Development of a Reinforcement Learning System for stable flight control of a Quadcopter

BACHELORTHESIS

Jan Larwig

Applied Computer Science

RheinMain University of Applied Science

June 2018

This thesis was created during

Summer Semester 2018

Advisor:

Prof. Dr. Ulrich Schwanecke / Prof. Dr. Adrian Ulges

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

RheinMain University of Applied Science, June 28, 2018

Jan Larwig

Contents

Abstract

The ability to reliably and autonomously control a Quadcopter or drones, in general, is an important endeavor, to ensure the success of new applications of drones. These applications include the monitoring of climate changes in the environment, surveillance of highly secure areas or the exploration of difficult terrain. All these applications depend on the ongoing development to improve flight systems for drones.
Inspired by the vast success of deep neural networks and Reinforcement Learning systems in recent years, the idea to build a self-learning system for controlling a Quadcopter was sparked.
For this reason, a Reinforcement Learning system for learning to fly a Quadcopter without human interaction was implemented in this thesis. The implementation is split into the exploration and exploitation strategy. Meaning, that trajectories are generated autonomously without human interaction and two neural networks, a policy and a value network respectively, are trained on this data to predict the thrust of the Quadcopter rotors needed to fly towards a target position.

Chapter 1

Introduction

The development of drones was significantly propelled over the last decade. Especially consumers got more and more in touch due to large commercial companies. But many other industries have shown their interest in recent years, too. For example logistic and delivery companies are interested in using drones as a fast delivery method for small consumer products. Other possible use cases are the exploration of difficult terrain, the observation of agricultural areas, collection of meteorological data, the monitoring of highly secure zones and many more.[1]
The ongoing development of drones is crucial for all these applications. Especially the ability to autonomously fly in a stable and robust manner is important to be even more reliable in the future. Drones need to be capable of consistently flying stable, even under harsh conditions with lots of turbulences caused by bad weather. This would make it easier and more cost efficient to rely on them, as human interaction could be reduced to a minimum. Given that, it is no wonder that many universities and companies actively and consistently develop and improve the abilities of drones. There are a few universities that have to be mentioned as pioneers in this field: The ETH Zurich, TU Munich, TU Delft, Boston University and the University of California.

1.1 Current state of stable flight control

While writing this thesis the industry standard for flight control of drones remains in the hands of manually tuned PID controllers and handwritten mathematical models. PID is an acronym for the three modes such a controller consists of: Proportional, Integral and Derivative. PID controllers are used for many automatically controlled industry applications. Examples are the regulation of temperature, pressure levels, automatic braking and many more. The following summary is dedicated to explain the concept of a PID controller and to understand its benefits and disadvantages. In Figure 1.1 the three modes of a PID controller are illustrated.[2]

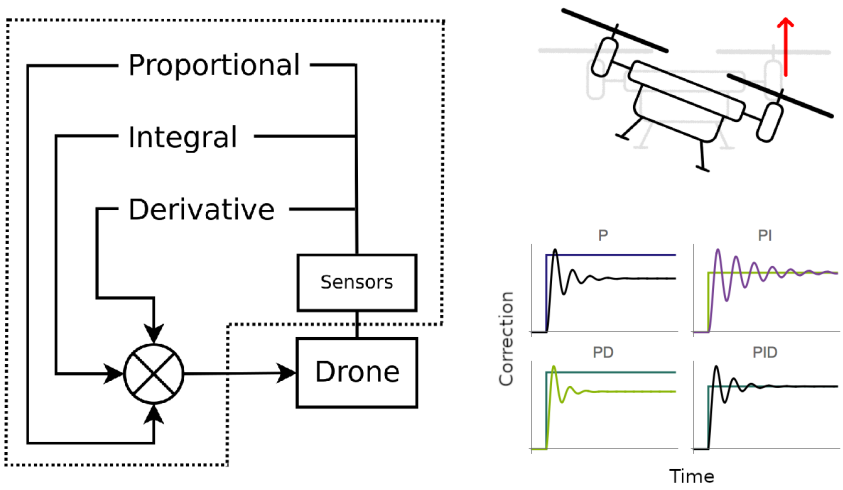


Figure 1.1: Simplified flight controller on the left. Tilted drone and PID example correction on the right.

The most basic way to explain the three different modes and how they work together is based on the timeframes they operate in. The proportional mode reacts to present events, the integral mode looks at the past and the derivative one tries to predict future events. In case of a tilted drone, as seen in Figure 1.1, the proportional mode compares the present state / orientation of the drone and gives a lot of initial thrust and rapidly overshoots its goal of a stable parallel orientation to the horizon. The proportional mode corrects again and again, which leads to oscillation around the target. It often gets more unstable closer to the target and might never truly converge, as can be seen in the top left P-graph in Figure 1.1. The integral mode helps the proportional mode by looking at the accumulated error of the past proportional corrections. Its purpose is to precisely match the target. Due to the accumulation of past errors the oscillation pattern might persist or even increase, thus creating a more unstable correction, as seen in the top right PI-graph. The derivative mode tries to anticipate the future orientation of the drone, by looking at the present correction to be taken by the proportional mode. Its counter correction will then help to decrease the oscillation around the target, contributing to a faster convergence. This can be seen in the bottom left PD-graph. At last all three modes are combined to a fully fledged PID controller and applied as one compounded correction to the drone rotors.[3]

1.2 Objective of this thesis

The problem of the traditional way of controlling drones via PID controllers are the endless scenarios and events a developer has to anticipate in his model. In addition, the adaptation of specifically tuned PID controllers is quite labour intensive because the controller values have to be fitted to every single drone model. On these grounds, the objective of this thesis is to implement and test a system which is capable of learning to control a Quadcopter fly by itself. The motivation behind this system is that a sophisticated self learning system could be capable of adapting to other drone models and handle unforeseen events better

than a strict and finite PID controller implementation. The self learning system of this thesis will be based on a Reinforcement Learning setup, developed in a previously published study.[4] This setup will collect flight data in form of trajectories and train a neural network to control the four rotors of the AscTec Hummingbird drone, which is used for this thesis.[5] The neural network and the training procedure are implemented in Python using Tensorflow. As a simulation environment the Gazebo simulator is used. Chapter 3 describes the implementation and how the mentioned tools work together.
To get a rough understanding of how the components of this setup work, a brief explanation of Tensorflow, the Robot Operating System, Gazebo, neural networks and Reinforcement Learning is provided in the next chapter. Reinforcement Learning differs from other machine learning techniques. It tries to mimic psychological or biological behavior with a feedback loop of actions, instead of relying on human feedback to verify the correctness of the output. Which would normally be the case in traditional supervised learning systems. The main focus of Reinforcement Learning is on performance, which is represented by a balancing act between exploration and exploitation. In other words, a trade-off between the exploration of the uncharted territory of the learning tasks environment and the current knowledge has to be made. Which means that randomness needs to be part of the learning process to ensure that the system eventually finds actions that are superior to the ones it would have done by purely relying on its current knowledge.[6]
To recapitulate, the problem with PID controllers in drones is the limited amount of scenarios a developer can anticipate in his model. For this reason, the idea is to create a Reinforcement Learning system that is able to better react to unpredictable disturbances.

1.3 Research questions

Following the objective in Section 1.2, the list below summarises the main research questions of this thesis.

- How to built a Reinforcement Learning system with Tensorflow and Gazebo?
  - How to implement a reliable exploration strategy?
  - How to implement a custom optimization strategy?
- How does the feedback loop for the neural network optimization work in detail?
- How can the developed system be used on a real machine in the future?

Chapter 2

Background

The purpose of this chapter is to make the reader familiar with the background knowledge required to fully understand the objective and realisation of this thesis. To understand how the implemented setup works, a brief introduction of the frameworks and software used in this thesis is given in Section 2.1. Furthermore an introduction of neural networks is given in Section 2.2 and in Section 2.3 Reinforcement Learning and its benefits compared to more traditional learning techniques are explained. To better understand the origin and inspiration of this thesis, Section 2.4 summarizes a research paper of ETH Zurich. The paper lays the basis for this thesis.

2.1 Frameworks & Software

This section explains the main frameworks and software that this thesis is build upon. These include: Gazebo, the Robot Operating System (ROS) and Tensorflow.

2.1.1 Gazebo & RotorS

Gazebo is a free robotics simulator for creating physically correct applications for real robots.[7] Such a simulator enables the developer to save costs and time. Instead of depending on an actual machine. The developer is able to test algorithms, perform regression testing or train AI systems using realistic scenarios in the simulator beforehand.
The RotorS simulator extension was built for Gazebo to enhance the very specific development of Micro Air Vehicle (MAV) systems.[8] It was built to provide a quick starting point for performing research on MAVs via ROS and Gazebo. The developer is provided with several pre made multirotor models based on the ones used in industry, like the AscTec Firefly and the AscTec Hummingbird, the later is used for this work. Furthermore, the framework provides controllers and services to communicate on a realistic basis with these drones. After installing the simulator, it can be simply started and controlled by the following example commands:

```
$ roslaunch rotors_gazebo mav.launch mav_name:=hummingbird world_name:=basic
$ rostopic pub /hummingbird/command/motor_speed
    mav_msgs/Actuators '{angular_velocities: [460, 460, 460, 460]}'
```

The first command starts Gazebo with the RotorS extension and sets a multirotor in a specified world. The parameter **`mav_name:=hummingbird`** specifies the multirotor model to be the AscTec Hummingbird and the parameter **`world_name:=basic`** specifies a basic empty world without any additional obstacles or models. The second command uses a

ROS publisher to send four angular velocities to the Quadcopter, one for each rotor. More about controlling the simulator is explained in the next section.

