t - u_{t-1}$. Note that hard limits can also be imposed in the simulation of the system.

The example environments are fairly simple, so the reward functions can also be rather simple and general. In more complex systems or environments, a well-designed reward function is very important for convergence of learning, and for finding a good policy.

3.3.4 Simulation of system dynamics

The simulation of the system dynamics is implemented in one of two ways, depending on the environment. The simplest way is to set some fixed time constant, and calculate the state updates directly at each time step, using the differential equation(s) and the time constant.

The second method also sets a fixed time constant, which is the time each iteration of the `step` function takes, i.e. when the agent can update the system input u. However, during each iteration, an explicit Runge-Kutta method is used to solve the ODEs over the step duration. Specifically, the `integrate.RK45()` class from SciPy's integration and ODEs module [13] is used, which implements an explicit Runge-Kutta method of order 5(4). According to the documentation, the error is controlled assuming accuracy of the fourth-order method accuracy, but steps are taken using the fifth-order accurate formula, and local extrapolation is done, while a quartic interpolation polynomial is used for the dense output.

The Runge-Kutta method is fairly computationally expensive compared to the first method, but ensures a stable simulation of the system dynamics.

3.3.5 OpenAI's Gym interface

The environments are implemented using the OpenAI Gym [5] interface. In Gym, the core interface is implemented as a class called `Env`, which encapsulates an environment, which can be either partially or fully observable, hiding the specifics of the environment dynamics. Different environments are implemented by inheriting the `Env` class, and overriding its main methods and attributes. The main API methods are

- `reset()`: Resets the state of the environment, and returns the initial percept. This is used for initialization and reinitialization of the environment.
- `step(action)`: Takes an action as input, and runs one time step of the environment dynamics. Returns the percept of the next time step.

- `seed(seed)`: Sets a seed for the random number generator(s) used in the environment. This is used when the environment has some stochastic element which requires randomness (e.g. disturbance). The input parameter `seed` defaults to `None`.
- `render()`: Used for rendering the environment, useful for illustrating the progress and/or performance of an agent when simulating real world control problems (e.g. cartpole with a pendulum). This method is not implemented in the environments in this thesis, as data is logged and can be plotted afterwards.
- `close()`: Can be used if manual cleanup is required. Not implemented in the environments in this thesis.

As an example, a call to `step()` could look like this:

```
observation, reward, done, info = env.step(action)
```

where `env` is an instance of a specific environment class, and `action` is the action to be performed in the current time step.

The percept returned by `reset()` and `step()` contains the following variables:

- `observation`: An observation object, which contains the new state of the environment, which is the agent's current observation.
- `reward`: Reward value, which is the amount of reward given to the agent after performing the previous action
- `done`: A boolean value indicating whether the current episode of the environment is completed or not. When this value is `true`, further calls to the `step()` function will return undefined results.
- `info`: Contains addition diagnostic information, which is mostly used for debugging, but sometimes for learning as well.

The `Env` class also has three main attributes:

- `action_space`: The action space, which defines the space of valid actions the agent can choose from.
- `observation_space`: The observation space, or state space, which defines the space of valid states the environment can be in.

- `reward_range`: A reward range tuple which sets a boundary on minimum and maximum possible reward. The default value is $(-\infty, \infty)$, and it is only set if a more narrow range is required.

The `action_space` and `observation_space` attributes are objects of type `Space`, which is implemented by the Gym framework specifically for action and observation spaces. The most common spaces are `Discrete` and `Box`. The `Discrete` space consists of a finite amount of non-negative numbers, while the `Box` space takes bounds represented as a matrix or multiple arrays as input, and creates an n-dimensional box to represent the space.

In short, creating an environment consists of constructing an environment class, defining the necessary constants and variables as attributes, and implementing the methods described above. This includes creating a simulation of the system dynamics, and designing a reward function, which are the two main parts of the environment. To allow the algorithm implementations to remain completely general, and ensure decoupling, all the simulation and system details are defined and implemented inside the environment.

The environment can be integrated into the Gym repository by registering the environment and its entry point (the details of this process is included with the code delivery), and used as any other Gym environment. In Gym, the naming convention for environments are `EnvName-vN`, where `N` is the version number.

3.3.6 Environment classes

The different environments are implemented as classes, and follow the general Gym interface discussed above. Different ways of controlling dynamic systems were explored:

- Directly controlling the system. The states of the system are used to create the input for the DRL algorithm, which directly gives the system input as its output.
- PID tuning. Both the system state and the parameters of the PID controller are used to create the input for the DRL algorithm, which in turn manipulates the PID parameters through its own output.
- Output adjustment. The DRL algorithm works in parallel with a PID controller, producing output adjustments for the controller output. The outputs

Type	Name	Class	File name
Basic	FirstOrder-v0 SecondOrder-v0	FirstOrderEnv SecondOrderEnv	first_order.py second_order.py
Scenario	SetpointChange-v0 InputDisturbance-v0 OutputDisturbance-v0	SetpointChangeEnv InputDisturbanceEnv OutputDisturbanceEnv	setpoint_change.py input_disturbance.py output_disturbance.py
PID tuning	PidTuning-v0 OnlineTuning-v0	PidTuningEnv OnlineTuningEnv	pid_tuning.py online_tuning.py
Output adjustment	OutputAdjustment-v0	OutputAdjustmentEnv	output_adjustment.py

Table 3.1: Overview of the different environments.

of the DRL algorithm and the PID controller are combined and used as input in the system. Both the system state and the parameters of the PID controller are used to create the input for the DRL algorithm.

