Bachelor Thesis

Reinforcement Learning for Control of Flying Drones

Johannes M. Tölle

Date of Submission: 06. 09. 2023
Advisor: Prof. Dr. Carlo D'Eramo

Julius-Maximilians-Universität Würzburg
Reinforcement Learning and Computational Decision-Making

Declaration

I certify that I have authored this thesis, including all attached materials, constantly and without the use of any other aids than those specified.
All passages taken literally or analogously from published or unpublished works are clearly identified as such in each individual case, stating the source.
The work has not yet been submitted in the same or a similar form as a thesis.
I am aware that violations of this declaration and deliberate deception may lead to a grade of 5.0.

Würzurburg, July 16, 2023

Johannes M. Tölle

Contents

1 Introduction **2**

1.1 Related Work . . . 3

2 Basics **4**

2.1 Quadrocopter Basics . . . 5

2.1.1 Quadrocopter Flight Dynamics . . . 6

2.1.2 Autonomous Quadrocopter . . . 9

2.2 Reinforcement Learning Basics . . . 12

2.2.1 Classification of Machine Learning Areas . . . 13

2.2.2 Markov Decision Process . . . 15

2.2.3 V,Q Function & The Bellman Equations . . . 17

2.2.4 Neural Networks . . . 21

2.2.5 State of the Art Algorithms . . . 24

2.3 Pybullet Physics Simulator & Gym Pybullet Drones . . . 30

2.3.1 BaseAviary Class . . . 30

2.3.2 BaseSingleAgentAviary Class . . . 30

3 Flight-Control Concept **31**

4 Implementation **32**

4.1 WindSingleAgentAviary Environment Class . . . 33

4.1.1 Wind Class . . . 34

4.1.2 Modes . . . 36

4.1.3 Observation Space & State Space . . . 38

4.1.4 Action Space . . . 39

4.1.5 Reward Function . . . 40

4.1.6 Constraints . . . 41

4.1.7 Optimal Rewards . . . 42

4.2 Scripts & Evaluation Tools . . . 43

4.2.1 Learning Script . . . 43

4.2.2 Evaluation Script . . . 45

4.2.3 Evaluation Tools . . . 47

5 Evaluation **49**

5.1 Drone Model . . . 49

5.2 Setup . . . 50

5.3 Results . . . 50

5.3.1 Fixed Goal Modes . . . 51

5.3.2 Random Goal Modes 56
5.3.3 Wind Disruption 57

6 Conclusion & Future Work 58
6.1 Of Migrating to a real Drone 58
6.2 Of Hovering & Pathfollowing 59
6.3 Of Improvments in Self-Pased Curriculm Learning 60
6.4 Of Policy Hierarchy RL for Sub-Problem-Solving 61

List of Symbols 62

List of Abbreviations 67

Bibliography 68

List of Figures

2.1 "X" and "+" configuration of a quadrocopter with the motors ($M_1 ... M_4$), the flight controller(FC) and the rotational movement of the corresponding propeller (red clockwise, blue counterclockwise) 5

2.2 Visualization of Roll, Pitch and Yaw rotational movement (from left to right) and the corresponding difference in the rotational speed ω_i 7

2.3 Sketch of an autonomous quadrocopter flight control system 9

2.4 Concept onion of Artificial Intelligence [DJS20] 12

2.5 The main learning strategies of machine learning including the related approaches like classification and regression under the Supervised Learning, clustering under the Unsupervised Learning and a sketch of the Reinforcement Learning concept 13

2.6 The basic concept of RL: the agent takes an action based on the observed state and receives a numerical reward at each time step 14

2.7 Visualization of a simple MDP with 4 states (s_0, s_1, s_2, s_3) (blue nodes), 2 actions (a_0, a_1), state-action edges (green) and action-state edges (dark red) containing a tuple (possibility, reward) 16

2.8 The McCulloch-Pitts model of a single neuron uses a weighted sum with the inputs x_i and weights w_i and a non-linear activation function $g()$ in order to compute the output z [Bis94] 21

2.9 8 different Activation Functions. 22

2.10 Simple Feed-Forward Network with 3 neurons in the input layer, 2 hidden layers with 4 neurons and with 2 layers in the output layer 23

2.11 Overview of the different categories of RL algorithms with examples: PPO and DDPG Algorithm coloured because they were used in this work. . . . 24