2.1.2 Robot Operating System

The Robot Operating System is a robotics middleware providing services to communicate with robots.[9] This middleware provides an abstraction level which makes it possible to build a system that works the same on an simulated robot as on a real machine. To communicate with ROS, a Python and C++ library exists and ROS is integrated into Gazebo, thus making it the perfect choice for the intention of this work. As shown in the previous section, Gazebo / RotorS can be controlled via ROS commands. For further understanding of controlling the simulator via Python, it is required to understand the architecture of ROS.

As stated before, ROS is not really an operation system but a robotics middleware, which was designed with extensibility in mind. The easiest way to understand ROS is to look at the publisher-subscriber pattern, which is often used in the software architecture of distributed systems. Gazebo provides ROS subscriber and publisher nodes, for which a developer can built a publisher or subscriber respectively. In this way, the developer is able to receive or send data in a stream like manner. These nodes belong to the ROS-Topics. An example subscriber could constantly receive the model state, which would consist of the Quadcopters position, orientation and the current velocity. In addition, ROS does not only provide the stream like ROS-Topics but also the so called ROS-Services for on-demand connections. These can be used, if the developer does not need a constant stream of data or just wants to terminate a task quickly. Another important issue is the fact that ROS-Topics only work, if the simulation is still running and not paused. If the developer, for example, wants to pause the simulation and then change the model position, it is required to use the ROS-Service architecture. This thesis uses the `rospy` Python library to communicate with the Gazebo simulator via ROS. The following code provides the reader with basic examples to understand the usage of the library.

```python
import rospy
from mav_msgs.msg import Actuators
from gazebo_msgs.srv import (GetLinkState, SetLinkState)
from gazebo_msgs.msg import LinkState

# Initialize a new ROS node with a arbitrary but unique name
rospy.init_node('gazebo')

# Create a publisher for the hummingbird rotors
pub = rospy.Publisher('/hummingbird/command/motor_speed', \
                      Actuators, queue_size=10)

# Publish the velocities for every of the four rotors
pub.publish(angular_velocities=[480, 480, 480, 480])

# Create a service for on-demand data fetching
get_link_state = rospy.ServiceProxy('/gazebo/get_link_state', GetLinkState)

# Fetch position and orientation of the hummingbird model base
base = get_link_state(link_name='hummingbird/base_link', \
                      reference_frame='world').link_state

# base is now a instance of class LinkState
# base.pose contains information about the position and orientation
# base.twist contains information about the linear and angular velocity

# Create a second service for on-demand data sending
set_link_state = rospy.ServiceProxy('/gazebo/set_link_state', SetLinkState)

# Change the current height of the drone and send the new state
base.pose.position.z += 10
set_link_state(base)
```

Code Listing 1: Example usage of the ROS Python library rospy. To demonstrate how to send and receive basic information to and from Gazebo

2.1.3 Tensorflow

Tensorflow is an open source software library for high performance numerical computation and is designed as a dataflow programming framework. [10], [11] This means that it is modelled in the form of directed graphs consisting of operations and data flow. Tensorflow is being developed by Google and is mainly used for a wide range of machine learning tasks. It provides a wide variety of machine learning techniques needed for neural networks. For performance reasons, its main core is written in C++ and the main API is accessible through Python. Thus making it the perfect choice for this thesis to allow an easy exchange of data between ROS via rospy and Tensorflow.

To get a better understanding of this framework, the following paragraphs explain the basic concepts and how simple operations are built and executed. As Tensorflow is designed as a dataflow framework, it does not execute its operations directly, as it is usually the case in Python. Instead every operation is added to a graph. Such a node can later be executed in a running session over and over again. Yet, the following example may seem to complicated

for a simple addition of two numbers but it explains the basic idea of the graph execution concept.

```python
# First of all import tensorflow
import tensorflow as tf

# Create two constant values
a = tf.constant(2)
b = tf.constant(3)

# print(a) wont show the value of 2 because tf.constant creates
# a so called Tensor object, as seen bellow:
# Tensor("Const:0", shape=(), dtype=float32)

# Create a third graph node for the addition operation
c = tf.add(a, b)

# Create a session and execute a specific node
while tf.Session() as sess:
    # The following code will print out 2, because the session will only
    # execute the graph up to the constant node "a"
    print(sess.run(a))

    # The following will print out 5, because the session will execute the
    # graph from the beginning up to the point when "c" was defined
    print(sess.run(c))
```

Code Listing 2: Tensorflow graph execution example to demonstrate the basic idea behind the indirect dataflow technique

The previous code generates a graph similar to the one in Figure 2.1 and is executed top down, when calling **`sess.run`** on the operation node **`c`**.

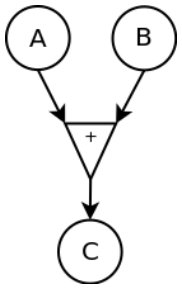


Figure 2.1: A simple flow graph for adding two values

With this architecture, the developer is able to make use of dozens of pre-built numerical operations which are optimized for high performance. Due to the graph structure, a complex calculation can be run several times by calling the specific node and feeding it with the required data, without constantly declaring any intermediate variables. This is done with so called Tensorflow Placeholders. Code Listing 3 will give the reader a simple example for understanding their usage.

```
# Define a constant value and a placeholder
a = tf.placeholder('float')
b = tf.constant(3)
# Define a operation node
c = tf.multiply(a, b)

while tf.Session() as sess:
    # Feed in the value for placeholder a and execute the graph up to node c
    # The result would be 6
    print(sess.run(c, feed_dict={a: 2}))
    # It even works with matrices
    print(sess.run(c, feed_dict={a: [[3,2,6], [1,7,4]]}))
    #array([[ 3,  6,  9],
    #       [12, 15, 18]], dtype=int32)
```

Code Listing 3: Tensorflow graph demonstration for how to feed data into placeholders

2.2 Neural Networks

As described in Section 1.2 the objective of this thesis is to build a self learning system for flight control of a Quadcopter. The most versatile technique for such a complex learning task are so called neural networks. Neural networks are built using connected layers of nodes or artificial neurons, which are vaguely inspired by the structures of neurons in living brains. The original idea of the neural network structure was intended to simulate the human brain. Today, the technique is used instead for solving a large variety of machine learning tasks. Over the past decade, it was discovered that neural networks perform extraordinary well in the following areas of machine learning: Object recognition in images, voice / language recognition, translation with semantic knowledge, social network analysis, medical diagnosis and playing computer games on a human level or even better.[12]–[14]

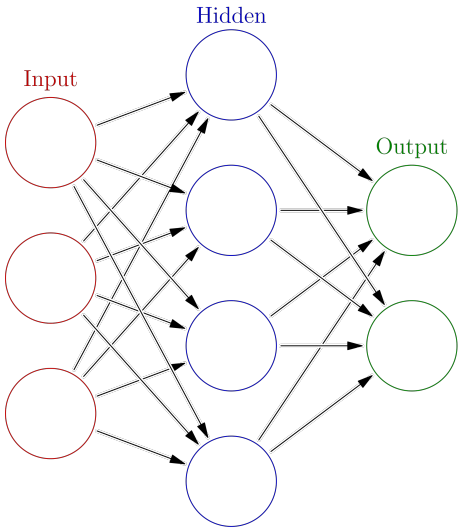
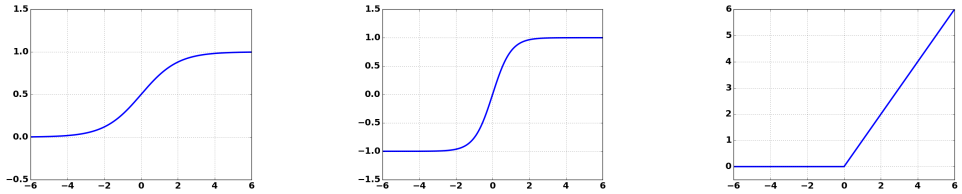


Figure 2.2: A fully connected network with one hidden layer, three input features and two outputs.[12]

As mentioned in the previous paragraph, neural networks consist of layers of nodes, so

called neurons. Figure 2.2 helps to get a better understanding of the structure. The first red layer is the input layer, the green layer is the output, and all layers in between are called hidden layers. A neural network can consist of as many hidden layers and neurons as needed for its task. The connections between the neurons are called weights. Moreover in most neural networks every neuron has an associated bias. The output y of an neuron is calculated by a weighted sum. That means all its inputs x are multiplied by their corresponding weights W and then are summed up. In addition, a bias b can be added. This procedure is described by $y = \sum(Wx) + b$. This formula gives a result ranging from -inf to +inf. However, as neural networks are vaguely based on living brains, it is desired to build a threshold to act as an activation or firing behavior for the neurons. For this purpose, activation functions are used to normalise the output y. There are simple linear activation functions like the rectified linear unit illustrated in Figure 2.3 and more complex non-linear, logistic and hyperbolic functions like the Sigmoid function or Tanh. The later is applied for the neural networks in this work.

$$f(x) = \frac{1}{1 + e^{-x}} \qquad f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \qquad f(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases}$$



(a) Sigmoid **(b)** Tanh **(c)** ReLU

Figure 2.3: Three functions which are commonly used as activation functions

With this structure, neural networks are capable of learning a huge variety of tasks. The learning process itself is done by applying a technique called backpropagation. Meaning in case of a specific neural network, e.g. for predicting the rotor thrusts of a Quadcopter from a state, the network would be fed with flights recorded by well trained human operators. The features of the state could be the position, orientation and velocities of the Quadcopter at a given point in time. By the definition of the backpropagation technique an error as the difference between the desired output and the prediction of the network will be calculated. The error will then be distributed back through the network layers. This technique is used by the gradient descent algorithm to calculate a gradient of all parameters (weights and biases) w.r.t. a loss function and apply the gradients as an update to the neurons parameters. By doing so, all parameters of the network will be slightly optimized for every pair of desired output and predicted output. This has to be done with as many possible different inputs to ensure that more reliable predictions are obtained for future inputs.