There are two basic environment classes, which implement the dynamics of first-order and a second-order systems described in section 3.1. Then, some environment classes implement different simulation scenarios on top of one of these system dynamics classes, using class inheritance. The scenario environments are themselves instantiated and used for simulation by other environments.

Note that in the environment implementations, the variable `state` refers to the state of the environment, which is made available through the interface as an observation. The state of the system is denoted by `x` (numbered if there are multiple state values).

As mentioned earlier, the Gym interface requires defined action and observation spaces. Many DRL algorithm implementations normalize their actions to stay within the range $[-1, 1]$, therefore this limit is used for inputs, which are then scaled up if needed. Because it is required, limits are also imposed on the state, but are mostly chosen such that they should not be reachable with the limit on u, or at least so that the optimum solution lies within the range.

Basic environments

Both of these environments mainly implement system dynamics, but also include a simulation of a simple scenario, where the state is initialized to 0, some constant `setpoint` value is used, and the system is simulated for a constant number of

steps, dictated by the variables `duration` and `step_size`.

The first environment, `FirstOrder-v0`, was implemented to simulate the first order system described in section 3.1. It is a simple environment, using the system state x as observation, and the action is the system input u. The simulation of the system dynamics is done by solving the differential equation at each iteration, using a constant `time_step` variable as Δt.

The second environment, `SecondOrder-v0`, simulates the dynamics of the second order system described in 3.1. It is similar to `FirstOrder-v0`, but slightly more advanced, mainly due to its dynamics being more complex. The reward is calculated in the same way as in equation 3.11. The observation is the state vector $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, and the action is the system input u. The state and input constraints are:

$$x_1 \in [-50, 50], x_2 \in [-20, 20], \tag{3.14}$$

$$u \in [-10, 10] \tag{3.15}$$

The `SecondOrderEnv` class is structured a bit differently from `FirstOrderEnv`, mainly because it is used as a superclass for the environments in subsection 3.3.6 below. It has three important additions apart from the standard API methods:

- `system_init()`: Initializes the system constants and other attributes used in the simulation, except for the ones used in `action_space` and `observation_space`.
- `model(t, x)`: Contains the ordinary differential equations which describe the dynamics of the system. This function is not used explicitly, but passed to the ODE solver in `simulate()`.
- `simulate(t_start, t_stop, state)`: Simulates the system from time `t_start` to `t_stop`, using `state` as the initial state. The simulation is done by solving the ODEs described in the `model()` class method using the Runge-Kutta method implemented by the `scipy.integrate.RK45` class described in section 3.3.4.

These methods are inherited by the subclasses, and used in the simulation.

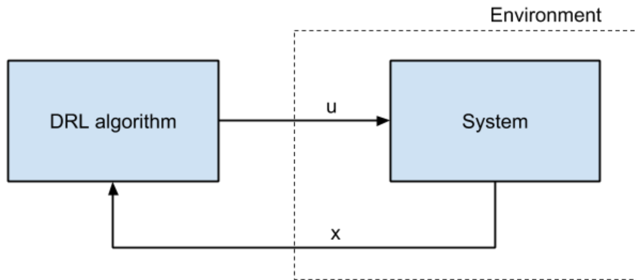


Figure 3.4: Illustration of the direct control environments. Reward is left out.

Scenario environments

These environments extend the basic environments, and add some additional functionality to simulate different control scenarios, i.e. they are subclasses of the `SecondOrderEnv` superclass. They override the initialization function and the main API methods `step` and `reset`, but inherit the other methods.

In these environments, the agent controls the system directly. Figure 3.4 shows this architecture. In this regard, the environments are similar to the basic environments: the action is the system input u. The observation, however, is different, it now includes the error value of the current step

$$\texttt{state} = \begin{bmatrix} x_1 \\ x_2 \\ e \end{bmatrix} = \begin{bmatrix} x_1 \\ \dot{x}_1 \\ e \end{bmatrix} \tag{3.16}$$

The reason for the addition to the `state` variable is that the scenario environments introduce randomness into both the timing and magnitude of certain events affecting the error value (e.g. change in setpoint or disturbance), which makes them unpredictable. Adding error to the observation allows the agent to learn to adapt to the different events.

The `SetpointChange-v0` environment simulates a scenario where the setpoint changes from 0 to some value `setpoint_change_value`, at some time step `setpoint_change_time`. Both of these variables are initialized to a random value inside the environment `reset` method.