3.1 Proposed Concept of autonomous flight control with the use of a intelligent agent implemented with the use of a NN. The output of the NN is denormalized in order to translate them to motor signals. With the use of the sensors and position estimation the input for the next control step are calculated. 31

4.1 Concept of the implemented software: the different tools(red), scripts(violet) that are used in order to learn the intelligent Agent robust flight control with the use of RL. 32

4.2 Visualization of three different wind fields. Vectors represent a force vector that impacts the drone in the matching position. (a) is a random vector field made with a 3D function, (b) a predefined trigonometric vortex vector field and (c) with a random linear vector field. 34
4.3 Visualization of the used reward function with the goal $(0, 0, 0.5)$ and a color scale for different positions in space. 40
4.4 Visualization of the reward functions over the total distance[m] 41

List of Tables

2.1 Example of all two step trajectories with the matching probability, reward and expected reward of the MDP shown in Figure 2.7 18

4.1 Overview of the initialization parameters of the WindSingleAgentAviary environment class. 33
4.2 The different Modes of a WindSingleAgentAviary environment. 36
4.3 The different ActionTypes with the corresponding dimensionality of the action, its range and how it is processed. 39
4.4 Overview of the Arguments ξ parsed to the Learning Script 43
4.5 Overview of the Arguments ξ parsed to the Evaluation Script 46
4.6 The evaluation metric. 47

5.1 Overview of the evaluated Policies on fixed goal modes 51
5.2 Evaluation of the Policies with mode 1 53
5.3 Overview of the evaluated Policies on random goal modes 56

List of Algorithms

1 Proximal Policy Optimization [SWD$^+$17] 26
2 Soft Actor-Critic [HZAL18] 29

3 Evenly distributed Sampling from a half sphere G_{HS} 37
4 Learning Script 44
5 Linear Curriculum Learning Algorithm 45
6 Evaluation Script 46
7 Update Algorithm of EvalWriter 47
8 Evaluation Algorithm of EvalWriter 48

Abstract

content...

1 Introduction

1.1 Related Work

2 Basics

The task to perform autonomous Drone-Control may be described as a rather challenging task, because it combines a couple of different topics from various areas of computer science:
Quadrocotper possess rather unique flight dynamics and their flight control is its own unique research topic with large requirements. In order to achieve accurate flight control it is crucial to use sensor data with a high accuracy, process and apply them in realtime to the actuators. At the same time there is research to improve the error tolerance of flight control and make it more adaptive to a change in the environment or the payload. These Errors can occur due to turbulent conditions like wind.
Reinforcment Learning is a natural approach that is known to show more accurate results than the classic PID Controller in many use cases, because of the ability to approximate and generalize decision making. In Addition, a control implemented with an intelligent agent that had been learned with RL is often more robust to environment changes and is capable of adaption to these changes.
Although training a real drone from the beginning on flight seems more intuitive, this shows some risks. The first learning episodes show a bad behaviour that may risk the real hardware of the drone. A Reinforcement Learning episode is in need of quite complex calculations that would overstrain the hardware. Also, there can be be multiple parallel training episodes and simulation time must not be synced to the real world time. As a consequence of this the time the learning takes can be decreased at the cost of complexity to translate the simulation to the real world.

This chapter provides the needed basics in order to understand the idea of using RL for robust Drone-Control. Therefore, it provides a summary of the quadrocopter flight dynamics and sketches the autonomous quadrocopter concept. In addition essential parts of conventional autonomous quadrocopter are explained. Then, the basic concept of Markov Decision Processes and RL are presented. At last, the used Physics Simulator is explained as well as the extension that can be used for RL quadrocopter problems.