2.3 Reinforcement Learning

A Quadcopter can experience a lot of unforeseen turbulence while flying. Due to this, it is not reasonable to use supervised learning to train a Quadcopter to fly, as described in Section 2.2. This would meant that humans would have to fly and record trajectories for hundred of thousands of hours, which is obviously a costly and inefficient solution. As briefly explained in Section 1.2, Reinforcement Learning is a technique to build self learning systems. This technique is largely inspired by psychological behavior and how learning works in humans.[6], [15] A typical reinforcement learning system is build upon the components illustrated in Figure 2.4.

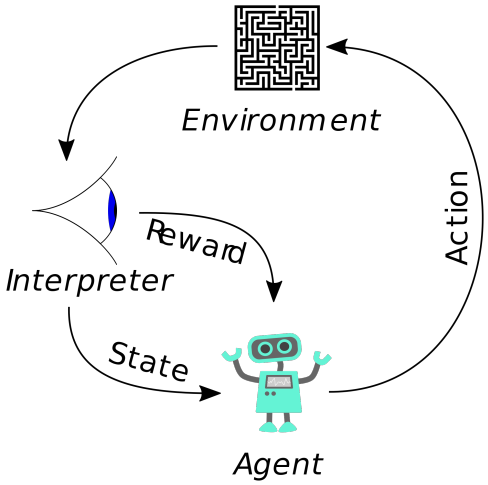


Figure 2.4: A Reinforcement Learning agent with a reward function w.r.t. its own actions in a specific environment. The core of the learning happens in the interpretation of the agents actions in relation to the reward generated in the environment.[6]

In a reinforcement system, an agent takes actions in a specified environment and these actions will then be interpreted by a reward / value function. Out of this, a representation in form of a state is generated, which is fed back into the agent. This can be compared to how humans learn. To give a simple entry example to Reinforcement Learning, the process of a child learning to walk is described in the following paragraph.
A child tries to walk out of instinct and is primarily driven by the observation of its surroundings. In this specific scenario, the child is considered to be the agent and the actions are its attempts to walk. The feedback loop would consist of the reactions the child gets from its parents, the distant it has covered and the unpleasant feeling of falling. Thus the child will try over and over again. Developers try to replicate that behavior with Reinforcement Learning in a more structured way by formulating reward functions and exploration strategies for specific environments. This is especially done for certain complex learning tasks, for which more traditional supervised learning is not sufficient.
The first working Reinforcement Learning system with neural networks was demonstrated by DeepMind in 2013.[16], [17] Since then the development of so called Deep Reinforcement Learning systems significantly accelerated. DeepMind developed a system which was

capable of learning to play all kinds of Atari games, just from analysing the pixels of every frame. It had no previous knowledge of the games or what the controls are. The system only got the pixels and current game score as input. By exploring all possible actions in a game and receiving the in-game rewards it was able to learn playing many Atari games at a human level or above.

For the purpose of getting a better understanding of how a Reinforcement Learning system works, a introductory implementation of the multi-armed-bandit problem is presented in the following paragraphs.

The multi-armed-bandit problem is a classic probability theory problem.[18] The name comes from the idea of a gambler sitting in front of slot-machines which are sometimes called one-armed-bandits. The gambler has to decide which slot-machine to use. Assuming all slot-machines have fixed payout rates, the obvious decision would be to use the slot-machine with the highest probability of winning. For this reason the gambler has to randomly try them out for a while to be able to calculate their probabilities. To maximise his reward, he then always uses the bandit with the highest probability. In this scenario, the second best or an even worse bandit could be chosen, if the probabilities of the best bandits are close to each other. To find the best bandit, an ϵ-greedy decision process can be used.[19] Meaning, for most of the time the gambler will still chose the bandit he calculated to have the highest probability but for a certain percentage he will sometimes chose another one randomly. The goal of this decision process strategy is to truly maximize the reward. This demonstrates the classical balancing act in Reinforcement Learning between exploration and exploitation of the environment.

The typical attributes that make a problem a Reinforcement Learning problem are:

1. Actions in the environment change the state of the agent and yield different rewards. E.g. playing the classic game of Snake, the action to go left may lead to open space and going right may lead to bite itself.
2. The actions do not trigger instant rewards. In other words, rewards are delayed over time. Meaning, even if going left in the above example may be the right decision, the reward in form of an apple may not occur until later.
3. The reward for actions is conditioned on the environment. In the above example going left may be the right choice to reach the apple later on. In another scenario going right, top or bottom could be the right choice.

Therefore, the multi-armed-bandit problem was chosen because of its simplistic nature, as the implementation does not need to consider point 2 und 3.[20] The only thing to be taken into consideration is which rewards are gained from the different bandits and to choose the best one over time. These learned environmental rules are called a policy.

The implementation is presented through several Code Listings in the following paragraphs. Firstly, the environmental behavior has to be defined.

```
1  # constants
2  bandits = [-0.2, 0.3, -4, 1.6]
3  num_bandits = len(bandits)
4
5  # feedback function
6  def pullBandit(threshold):
7      random = np.random.randn(1)
8      if random > threshold:
9          return 1
10     else:
11         return -1
```

Code Listing 4: The function *pullBandit()* gets the payout threshold of a bandit and compares it to a normal distributed random number.

Every bandit in the implementation has a fixed payout threshold, as can be seen in line 2 of Listing 4. The function *pullBandit()* will be used as the main environmental feedback function to generate the rewards. The next step is to implement a fairly small neural network which only consists of four weights, each corresponding to one bandit and taking the maxima of the weights as the future action. For the training, the standard Gradient Descent Optimizer of Tensorflow is used. The loss function is defined as $L = -log(\pi)A$. The advantage A is used in Reinforcement Learning to determine how good an action was in relation to a baseline. In the example of the multi-armed-bandit, zero reward can be considered to be the baseline and A can simply be described as the reward the agent receives through its chosen action. The policy π corresponds to the chosen actions weight in the network. The implementation of the network is illustrated in Listing 5.

```
1  # network weights and output
2  weights = tf.Variable(tf.ones([num_bandits]))
3  chosen_action = tf.argmax(weights,0)
4
5  #The next six lines establish the training proceedure.
6  # For the training procedure a few placeholder are required
7
8  # chosen action and the received reward
9  reward_holder = tf.placeholder(shape=[1],dtype=tf.float32)
10 action_holder = tf.placeholder(shape=[1],dtype=tf.int32)
11
12 # responsible weight or to be more precise the used bandit
13 responsible_weight = tf.slice(weights,action_holder,[1])
14
15 # loss function for the gradient
16 loss = -(tf.log(responsible_weight)*reward_holder)
17 optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
18 train_step = optimizer.minimize(loss)
```

Code Listing 5: Initialization of the network and its optimization strategy

At last, the network can be trained by exploring the environment. To put it in other words, the agent has to try the bandits a thousand times. The full training procedure is implemented in Listing 6.

```
1  # Total number of times the agent chooses an action
2  total_episodes = 1000
3  # Epsilon for greedy decision making
4  eps = 0.1
5
6  with tf.Session() as sess:
7      sess.run(tf.global_variables_initializer())
8      for _ in range(total_episodes):
9          # Choose either a random action the best one currently known
10         if np.random.rand(1) < eps:
11             action = np.random.randint(num_bandits)
12         else:
13             action = sess.run(chosen_action)
14
15         # Getting a reward from the environment
16         reward = pullBandit(bandits[action])
17
18         # Executing the optimization
19         sess.run([train_step], feed_dict={reward_holder:[reward], \
20                                           action_holder:[action]})
21
22     print('Weights', sess.run(weights))
23     print('Best Bandit', sess.run(chosen_action)+1)
```

Code Listing 6: Training the agent on 1000 episodes of choosing an action and receiving a reward

After initializing the constants *eps* and *total_episodes* in lines 1-4 the training process starts in line 8. The agent chooses a bandit with the ϵ-greedy decision process in lines 10-16. The generated reward and the corresponding action is then fed into the optimization step in line 19. This is done 1000 times. After an example run, the last two lines generated the following output:

```
('Weights', array([1.0000141 , 0.996008  , 1.6830083 , 0.97571737]))
('Best Bandit', 3)
```

To recapitulate, Reinforcement Learning was inspired by biological and psychological behavior. The idea is to not train a system on pre-labelled samples but to train a policy by randomly choosing actions in an environment and updating the system's parameters and the policy / decision making process by evaluating the rewards received from the environment. In the example implementation above, the policy parameters consist of the four weights. The final policy after exploring the environment with the ϵ-greedy strategy is to always choose the maxima out of the four. In this example, the agent should always choose bandit three.

2.4 Related research

This thesis is largely based on a previously published study by a research team of ETH Zurich.[4] A lot of ideas and methods of that paper were replicated and elaborated in detail for this thesis. For this reason, a short summary of their system will be given in this section, followed by a comparison with the implementation in the present work in Chapter 4. The training system consists of two neural networks, as seen in Figure 2.5. A policy and a value network. The policy network is dedicated to predict the thrust of the rotors for the present state of the drone. The state consists of the orientation, the angular and linear velocity as well as the relative position of the drone to its target. The value network is only used for the learning process. It is trained to evaluate the current state in relation to a given termination state of the corresponding trajectory. While training the policy network, the value network's predictions are used to evaluate a given state in relation to the target position. Its scalar output is used as a factor in the feedback loop. Both networks are structured in the same manner, except for the output layer. The policy network has four outputs, each corresponding to one rotor of the drone. The value network has only one output, which is used as a factor in every learning step of the policy network.