In the `InputDisturbance-v0` environment, the state is initialized around 0, and

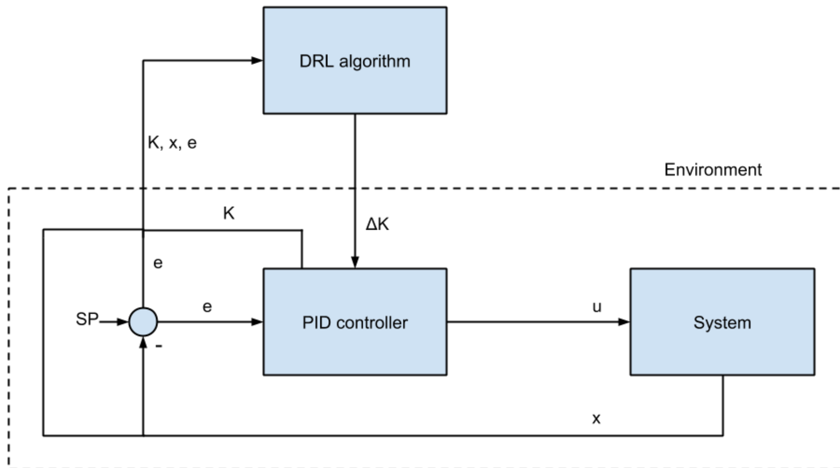


Figure 3.5: Illustration of the PID tuning environments. Reward is left out.

the `setpoint` is 0 during the simulation. At some random time step `disturbance_time`, a constant step-like disturbance of value `disturbance_value` is introduced to the system input, and remains active for the duration of the simulation. As in `SetpointChange-v0`, these variables are initialized to random values.

In `OutputDisturbance-v0`, a disturbance with a value of `disturbance_value` is introduced at time `disturbance_time` as in `InputDisturbance-v0`, but the disturbance is added to the system state output instead of the input.

The disturbances can easily be modified to have a value given by e.g. some time dependent function, or to include noise.

PID tuning

Sedighizadeh and Rezazadeh [29] and Chen et al. [6] propose architectures where RL agents do adaptive PID tuning. The PID tuning environments explore whether the DRL agent is capable of tuning the parameters of a PID controller, both adaptively, and as a one-step process at the start of each simulation. There are two different environments, `PidTuner-v0` and `OnlineTuner-v0`. In both environments, the PID controller is connected to the second order system, while the DRL agent adjusts the parameters of the PID controller in some way (see figure 3.5).

The scenario environments above are used to simulate system dynamics, by creating an instance of the scenario environment class as an attribute of the PID tuning environment class,

```
import gym
self.env = gym.make('EnvName-vN')
```

The PID controller itself is also imported in a similar way, using the implementation discussed in section 3.2,

```
import pid_controller
self.pid = pid_controller.PID()
```

Note that the constructor arguments of the `PID` class are left out for simplicity. See the environment code for more details.

In the `PidTuner-v0` environment, each episode only has a single time step. This means that there are only two states, the initial state and the end state, and only a single action is chosen. The action is a vector containing the PID parameters:

$$\texttt{action} = \begin{bmatrix} K_p \\ K_i \\ K_d \end{bmatrix} \tag{3.17}$$

subject to the restrictions

$$K_p, K_i, K_d \in [0, 10] \tag{3.18}$$

The state is initialized to 0. During the single time step, the scenario environment is ran until completion, using output from the PID controller as input for the `env.step` function, with parameters as set by the agent action input. When the scenario output returns `done = True`, the total accumulated error from all scenario time steps is used as the final state, and also used in the reward calculation:

$$\texttt{state} = [e_{total}] \tag{3.19}$$

$$r = -e_{total} \tag{3.20}$$

The `OnlineTuner-v0` environment implements an adaptive PID tuning scenario. It also imports a scenario environment, but only runs its `env.step` function once per its own episode time step, i.e. the tuning environment has as many time steps as the scenario environment it uses to simulate the system. In each time step, the DRL agent can change the parameters of the PID controller, its action is a vector

of changes in each parameter:

$$\texttt{action} = \begin{bmatrix} \Delta_{K_p} \\ \Delta_{K_i} \\ \Delta_{K_d} \end{bmatrix} \tag{3.21}$$

subject to the restrictions

$$\Delta_{K_p}, \Delta_{K_i}, \Delta_{K_d} \in [-5, 5] \tag{3.22}$$

In this environment, the state vector contains the PID parameters, the system state, and the current error:

$$\texttt{state} = \begin{bmatrix} K_p \\ K_i \\ K_d \\ x_1 \\ x_2 \\ e \end{bmatrix} \tag{3.23}$$

The parameters are subject to the same restrictions as in equation 3.18, and the system states as in 3.14.