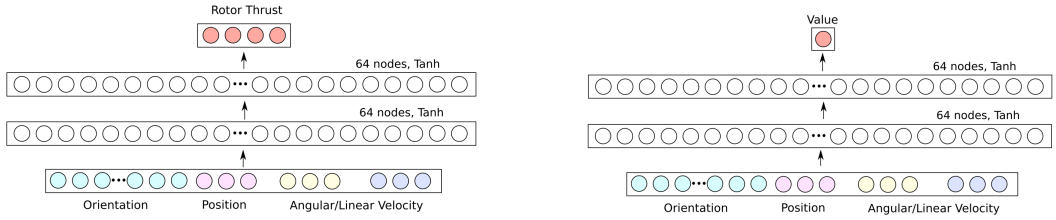


Figure 2.5: Policy network (a) and value network (b).[4]

The exploration strategy for this reinforcement system is presented in Figure 2.6. It can be separated into three steps. Firstly, the drone flies an initial trajectory predicted by its current policy network. Secondly, additive noise is randomly added at several time slots of the initial trajectory, thus creating junctions. These junctions are represented by red dots in Figure 2.6. Thirdly, the policy network predicts branch trajectories, starting from every position in time where noise was added. With this exploration strategy, a large variety of flights can be generated fully autonomously without any human interaction. This data is then used to train the value network based on all on-policy samples. After that, the policy network is trained on all junctions. That means a comparison between the flight outcome before and after the noise addition is performed and fed back into the network. For this step, the prediction of the value network is used as a reward factor to compare the quality of the trajectories before and after the noise addition. As illustrated in Figure 2.6, all predicted actions are called on-policy samples denoted by the superscript p and all samples created by noise are called off-policy samples denoted by f.

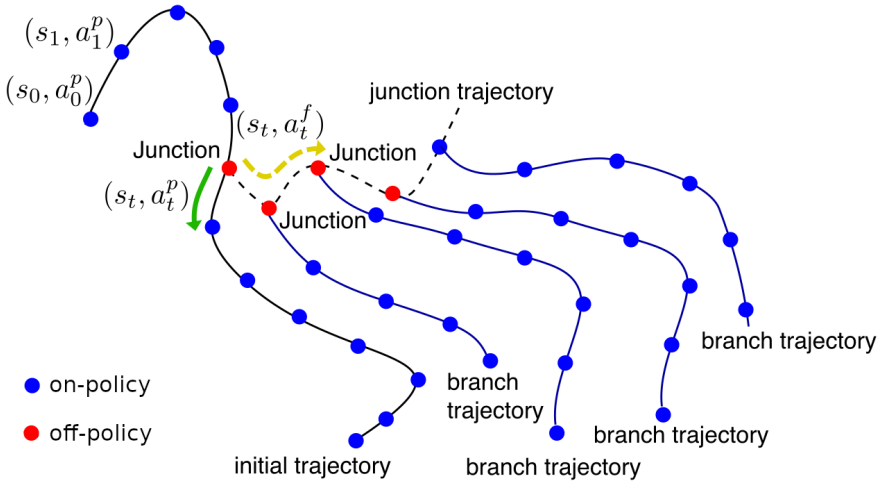


Figure 2.6: The exploration strategy consists of initial trajectories paired with branch trajectories, which are generated by adding noise to its corresponding initial trajectory.[4]

Chapter 3

Methods

The structure of this chapter is based on the research questions listed in Section 1.3. Consequently, a summary of the full learning system is provided in Section 3.1, how its components work together and how the exploration and exploitation strategy are used. In Section 3.2, the mathematical terms and notations which are used to formulate the learning process are provided as an overview. The main Section 3.3 contains the implementation of the exploration and exploitation strategy and gives an in-depth explanation of the optimization process.

3.1 System Setup Overview

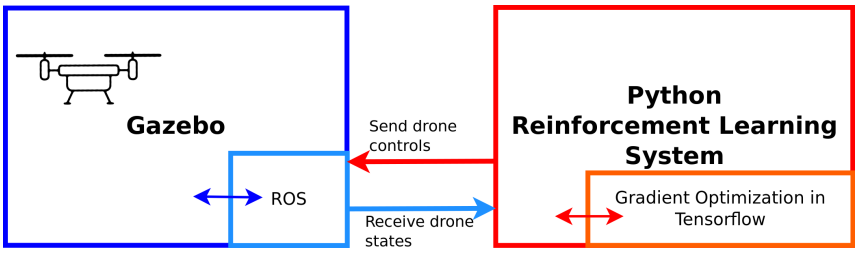


Figure 3.1: The core Python code (right) instantiates the neural networks with Tensorflow and communicates through ROS with the Gazebo simulation environment (left).

Figure 3.1 explains, how the main components are connected. The system consists of two core components: The simulator and the controlling program. The red block on the right symbolizes the core Python component of the system. Its task is to control the program flow including the initializing of all important functions for the data collection and the training procedure, the initialization of the Tensorflow neural networks and the communication channel for Gazebo over ROS.

The Reinforcement Learning system consists of two neural networks. One for the policy and one as a value function. This was inspired by the ETH Zurich's research paper, which is described in Section 2.4. These neural networks are both structured as previously described in Figure 2.5. For the input, a state of 18 elements, consisting of the rotation matrix for the orientation, a relativ position to the target and the angular and linear velocities, are used. Two hidden layers are then initialized, each with 64 neurons and finally four outputs neurons in the policy network and one output neuron in the value network. The policy is learned by the policy network, hence its name. It is dedicated to learn to reliably predict the thrust of the Quadcopter rotors through input states.

The value network is only used as a factor in the training / optimization process of the policy network. This too resembles the structure implemented by the ETH research team. Because neural networks are used for the Reinforcement Learning system, in this thesis it can be considered a Deep Reinforcement Learning setup as mentioned in Section 2.3. For this reason the Reinforcement Learning scenario Figure 2.4 was modified to fit the scenario of this thesis. This is illustrated in Figure 3.2.

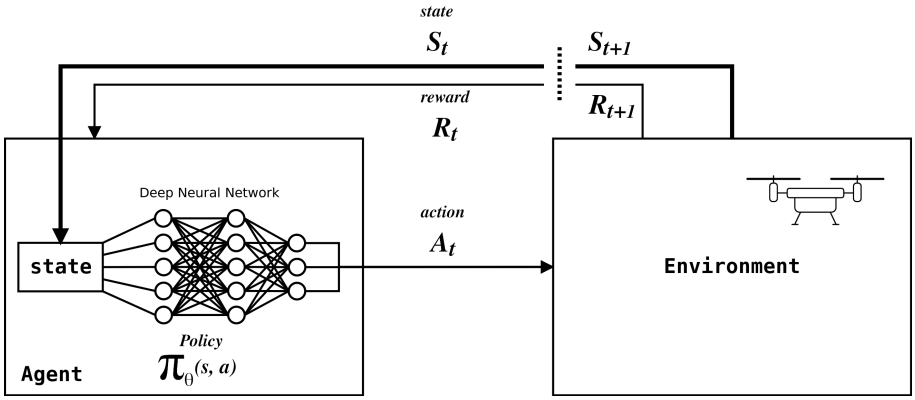


Figure 3.2: Reinforcement Learning cycle of the scenario in this thesis.

The policy $\pi(s, a)$ of this thesis is illustrated by the neural network in the agent. It processes the inputs generated by its own actions taken in the simulation environment. An in-depth explanation of the data collection and exploration strategy can be found in Section 3.3.2. More about the implementation and training procedure can be read in Section 3.3.3 and 3.3.4.

3.2 Terminology & Notation

As mentioned before, this thesis is based on a Reinforcement Learning system of another paper. This system consists of two neural networks which are notated with $\pi(s|\theta)$ and $V(s|\eta)$ for the policy and value network respectively. The policy network parameters are symbolized by theta (θ). The value network parameters are symbolized by eta (η). Furthermore, several terms are important to understand the mathematical basis of the optimization of the Reinforcement Learning system. For this reason, table 3.1 contains the most important terms and their description.

Terms	Description
$\pi(s\|\theta)$	Policy network with input state s and parameters θ
$V(s\|\eta)$	Value network with input state s and parameters η
v_i	Value function for value network learning
r_t	Cost function to evaluate the quality of a given state
p and f	These superscripts stand for on- and off-policy values respectively
i, j, k	Time, iteration and junction index respectively
γ	Discount value for attaching weight to components
Σ	Covariance Matrix for generating the noise and calculating every policy optimization step.

Table 3.1: Important terms for the Reinforcement Learning system in this work

3.3 Reinforcement Learning System

The idea of this sections is to give an in-depth explanation of the implementation of the Reinforcement Learning system. Firstly, the whole system flow is explained by reference to Figure 3.3. The main components of the flowchart are then described in greater detail in their respective paragraphs.

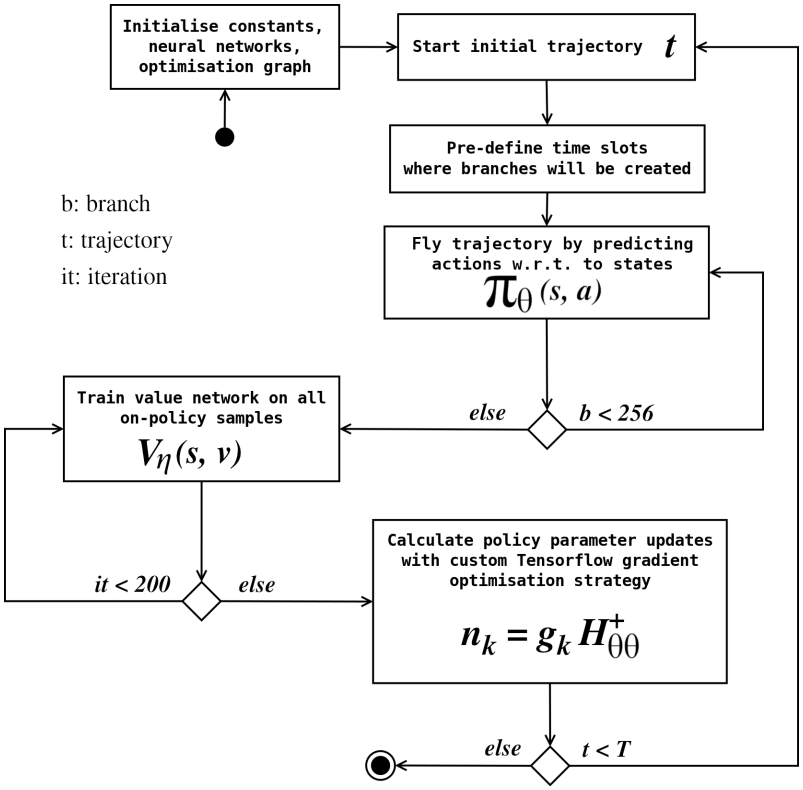


Figure 3.3: The main flowchart of the learning system, consisting of the exploration strategy and the training procedure.

For the main Python code to run properly, the Gazebo simulator must be running with the Hummingbird Quadcopter. The first step is the initialization of the neural networks and the required constants like the discount value γ or the number of branches per initial trajectory. The second step, as illustrated in Figure 3.3, is to start an initial trajectory. Before the initial flight is predicted, random positions in time are chosen for the branches. The exact pose, consisting of the orientation, position and velocities of the Quadcopter, has to be saved at the branching points while flying the initial trajectory to exactly recreate these poses later on and add noise for the creation of the junctions, as seen in Figure 2.6. All branching trajectories are predicted by the policy network $\pi_\theta(s, a)$ with the state of the Quadcopter as input and the rotor thrust as action outputs. After flying all branches, the values of every on-policy generated state is calculated. Meaning, only the states are used that were created by actions from the policy network and not the additive noise. The value network $V_\eta(s, v)$ is trained with standard gradient descent on the values for 200 times. After optimizing the value function in the form of the value network, the policy network's parameters θ are optimized by calculating the per-sample natural gradient n_k for every junction k with a custom optimization algorithm.

3.3.1 System initialization

The system depends on several important constants which are initialized in the first lines of code.

```
# initialize ROS node for communication with Gazebo
rospy.init_node('hummingbird')
# constants for controlling the program flow
# trajectories to record
TRAJECTORIES_N = 256
# branches to record per initial trajectory
BRANCHES_N = 128
# initial trajectory length
TRAJ_INTIAL_LENGTH = BRANCHES_N * 2
# number of times noise will be added at a junction
NOISE_DEPTH = 2
# value iterations and loss precision limit
VALUE_ITERATIONS = 500
VALUE_LOSS_LIMIT = 0.0001
# discount value for the policy network training
DISCOUNT_VALUE = 0.99
# noise covariance for generating noise
NOISE_COV = np.array(np.diag([600,600,600,600]), dtype=np.float32)
def NOISE(): return np.diag(np.diag(np.random.normal(80, 10, 4)) + NOISE_COV)
```

Code Listing 7: Initialization of the main constants for the system setup.

Due to the architecture of Tensorflow, the initialization and training of two neural networks is not entirely straightforward. While training the neural networks with an optimizer like the **`GradientDescentOptimizer`**, the function **`tf.trainable_variables()`** is used to obtain the parameters. The problem is that the function returns all possible trainable

variables of the Tensorflow graph. To ensure correct behavior while training the networks, they have to be initialized in separate graphs. Splitting them guarantees that the correct parameters are called when the optimization process is executed for either of them in the same program flow. For this reason, a policy graph and a value graph are initialized, as can be seen in the following listing.

```
# create policy network with own tensorflow graph
policy_graph = tf.Graph()
policy_sess = tf.Session(graph=policy_graph)

policy_net = NN(shape=[18,64,64,4], graph=policy_graph)
policy_net.create_model()

# create value network with own tensorflow graph
value_graph = tf.Graph()
value_sess = tf.Session(graph=value_graph)

value_net = NN(shape=[18,64,64,1], graph=value_graph)
value_net.create_model()
```

Code Listing 8: Initializing the neural networks in their own graphs

For the optimization of the value network, a traditional Gradient Descent Optimizer is used with Huber loss as its cost function. More will be explained in Section 3.3.3. The neural network uses a Gradient Descent Optimizer to apply only the gradients calculated by a custom natural gradient implementation. For the creation of the two neural networks, a short class `NN` for the generic generation of neural networks with different sizes was written.

3.3.2 Exploration Strategy

For the exploration of the environment, the Monte Carlo method is used. Meaning the collection of random samples for a deterministic problem.[21] One collection cycle is started after every policy optimization step. A randomly initialized policy network is used for the first collection cycle. The collection of trajectory samples can be split into several step, as presented in Algorithm 1.

Algorithm 1: Monte Carlo Trajectory Collection Algorithm

1. Generate random position and velocities for Quadcopter
2. Define random points of time for a set amount of branches
3. Fly initial trajectory with a set length of predictions
    - 3.1 While flying save the exact poses at every branching point
    - 3.2 Save all states and actions
4. Set Quadcopter to the saved branching points
    - 4.1 Add random noise
    - 4.2 Fly branch trajectory with a set length of predictions

At first, a random position, orientation and velocity for the Quadcopter, are generated and applied. Afterwards the future branch positions are predefined. For a specified length,

states are processed by the policy network and the made predictions are used as actions for the rotors of the Quadcopter. At every predefined branch position, the full pose with orientation, position and velocities of the Quadcopter is saved into a branch list, while flying the initial trajectory. This list is than used to add noise and create the branch trajectories. After flying and predicting a full cycle of branches, the training process is started. This creation of branches is visualized in Figure 3.4.

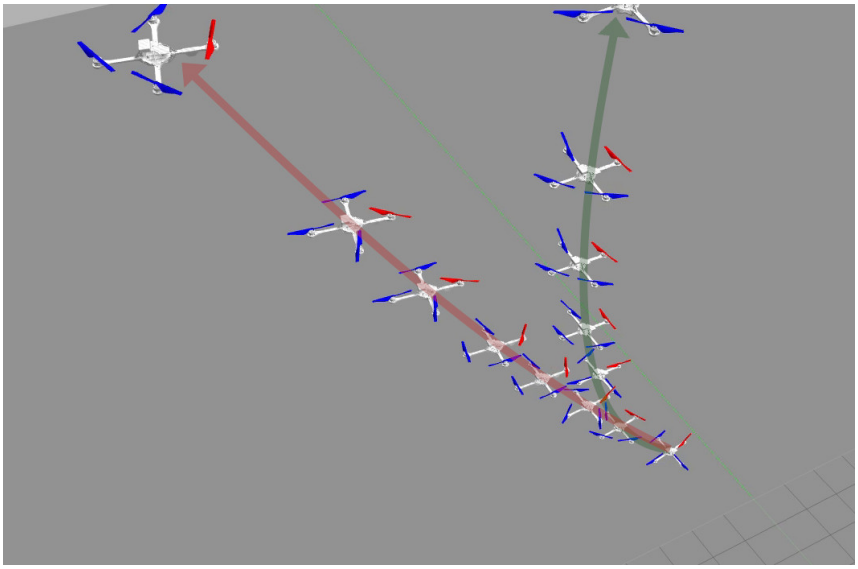


Figure 3.4: Creation of branching trajectories in Gazebo, thus creation junctions for the policy training.

To perform the exploration strategy explained above, two classes were written for the communication with Gazebo. As clearly apparent a communication channel between the simulation environment Gazebo and the Python code is needed to save the states and actions taken to train the networks in the next step. With the architecture of ROS in mind, receiving and sending data from and to Gazebo was split into two classes. The `DronePublisher` and the `DroneSubscriber` classes. The publisher class has several functions to control the simulator, like resetting the position of all models, pausing and unpausing the physics simulation and setting the thrust of the Quadcopters rotors. The subscriber class has fewer functions because there are only two instances when the training system needs information from the simulator. Firstly, to retrieve the current state of the Quadcopter to feed the neural networks and secondly the full pose of the Quadcopter to generate the branch trajectories.