Output adjustment

Anderson et al. [3] propose an architecture where an RL agent is combined with a PI controller to produce system inputs. The last environment, `OutputAdjustment-v0`, contains a simulation of a system with a connected PID controller. The outputs of the DRL agent and PID controller are combined to produce the system input u. Figure 3.6 shows an illustration of this setup. Both the action of the agent and the combined input is restricted to $[-10, 10]$. The observation contains the PID parameters and system state information as in equation 3.23. The system is simulated by an instance of a scenario environment.

3.4 DDPG algorithm

The main algorithm chosen for testing is the deep deterministic policy gradient (DDPG) [19] described in section 2.2.7. This section will discuss the Spinning Up [2] implementation of DDPG, which was chosen among the various versions

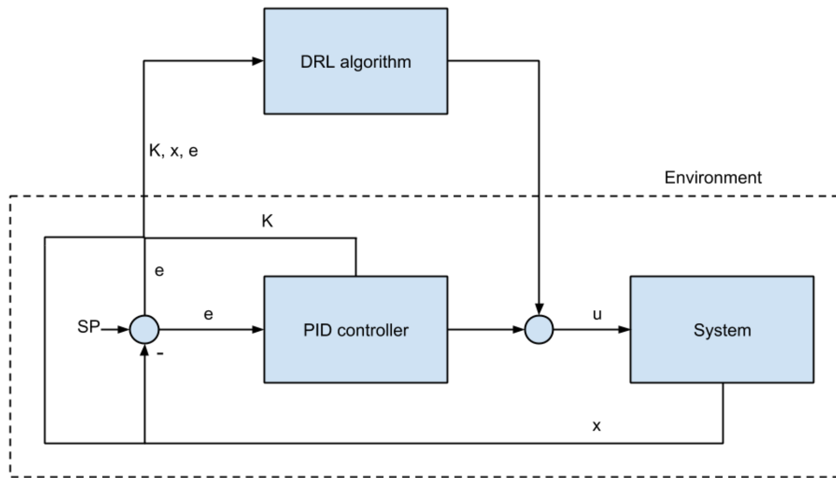


Figure 3.6: Illustration of the environment OutputAdjustment. Reward is left out.

discussed in section 2.3, mainly due to its high quality documentation, simple implementation, comprehensive functionality and solid performance.

3.4.1 Neural networks

The implementation uses the Tensorflow library [1] for most of the neural network functionality. Both the actor and the critic are implemented with multilayer perceptrons (MLPs), which are feedforward neural networks with at least one hidden layer. The network architecture is customizable, the number of hidden layers and the number of nodes in each layer can be specified in the algorithm constructor. For DDPG, the default hidden layer setup is $(400, 300)$, which indicates two hidden layers with 400 nodes in the first and 300 in the second one. With this architecture, the networks have ~ 122000 parameters each. The neural network parameters are learned using the Adam optimization method [14], implemented by the Tensorflow class `AdamOptimizer`.

Parameter	Value	Description
steps_per_epoch	5000	Number of steps per epoch
epochs	100	Number of epochs
replay_size	10^6	Replay buffer size
batch_size	100	SGD batch size
gamma	0.1	Discount factor (γ)
polyak	0.995	Polyak interpolation factor value (ρ)
pi_lr	10^{-3}	Policy learning rate
q_lr	10^{-3}	Q networks learning rate
act_noise	0.1	Action noise standard deviation
start_steps	10^4	Number of start steps

Table 3.2: Overview of default DDPG hyperparameters.

Activation functions

There are two main node activation functions used, the rectified linear function and the hyperbolic tangent function:

$$\text{relu}(x) = \max(0, x) \tag{3.24}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.25}$$

Nodes using the rectified linear are called ReLUs (rectified linear units), and nodes using the tangent are simply called tanh units. By default, the actor uses ReLUs as nodes in the hidden layers, and tanh units in the output layer. Critics are implemented with ReLUs in the hidden layers, and no specific activation function in the output layer, which means they maintain a linear activation.

3.4.2 Hyperparameters

Hyperparameters are parameters that are set before learning starts, and dictate different aspects of the learning process. An overview of the default hyperparameters in the Spinning Up DDPG implementation is shown in table 3.2.

The two first hyperparameters, `steps_per_epoch` and `epochs` decide how many total steps the algorithm is going to perform. 5000 steps per epoch and 100 epochs results in 5×10^5 time steps in total, also referred to as number of environment interactions. The replay buffer size is the maximum amount of state transitions

which can be stored at once, and `batch_size` dictates how many transitions are in each minibatch used for stochastic gradient descent when the networks are updated.

The `polyak` parameter ρ relates to the hyperparameter τ used in the DDPG target network updates (see section 2.2.7):

$$\rho = 1 - \tau, \tag{3.26}$$

so that the target parameter updates become

$$\theta_{\text{target}} \leftarrow \rho\theta_{\text{target}} + (1 - \rho)\theta. \tag{3.27}$$

To improve exploration, the agent samples completely random actions from the action space for a set number of steps during the start of training. The start_steps parameter dictates how many random steps are chosen before following routine DDPG training behaviour.