3.3.3 Value Network Training

The value network has a rather traditional or straightforward optimization strategy. A predicted output for a given input is compared with the desired output, and the parameters of the network are tweaked with a Gradient Descent Optimizer. Gradient Descent is an iterative algorithm for finding a minimum of a cost function. Yet, it does not guarantee finding the global minimum. With that in mind, the value network's parameters are trained

with a Gradient Descent Optimizer by feeding it with states out of a trajectory and comparing its predictions with the desired outputs v_i calculated with Equation 3.2. The Huber loss is used as the loss function for the optimizer. This loss function is a combination of two other popular loss function: Square loss and absolute loss. The benefit of Huber loss is that it is not as sensitive to outliers then square loss.[22] The definition of the loss function can be found in Equation 3.1.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), \text{otherwise} \end{cases} \tag{3.1}$$

The desired output consists of a sum of weighted costs and the prediction of the termination state of the corresponding trajectory. With this approach, a scalar value can be generated to symbolize the quality of a state. The smaller the value, the better the quality.

$$v_i = \sum_{t=i}^{T-1} \gamma^{t-i} r_t^p + \gamma^{T-i} V(s_T|\eta) \tag{3.2}$$

The implementation of the value function in Equation 3.2 is shown in the following code listing.

```
def value_function(sess, value_net, costs, states, i):
    values = 0
    T = len(costs)
    value_factor = value_net.model().eval(session=sess, \
                feed_dict={value_net.input:[states[T-1]]})[0][0]
    for t in range(i, T):
        v = (DISCOUNT_VALUE**(t-i) * costs[t])
        values += v
    return values + (DISCOUNT_VALUE**(T-i) * value_factor)
```

Code Listing 9: Python implementation of the value function

The cost function r_t is calculated by weighting the position p_t, angular and linear velocities w_t, v_t respectively and the action predicted by the policy network a_t of a given state.

$$r_t = 4 \times 10^{-3}||\boldsymbol{p}_t|| + 2 \times 10^{-4}||\boldsymbol{a}_t|| + 3 \times 10^{-4}||\boldsymbol{w}_t|| + 5 \times 10^{-4}||\boldsymbol{v}_t||, \tag{3.3}$$

```python
def cost(position, action, angular, linear):
    position = 4 * 10**(-3) * np.linalg.norm(position)
    action = 2 * 10**(-4) * np.linalg.norm(action)
    angular = 3 * 10**(-4) * np.linalg.norm(angular)
    linear = 5 * 10**(-4) * np.linalg.norm(linear)
    return position + action + angular + linear
```

Code Listing 10: Python implementation of the cost function

The value function is designed to train the value network to predict the quality of a given state in relation to the termination state of the trajectory. The value network is then used to compare different trajectories, while training the policy network later on. The network is initialized with random normal distributed weights and biases. It is trained with all on-policy samples of the trajectories predicted in one collection cycle. Training is performed after every trajectory collection cycle for 200 iterations, to increase the precision of the network's predictions.

3.3.4 Policy Network Training

The primary goal of this Reinforcement Learning setup is to train a policy network to control a Quadcopter independently from human interaction in a stable manner. Consequently, the following paragraphs contain the most essential steps of the policy optimization process. Due to the mathematical complexity of the optimization process, the details of its inner workings are not fully explained in this thesis. For a more in-depth explanation, the research paper of ETH Zurich is recommended.[4]
To get a basic understanding of the policy training, Figure 3.5 has to be taken into consideration. The idea is to compare the trajectories before and after adding noise at each junction (red dots). As explained in Section 3.3.2, after adding noise to the initial trajectory, branches are recorded by predicting a deviating trajectory from the initial one. This comparison between trajectories A^π is obtained through Equation 3.4.

$$\begin{aligned} A^\pi(s_i, a_i^f) &= r_i^f + \gamma v_{i+1}^f - v_i^p, \\ \bar{L}(\theta) &= \sum_{k=0}^{K} A^\pi(s_k, \pi(s_k|\theta)), \\ \theta_{j+i} &\leftarrow \theta_j - \frac{\alpha}{K} \sum_{k=0}^{K} n_k, \\ \text{s.t. } (\alpha n_k)^T D_{\theta\theta}(\alpha n_k) &< \delta, \forall k, \end{aligned} \tag{3.4}$$

This optimization strategy originates in the State-action-reward-state-action algorithm (Sarsa).[23] Comparing Equation 3.4 with Equation 3.5, the similarities become apparent.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.5}$$

The Sarsa algorithm updates a policy based on taken actions, by determining an error through the calculation of the cost function r_t for a given state s_t and its corresponding action a_t. The difference between the possible reward $Q(s_t, a_t)$ at the given state and the real reward at the next state $Q(s_{t+1}, a_{t+1})$ is added to the cost function r_t. That result is weighted with a learning rate α, which determines the rate at which the newly generated error is used to applied an update to the existing parameters of the policy. The discount value γ determines, how important the future reward $Q(s_{t+1}, a_{t+1})$ is incorporated. In case of the policy network, it is desired that it learns to anticipate the ongoing trajectory from its current position in relation to its target. Due to that, a high discount value near one is chosen. A value matching one or above could lead to a diverging effect of the policy parameters, which is not a desirable scenario. The most significant difference to the introductory example in Section 2.3 is that no rewards are defined. For that reason, a value function has to be added to the gradient policy. In the case of Sarsa, the value function is $Q(s, a)$ and it has to learn the quality of states to be used as the primary feedback of the policy. In this thesis, the value function is substituted by the value network $V(s, \eta)$.
In the following, the structure of the neural network is explained in more detail. In this thesis, fully connected neural networks are used. Meaning that the neuron count of each layer has to be multiplied and added up to obtain the weight count. Given that, between the input layer and the first hidden layer $18 \times 64 = 1152$ weights and also 64 biases exist. Between the first and second hidden layer $64 \times 64 = 4096$ weights and additional 64 biases and between the second hidden layer and the output layer $64 \times 4 = 256$ weights and 4 biases. All in all, the policy network is made up of $1152 + 64 + 4096 + 64 + 256 + 4 = 5636$ trainable parameters which are symbolized by θ.
To recapitulate, the objective is to calculate a per-sample natural gradient n_k for every junction in the collected trajectories. The full process for one optimization cycle is visualized in Algorithm 2, and every single step is subsequently explained in more detail.

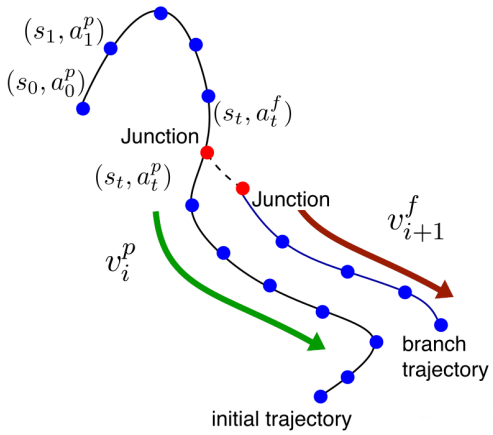


Figure 3.5: One recorded junction example. The green arrow marks the on-policy trajectory before adding noise used by v_i^p and the brown arrows trajectory originates from the added noise and is used by v_{i+1}^f.

Algorithm 2: Policy network training

for k in K

1. Calculate Jacobian $J = \frac{\partial a}{\partial \theta}$ for network parameters w.r.t. actions
2. Calculate pseudoinverse of Hessian $H^{+}_{\theta\theta}$ with SVD
3. Calculate A^{π} with cost function r_t and value predictions v^{p}_{i} and v^{f}_{i+1}
4. Calculate per sample gradients $g_k = \frac{\partial A}{\partial a} J$
5. Calculate per sample natural gradients $n_k = g_k H^{+}_{\theta\theta}$

end for

6. Apply parameters update with $\theta_{j+1} \leftarrow \theta_j - \frac{\alpha}{K} \sum_{k=0}^{K} n_k$

1. Calculate Jacobian: Firstly, for the optimization step, a Jacobian matrix has to be calculated for every junction k. Thus creating a matrix in a shape of 4 x 5636, each row consisting of the partial derivatives of all parameters w.r.t. one of the four actions. In general, a Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function. In this case, the vector-valued function is the policy network which predicts a 4-dimensional vector. So the Jacobian matrix can be defined by $J = \frac{\partial a}{\partial \theta}$. The Tensorflow implementation for the computation of the matrix is shown in Listing 11. Conditioned by the architecture of Tensorflow, the parameters of the network are divided up into variables. Two variables for each layer except the input layer, one for the weights and one for the biases. Leading to six Tensorflow variables needed for the neural network structure used in this thesis. Due to that circumstance for the calculation of the Jacobian J these variables have to be concatenated and flattened to one array of parameters.

```
J = tf.stack([tf.concat([tf.reshape(tf.gradients(action_pred[:, idx], param)[0],
                     [1, -1]) for param in variables], 1) for idx in range(4)],
                     axis=1, name='jac_Action_wrt_Param')[0]
```

Code Listing 11: Computational Tensorflow graph instantiation for the Jacobian matrix

2. Calculate pseudoinverse of Hessian: The Hessian matrix of the policy network can be calculated as the product of the transposed Jacobian matrix, the inverse of the noise matrix $D_{aa} = \Sigma^{-1}$ and the Jacobian matrix, thus defining the Hessian matrix as $H_{\theta\theta} = J^T D_{aa} J$. However the per-sample natural gradients n_k are obtained by solving the equation $H_{\theta\theta} n_k = g_k$, the inverse Hessian is required. However, since the computation of the inverse for a Hessian matrix containing all 5636 parameters and making it the shape of 5636×5636, is prohibitively expensive. For this reason, only the pseudoinverse of the Hessian is calculated by using Singular Value Decomposition (SVD) to save computation time. This leads to Equation 3.6, where L_{aa} is a Cholesky factor of D_{aa}. As D_{aa} has to be symmetric and positive-definite, the computation of the pseudoinverse becomes even more efficient by calculating only thin SVD on $L_{aa} J = V \Sigma_v U^T$. All this is done with a few Tensorflow functions as exemplified in Listing 12

$$\begin{aligned} H^{+}_{\theta\theta} &= (J^T D_{aa} J)^{+} = (J^T L_{aa} L^T_{aa} J)^{+} \\ &= (L^T_{aa})^{+} ((L^T_{aa} J)^{+})^T = V \Sigma^{+}_v U^T U \Sigma^{+}_v V^T \\ &= V (\Sigma^{+}_v)^2 V^T \end{aligned} \tag{3.6}$$

```
s_, U_, V_ = tf.svd(tf.matmul(tf.cholesky(tf.matrix_inverse(NOISE_COV)), J))
h00_pinv = tf.matmul(
                tf.matmul(
                    V_,
                    tf.matrix_diag(tf.square(1/s_))
                ),
                tf.transpose(V_)
            )
```