Chapter 4

Experiments and results

The DDPG algorithm implementation from section 3.4 was tested with the different environment classes from section 3.3. This chapter presents some of these experiments, and discuss the results.

4.1 Training

In the training progress comparisons, each agent is trained multiple times (5 unless otherwise specified) with different seeds, and then averaged. In training plots, the solid line represents the average, and the shaded area shows the standard deviation. Performance is measured by average returns (rewards) in test episodes which are ran during training. `TotalEnvInteracts` is the same as total time steps. The `SetpointChange-v0` environment is used for the agent comparisons. Most of the training is performed for 20 epochs, unless otherwise specified.

4.1.1 Network architectures

Different number of nodes in the hidden layers were tested, and some smaller networks were found to perform as well as the default setup of $(400, 300)$ on average. Networks with as small as $(128, 64)$ usually converge quickly, and rarely diverge or get stuck at local minima. Figure 4.1 shows a training comparison with two

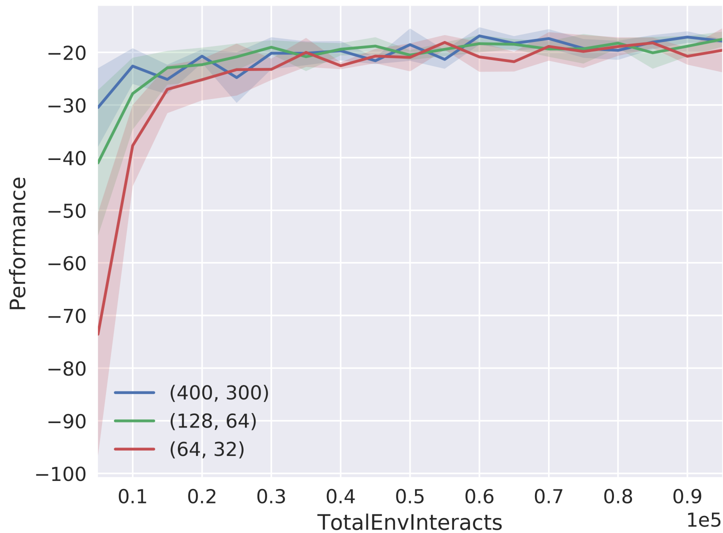


Figure 4.1: Benchmark test of two different network architectures for the `SetpointChange-v0` environment.

smaller networks with hidden layers as $(128, 64)$ and $(64, 32)$, which results in ~ 18000 and ~ 4800 network parameters, respectively, compared to the ~ 244000 of the default setup. Fewer parameters means faster training, but the difference is not drastic in terms of training time. Both of the smaller networks do learn a decent policy, but the smallest one is generally slightly below the two others. It also has a somewhat higher standard deviation. With a combination of fast learning and high performance, the architecture with $(128, 64)$ as hidden layers is used in most of the benchmark tests.

4.1.2 Hyperparameters

The default hyperparameters generally results in stable training. Figure 4.2 shows a comparison between two agents, one of which uses slightly modified hyperparameters. The updates are shown in 4.1. These changes causes the learning to become unstable, and the agent fails to converge on average. The DDPG algorithm is fairly sensitive to hyperparameter tuning. Small changes are generally unproblematic or can lead to faster learning in some cases, but since the default settings tend to work well, they are used for most of the examples. The modifications are

Parameter	Default	Modified
`polyak`	0.995	0.99
`pi_lr`	10^{-3}	10^{-2}
`q_lr`	10^{-3}	10^{-2}

Table 4.1: Default and modified hyperparameters used in figure 4.2.

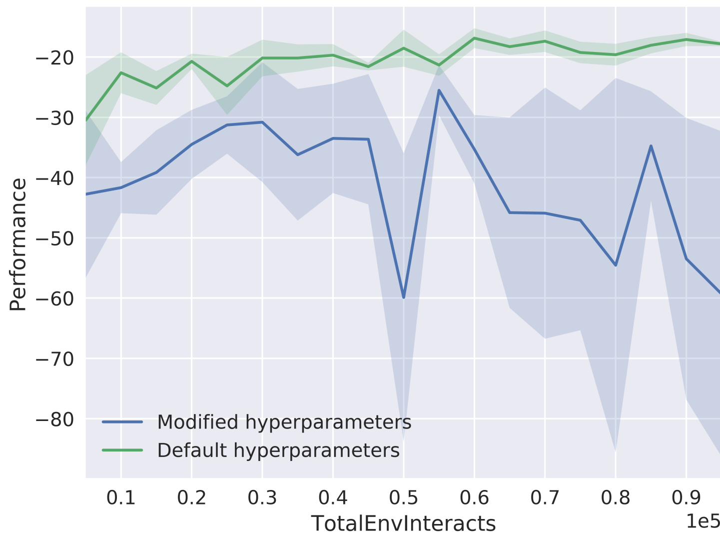


Figure 4.2: Benchmark test of two networks with different hyperparameters, using the `SetpointChange-v0` environment.

detailed in table 4.1.