Code Listing 12: Calculation of the Hessian pseudoinverse with the help of thin SVD

3. Calculate cost function A^{π}: The policy cost function A^{π}, consists of three elements, as defined in Equation 3.4. Firstly, the value cost function r_t which is calculated with the noise as the action vector as defined through the denotation r^f in Equation 3.4. The computation of r^f can be seen in lines $1-6$ in Listing 13. The other two elements are the values of the trajectory before the noise v_i^p and after the noise addition v_{i+1}^f. The computation graph for v_{i+1}^f is initialized in lines $8-19$. The initialization for v_i^p looks the same, except for the iteration start in line 8 and the parameters for the body of the while loop. Due to that, v_i^p is not explicitly displayed. The implementation is based on the value function in Equation 3.2. Finally, in line 25 the elements of A^{π} are joint together, as can be seen in Equation 3.7.

$$A^{\pi}(s_i, a_i^f) = r_i^f + \gamma v_{i+1}^f - v_i^p \tag{3.7}$$

```
position = 4 * 10**(-3) * tf.norm(position)
angular = 3 * 10**(-4) * tf.norm(angular)
linear = 5 * 10**(-4) * tf.norm(linear)
action = 2 * 10**(-4) * tf.norm(action_noise)

rf = position + action + angular + linear

iterf = tf.constant(1, dtype=tf.int32)
vf = tf.constant(0, dtype=tf.float32)
cf = lambda i, vf: tf.less(i, number_of_states)

vf_w = tf.while_loop(cf,
                     gen_body(policy_net.input[:number_of_states],
                              number_of_states,
                              action_pred,
                              termination_state_value,
                              0),
                     [iterf, vf])[1]
tmp = DISCOUNT_VALUE**tf.cast(number_of_states, dtype='float')
tmp *= termination_state_value
vf_w += tmp

...

policy_net.A = rf + DISCOUNT_VALUE * vf_w - vp_w
```

Code Listing 13: Computation of the policy cost function A^{π} as it has been defined in Equation 3.4

4. Calculate per sample gradients: After the calculation of A^{π}, the per-sample gradients g_k can be calculated by multiplying the Jacobian matrix of the parameters w.r.t. the actions with the Jacobian matrix of the actions w.r.t. cost A^{π}. The computational graph for $g_k = \frac{\partial A}{\partial a} J$ is defined in Listing 14.

```
grads = tf.gradients(policy_net.A, action_pred)[0]
gk = tf.matmul([grads[1]], J)
```

Code Listing 14: Computation of the per-sample gradients g_k

5. Calculate per sample natural gradients: With the intermediate results of step $1-4$, the equation $n_k = g_k H^{+}_{\theta\theta}$ can be solved. Which leads to a straightforward implementation in Listing 15 with just one multiplication of the Hessian pseudoinverse and g_k.

```
policy_net.nk = tf.matmul(gk, h00_pinv)
```

Code Listing 15: Computation of the per-sample natural gradients n_k

6. Apply parameters update: After iterating through steps $1-5$ for every junction k, all n_k are summed up and weighted with a learning rate. This weighted sum is then subtracted from the parameters of the network, as defined in Equation 3.4. Because the parameters of the network are divided up into Tensorflow variables, the concatenated gradients have to be transformed into six chunks and can then be applied with an optimizer, as can be seen in Listing 16.

```
param_update = (0.01/(len(policy_traj) - 1)) * np.array(param_update)

start = 0
grads_vars = []
for v in variables:
    if len(v.shape) == 2:
        limit = v.shape[0].value * v.shape[1].value
    else:
        limit = v.shape[0].value
    grads_vars.append((np.reshape(param_update[start:start+limit], v.shape), v))
    start = start + limit
policy_net.optimizer.apply_gradients(grads_and_vars)
```

Code Listing 16: Computation of the per-sample natural gradients n_k

To support the training process even further three PD controllers, one for each axis in the 3-dimensional space, were implemented to help the Quadcopter being more stable from the beginning and guide its orientation. Another training assistance is an added bias to the rotor thrust to ensure that it takes off from the start. All three are added together and used as the rotor thrusts. The PD controllers are only used for training purposes. Therefore, they are not supposed to be used with the final policy. The policy network should develop a much more sophisticated behavior.

Chapter 4

Experiments

The training was done on an Amazon EC2 P2 Instance which provides a Nvidia Tesla K80 GPU.[24] This setup was used because Tensorflow supports Nvidia GPU's to accelerate the computation of numerical calculations, as used in both neural network optimization processes.

The evaluation of the training system was done by verifying the results of the primary parts of the optimization processes. Firstly, the value network was evaluated by looking at the loss over time, as illustrated in Figure 4.1.

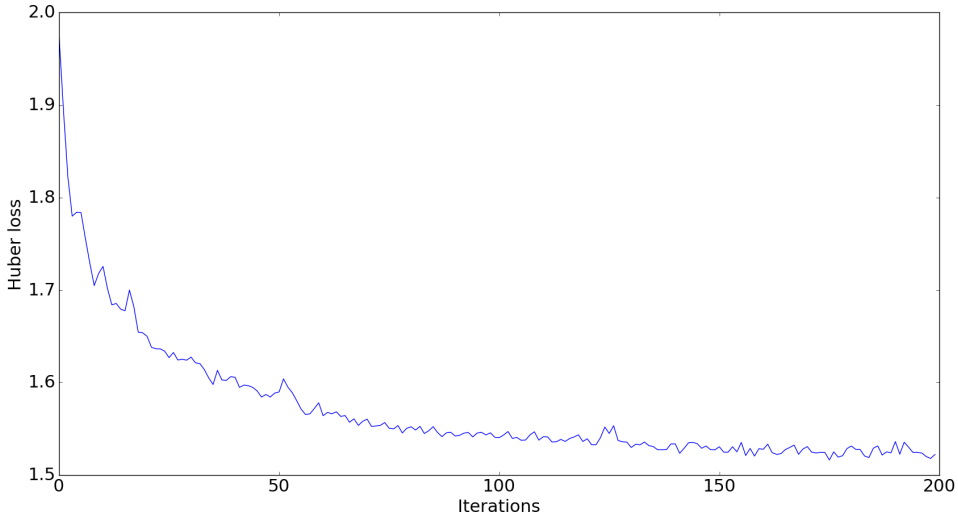


Figure 4.1: Huber loss over 200 iterations for a single learning step of the value network optimization.

The Huber loss consistently goes down for 200 iterations of training on the trajectories of one collection cycle. However, between several training cycles or epochs, the loss sometimes jumps but still goes down consistently, as can be seen in Figure 4.2.

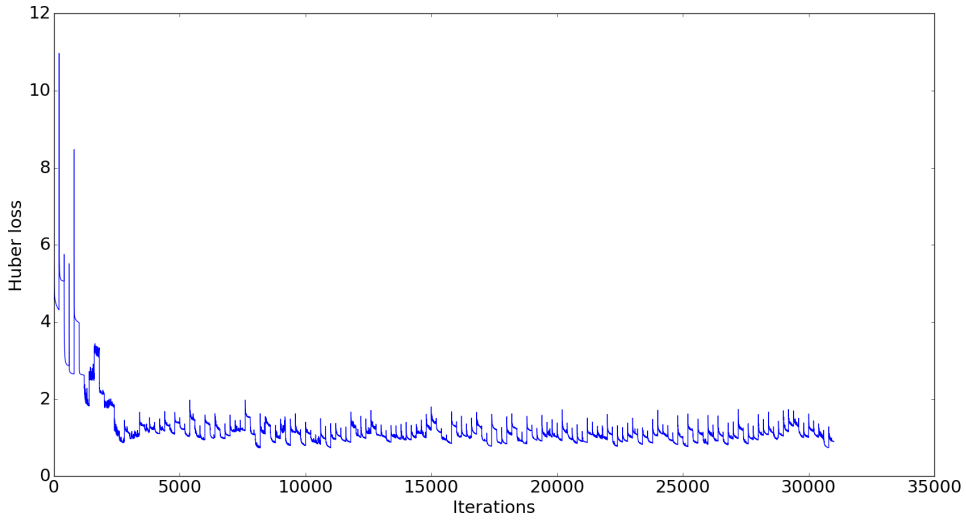


Figure 4.2: Huber loss over 155 training cycles of the value network, each with 200 iterations. Thus making it 31000 iterations in total.

After the verification of the value network training procedure. The policy network had to be tested. For the training of the policy network, three PID controllers were used to assist a nominal orientation from the beginning, as mentioned in Section 3.3.4. To be able to make a reliable statement, the policy network was trained a few times. The loss in all cases showed no indication of convergence. The corresponding policy loss to Figure 4.2 is illustrated in Figure 4.3.

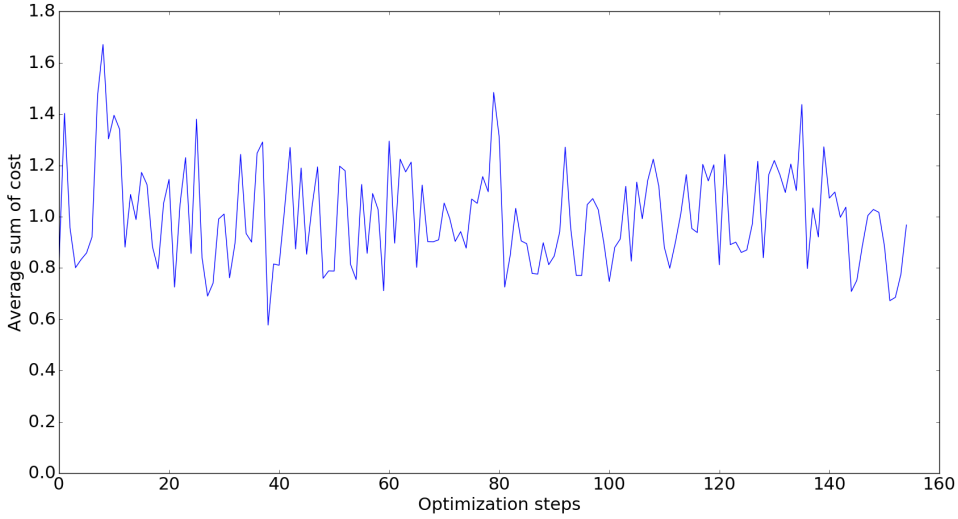


Figure 4.3: Policy loss over 155 training cycles. The loss stagnates and shows no indication of convergence.