4.1.3 Performance

Figure 4.3 shows the behaviour of the agent in a setpoint scenario. The performance itself is fine, comparable to that of the PID controller, but there are rather large input oscillations, even when the error is low. In an attempt to reduce input oscillations, a new reward function with a penalty for input changes was tested. The new reward function is as follows:

$$r_t = -e_t^2 - 0.01\Delta u^2 \tag{4.1}$$

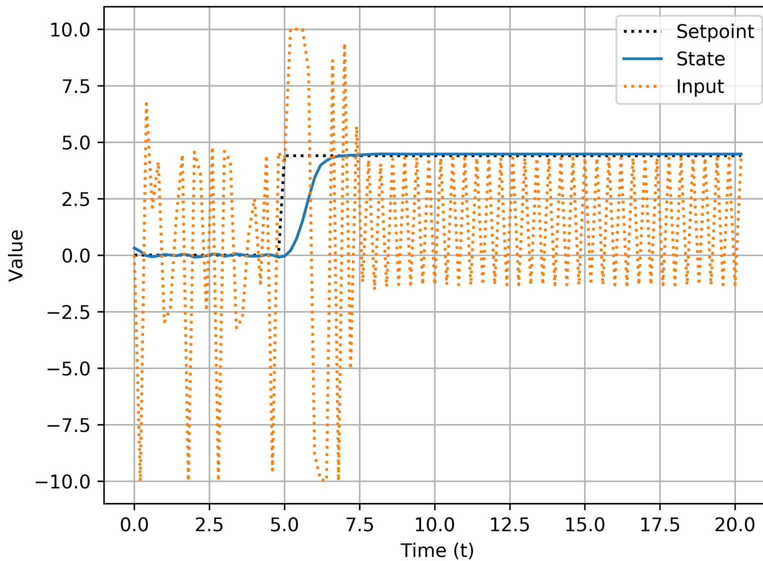


Figure 4.3: Example of input oscillations.

This is a slightly more advanced reward function, the training progress is shown in figure 4.4. The resulting behaviour is shown in figure 4.5. The rise time is still good, and there are no oscillations when the system reaches a steady state.

As a comparison, figure 4.6 shows the training progress of three different approaches to controlling the `SetpointChange-v0` environment. One-step PID tuning (implemented by `PidTuning-v0`), direct control (implemented by `SetpointChange-v0` and output adjustment (implemented by `OutputAdjustment-v0`). The result shows they all perform similarly, with the one-step PID tuning slightly outperforming the others on average.

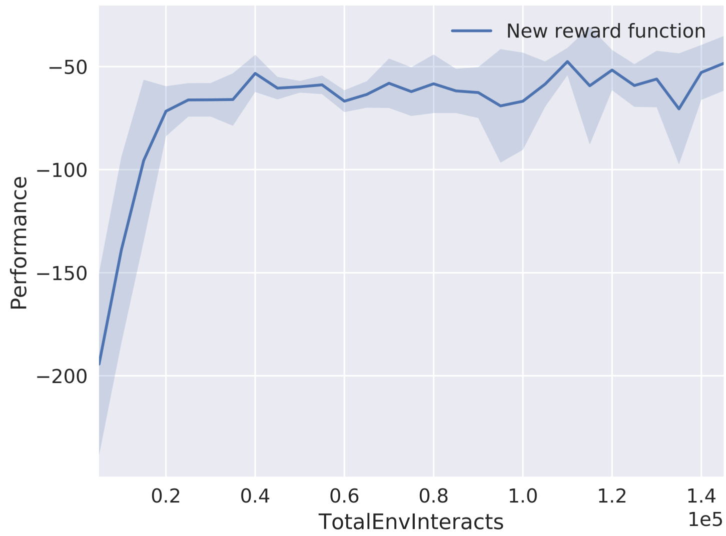


Figure 4.4: Training progress with a new reward function. 30 epochs of training in `SetpointChange-v0`.

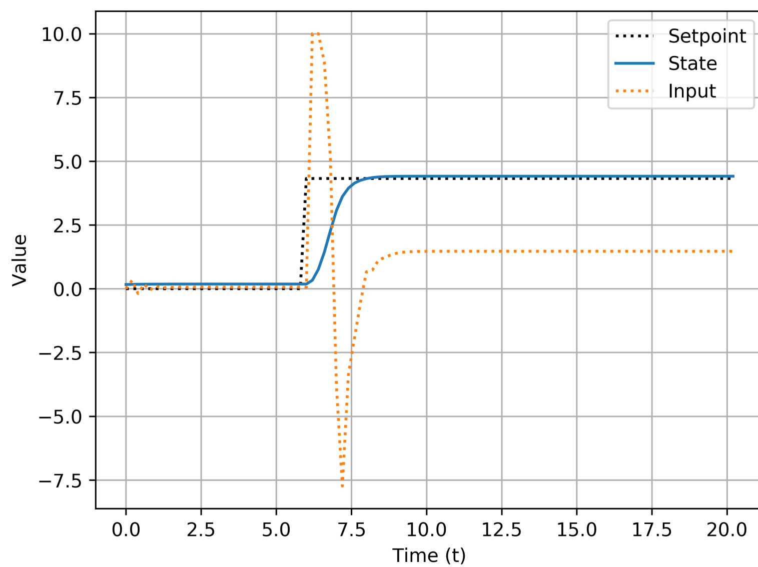


Figure 4.5: Performance of DDPG agent after training with the new reward function.

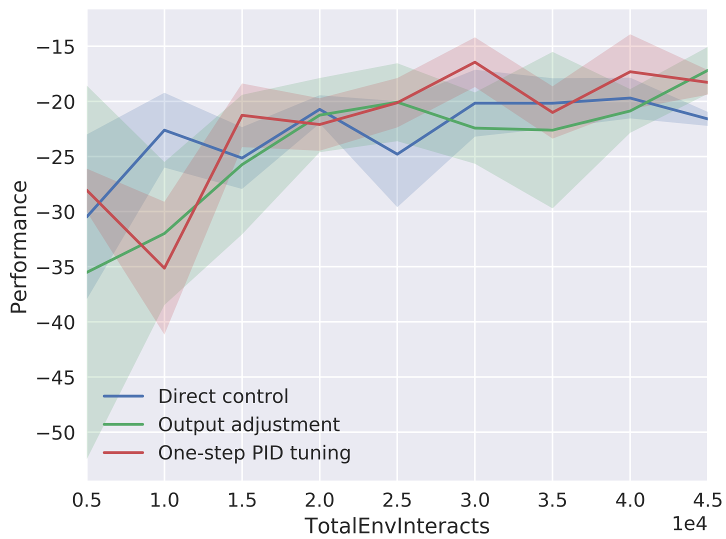


Figure 4.6: Training progress comparison for three different approaches for controlling `SetpointChange-v0`. 20 epochs of training.

Chapter 5

Conclusion

The DRL algorithm achieves fairly high performance in the demonstrated experiments, considering the general and generic implementation, short training time and little hyperparameter tuning. The DRL algorithm displays an impressive flexibility in a variety of control domains. The performance of the agent with a modified reward function demonstrates the power of using reward functions to design agent behaviour, and the level of generalization due to dynamic randomization is also notable.

However, there are substantial drawbacks to the DRL paradigm. In the control theory approach, robustness and stability are important aspects of any controller solution. When dealing with deep neural network approximators, there are few guarantees to be made in practice. In theory, given infinite simulation steps, the policy are guaranteed to converge, but this is not realizable when it comes to the real world. If something goes wrong in real systems, explanations and analysis are usually required, which can be considerably difficult or even impossible to obtain in black box systems. Even if DRL methods outperform the classic control theory methods, safety concerns and robustness and stability outweigh performance in most physical systems.

For simple problems, general implementations can yield satisfying results. In complex domains, a considerable drawback of using DRL is the vulnerability of convergence conditions, which is especially prominent in the DDPG algorithm. It can be very sensitive to the design of the reward function, and to the hyperpa-

rameters. Tuning a PID controller can seem easy when compared to the amount of different variables affecting the performance of a DRL agent.

Sample inefficiency is one of the biggest problems in DRL. The best performing algorithms usually require millions of environment interactions to find good solutions for complex problems, and few methods exist for when the sample size is small. This is also a problem for deep learning in general.

Purely data-driven approaches such as DRL might not be the leading solution for real-world applications yet, and they might never be, but there is no denying that these methods have achieved impressive feats. The attempt to combine the robustness and stability of control theory with the exciting performance of machine learning is definitely an interesting research area.

5.1 Further work

Other algorithms can be tested, such as the DDPG improvements TD3 and SAC. Functionality for hyperparameter search and even automatic tuning could be interesting. Tune [18] is an open-source alternative. Most research in DRL is focused on model-free approaches (such as DDPG), however, model-based approaches are also gaining popularity outside of very specific domains.

More control theory functionality can be added to the simulation framework, such as system analysis. The Python package `python-control` could be interesting to integrate. Test with other controllers, such as LQR or MPC.

Using DRL and dynamics randomization to transfer learned policies from simulation to the real world is an interesting field, see OpenAI's Learning Dexterity[1].

[1] `https://blog.openai.com/learning-dexterity/`

Bibliography

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

[2] Joshua Achiam. Spinning Up in Deep RL, 2018. URL `https://spinningup.openai.com/`.

[3] Charles W. Anderson, Douglas C. Hittle, Alon D. Katz, and R.Matt Kretchmar. Synthesis of reinforcement learning, neural networks and pi control applied to a simulated heating coil. *Artificial Intelligence in Engineering*, 11(4):421 – 429, 1997. ISSN 0954-1810. doi: https://doi.org/10.1016/S0954-1810(97)00004-6. URL `http://www.sciencedirect.com/science/article/pii/S0954181097000046`. Applications of Neural Networks in Process Engineering.