Contrary to the original paper, the successful policy network's optimization was not accomplished. For the sake of finding the reason why not, several different trajectory flight

length were tested. All leading to similar results, as can be seen in Figure 4.3. Furthermore, the execution time and its distribution were measured. A comparison between the execution distribution of this thesis and the measurements from the ETH research is illustrated in Figure 4.4.

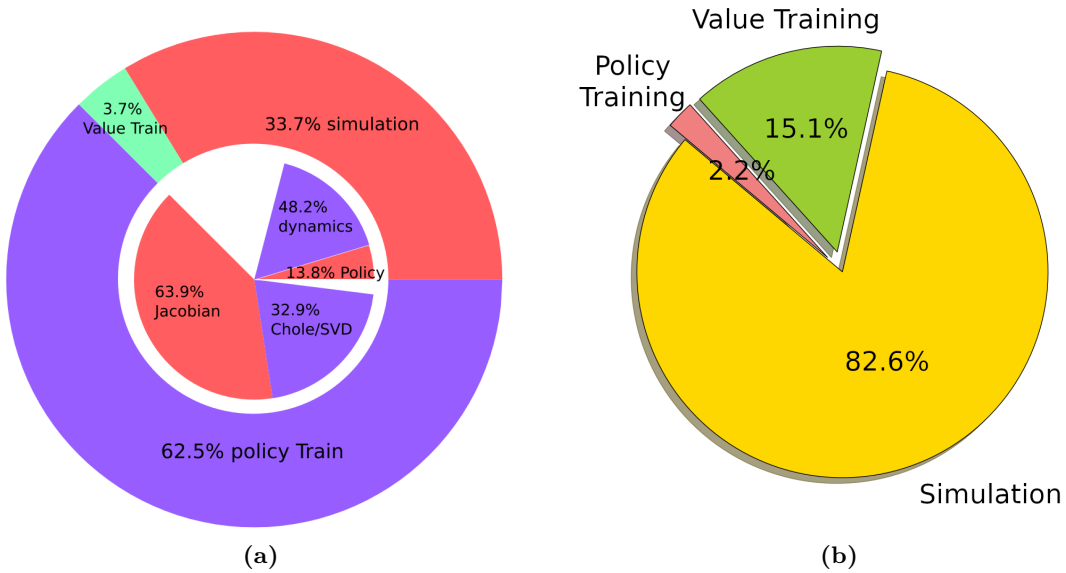


Figure 4.4: Comparison between the execution time of the ETH Zurich system (a) and the implementation in this thesis (b).[4]

As clearly apparent the simulation is the main bottleneck of this thesis. Since Gazebo is a real-time simulator and it is built to approximate the reality as closely as possible the collection of samples in the form of trajectories takes hours instead of only minutes as described in the implementation of the ETH. Another interesting aspect is the ratio between the value network training and the policy network. In the present work, the value network takes nearly seven times the computation time of the policy network optimization. Compared to the measurements of the ETH, their policy optimization took almost 17 times the computation time of the value network. The reason for this discrepancy could not be clarified in within the time frame of this thesis. However, the reason could be a misunderstanding of the word *iterations*. It could be that they did not train the value network on all trajectories of a collection cycle for 200 iterations but instead only trained the network on 200 trajectories for a single time per learning step.

Chapter 5

Conclusion

The objective of this thesis was to implement a Deep Reinforcement Learning system that is capable of learning to fly a Quadcopter without human interaction. For this task, Tensorflow was used to process the primary optimization steps and to initialize the neural networks. The Gazebo simulator was used to simulate the trajectories for the learning. The communication between Tensorflow and Gazebo was established by utilizing ROS.
A Deep Reinforcement Learning system can learn the difficult task of flying a Quadcopter autonomously. That was demonstrated by a research team of the ETH Zurich. However, to evaluate the system's capabilities a complex mathematical reward function has to be developed and thoroughly tested which is why the previously published paper from the ETH was used as the primary guideline for this thesis.

For the most part, the implementation of this thesis works properly. Trajectories can be generated by predicting the thrust to a given state with the policy network. The value network can be trained, and its loss consistently decreases over time. The policy optimization was tested but showed no indication to converge any time soon, as illustrated in Figure 4.3. For this reason, the single calculation steps of the policy optimization were tested and most of them could be verified to work as expected. It is assumed that the main problem lies in the calculation of the policy reward function A^{π}, as seen in Equation 3.4. That should be evaluated in more detail in future work.

The implementation was kept very close to the paper from the ETH Zurich. The most significant difference to their implementation is the simulation environment. They developed a simulator which only roughly follows the most basic rules of physics and ignores a lot of factors. The physics engines in Gazebo, on the other hand, are made to resemble the reality as close as possible. For this reason, the simulation in Gazebo is quite slow and cannot speed up a lot because of the computational limits. In retrospective, Gazebo was not the best choice for the endeavor of this thesis because it takes a lot of hours to collect trajectories.

If the policy network optimization is fixed in the future, the implementation of this thesis should work properly and will lay the basis for the on-going development of this Reinforcement Learning system. In the future, the primary goal is to reach a point when the system is reliable enough to be tested on a real Quadcopter.

References

[1] (May 2018), [Online]. Available: https://www.forbes.com/sites/danwoods/2017/01/25/10-killer-use-cases-what-drones-as-a-service-can-do-for-your-business/ (visited on 05/22/2018) (cit. on p. 1).

[2] (May 2018). Pid controller wikipedia, [Online]. Available: https://en.wikipedia.org/w/index.php?title=PID_controller&oldid=842411324 (visited on 05/22/2018) (cit. on p. 1).

[3] (May 2018). Pid for dummies, [Online]. Available: https://www.csimn.com/CSI_pages/PIDforDummies.html (visited on 05/22/2018) (cit. on p. 2).

[4] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning", *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, Oct. 2017 (cit. on pp. 3, 14, 15, 23, 31).

[5] (Nov. 2017), [Online]. Available: http://wiki.asctec.de/display/AR/AscTec+Hummingbird (visited on 06/21/2018) (cit. on p. 3).

[6] (May 2018). Reinforcement learning, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=837645445 (visited on 05/23/2018) (cit. on pp. 3, 10).

[7] (May 2018). Gazebo, [Online]. Available: https://www.gazebosim.org/ (visited on 05/23/2018) (cit. on p. 4).

[8] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "Robot operating system (ros): The complete reference (volume 1)", in, A. Koubaa, Ed. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23 (cit. on p. 4).

[9] (May 2018). Ros, [Online]. Available: http://www.ros.org/ (visited on 05/23/2018) (cit. on p. 5).

[10] (May 2018). Tensorflow, [Online]. Available: https://www.tensorflow.org/ (visited on 05/23/2018) (cit. on p. 6).

[11] (May 2018). Tensorflow wikipedia, [Online]. Available: https://en.wikipedia.org/w/index.php?title=TensorFlow&oldid=842571883 (visited on 05/23/2018) (cit. on p. 6).

[12] (Jun. 2018), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=844753106 (visited on 06/12/2018) (cit. on p. 8).

[13] (Dec. 2017), [Online]. Available: http://news.mit.edu/2017/reading-neural-network-mind-1211 (visited on 06/12/2018) (cit. on p. 8).

[14] (Apr. 2017), [Online]. Available: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (visited on 06/12/2018) (cit. on p. 8).

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2017, pp. 1–3. [Online]. Available: http://incompleteideas.net/book/bookdraft2017nov5.pdf (cit. on p. 10).

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning", *CoRR*, vol. abs/1312.5602, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602 (cit. on p. 10).

[17] (Jun. 2016), [Online]. Available: https://deepmind.com/blog/deep-reinforcement-learning/ (visited on 06/21/2018) (cit. on p. 10).

[18] (Jun. 2018). Multi-armed-bandit, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multi-armed_bandit&oldid=846442410 (visited on 06/19/2018) (cit. on p. 11).

[19] (Jun. 2018). Multi-armed-bandit, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multi-armed_bandit&oldid=846442410#Semi-uniform_strategies (visited on 06/19/2018) (cit. on p. 11).

[20] (Jun. 2018). Multi-armed-bandit tensorflow, [Online]. Available: https://medium.com/@awjuliani/super-simple-reinforcement-learning-tutorial-part-1-fd544fab149 (visited on 06/19/2018) (cit. on p. 11).

[21] (Jun. 2018). Monte carlo method, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Monte_Carlo_method&oldid=846165133 (visited on 06/23/2018) (cit. on p. 20).

[22] (Jun. 2018). Huber loss, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Huber_loss&oldid=846559642 (visited on 06/23/2018) (cit. on p. 22).

[23] (Jun. 2018), [Online]. Available: https://en.wikipedia.org/w/index.php?title=State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action&oldid=829903292 (visited on 06/16/2018) (cit. on p. 23).

[24] (Jun. 2018). Amazon ec2 p2 instances, [Online]. Available: https://aws.amazon.com/ec2/instance-types/p2/ (visited on 06/25/2018) (cit. on p. 29).

List of Figures