[4] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *CoRR*, abs/1705.08439, 2017. URL `http://arxiv.org/abs/1705.08439`.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[6] Pengzhan Chen, Zhiqiang He, Chuanxi Chen, and Jiahong Xu. Control strategy of speed servo systems based on deep reinforcement learning. *Algorithms*, 11(5), 2018. ISSN 1999-4893. doi: 10.3390/a11050065. URL http://www.mdpi.com/1999-4893/11/5/65.

[7] George Cybenko. Continuous Valued Neural Networks with Two Hidden Layers are Sufficient. Technical report, Department of Computer Science, Tufts University, 1988.

[8] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Controls, Signals, and Systems*, 2:303–314, 1989.

[9] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. https://github.com/openai/baselines, 2017.

[10] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *CoRR*, abs/1803.00101, 2018. URL http://arxiv.org/abs/1803.00101.

[11] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2455–2467. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7512-recurrent-world-models-facilitate-policy-evolution.pdf.

[12] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[13] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46056-5.

[16] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018. URL `http://arxiv.org/abs/1810.06339`.

[17] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[18] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL `http://arxiv.org/abs/1509.02971`.

[20] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[21] C. Grant Luckhardt. *Wittgenstein, Sources and Perspectives.* Cornell University Press, 1979.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[24] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with

model-free fine-tuning. *CoRR*, abs/1708.02596, 2017. URL http://arxiv.org/abs/1708.02596.

[25] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[26] Razvan Pascanu, Guido Montúfar, and Yoshua Bengio. On the number of inference regions of deep feed forward networks with piece-wise linear activations. *CoRR*, abs/1312.6098, 2013. URL http://arxiv.org/abs/1312.6098.

[27] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *CoRR*, abs/1706.01905, 2017. URL http://arxiv.org/abs/1706.01905.

[28] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, New Jersey 07458, 3rd edition, 2010.

[29] Mostafa Sedighizadeh and Alireza Rezazadeh. Adaptive pid controller based on reinforcement learning for wind turbine control. 2008.

[30] David Silver. Reinforcement learning course lectures, 2015. URL http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

[31] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014. URL http://dl.acm.org/citation.cfm?id=3044805.3044850.

[32] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2018.

[34] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press. URL http://dl.acm.org/citation.cfm?id=3009657.3009806.

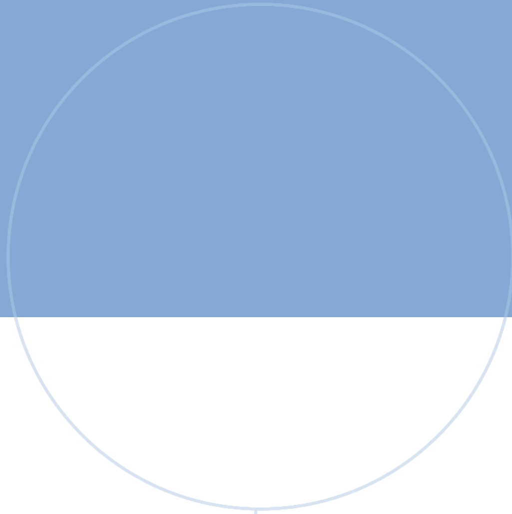
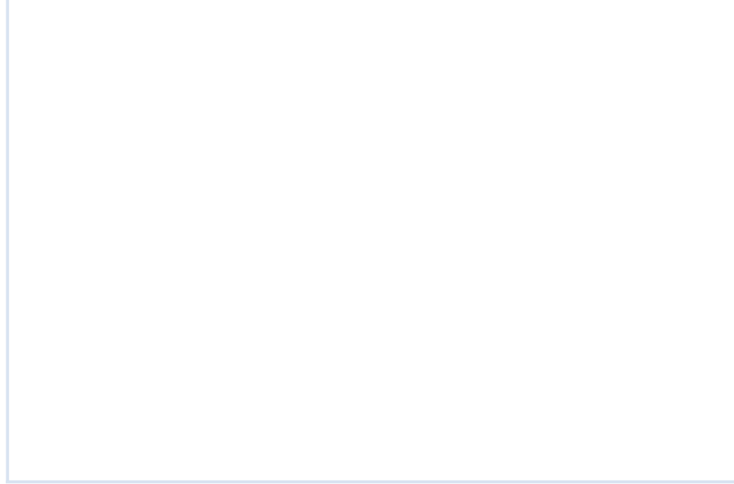
[35] Guido van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8 – Style Guide for Python Code. https://www.python.org/dev/peps/pep-0008/, 2001.

[36] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[37] Theophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. *CoRR*, abs/1707.06203, 2017. URL http://arxiv.org/abs/1707.06203.

[38] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

[39] John G. Ziegler and Natalie B Nichols. Optimum settings for automatic controllers. 1942.

NTNU
Norwegian University of
Science and Technology