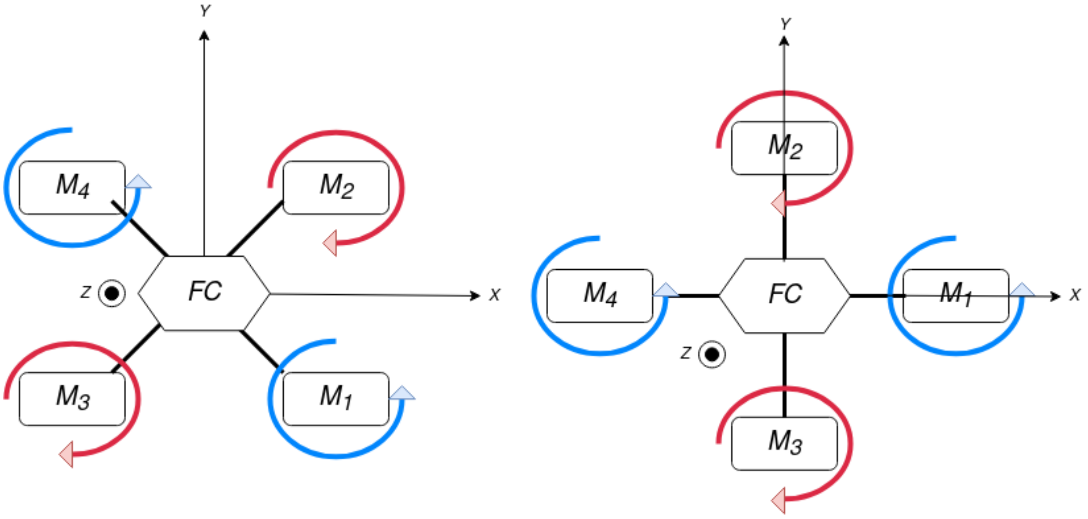


Fig. 2.1: "X" and "+" configuration of a quadrocopter with the motors ($M_1...M_4$), the flight controller(FC) and the rotational movement of the corresponding propeller (red clockwise, blue counterclockwise)

2.1 Quadrocopter Basics

A quadrocopter is a modern aircraft with a wide range of applications which is immensely popular in modern society. Typically, a quadrocopter can be described a an aircraft with 6 degrees of freedom (DOF), which consist of 3 translatational and 3 rotational DOF around the x, y, z axis with the corresponding angle Θ, ϕ, ψ.
As a consequence it has the ability to maneuver in 3D space with all possible rotations, although some rotations like a rotation of π around the x Axis ($\Theta = \pi$) may cause an inevitable crash.
The heart of the quadrocopter is the flight controller (FC) which consists of sensors and a MCU that controls the flight. Besides, the aircraft possesses 4 motors with corresponding propellers. There are different configurations how the motors can be placed in relation to the axis. The two most common configuration, the "X" and "+" configuration, can be seen in the above figure. The "+" configuration places the motors alongside the x and y axis. In contrast to that the each motor has an angle of 45° to the x and y axis in the "X" configuration.
Each of the propeller can be described by their rotational speed ω_i with $i \in [1...M]$ and spins either clockwise or counter clockwise (Figure 2.1). Depending on the configuration of the motors, the rotation speeds directly influence the *Euler Angles* Θ, ϕ, ψ and the upward thrust $\tilde{f}$. However, hereafter the most common "X" configuration is used to explain the quadrocopter further.

2.1.1 Quadrocopter Flight Dynamics

The quadrocopter flight dynamics mainly depend on the rotational speed ω_i of the propellers, which mainly produce the translatational and rotational movements. Besides, the thrust factor b which is a constant that mainly consists of propeller geometry and frame characteristics also influences the movement. Depending on the different ω_i, b an upward thrust $\tilde{f}$ and a rotation u_Θ, u_ϕ, u_ψ is produced that can be comprehended with the following set of formulas:

$$u_{\tilde{f}} = b \cdot \sum_{i=1}^{4} \omega_i^2 = b \cdot (\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \tag{2.1}$$

$$u_\Theta = b \cdot (\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \tag{2.2}$$

$$u_\phi = b \cdot (\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2) \tag{2.3}$$

$$u_\psi = b \cdot (\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) \tag{2.4}$$

Hover, Rise & Fall

Hovering can be described as holding a pose $p_{3D} = (x_p, y_p, z_p)$ above the ground, rising can be seen as increasing z_p and falling as decreasing z_p. Therefore, the quadrocopter should not have a rotation on x, y, z axis and mainly depends on Equation (2.1).
The product of the thrust factor b and the sum of the squared rotational speeds produces an upward thrust $\tilde{f}$. In order to hover equal rotational speeds must be applied to each motor/propeller in order to avoid a rotational movement, because:

$$u_\Theta = b \cdot (\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) = b \cdot 0 = 0 \tag{2.5}$$

$$u_\phi = b \cdot (\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2) = b \cdot 0 = 0 \tag{2.6}$$

$$u_\psi = b \cdot (\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) = b \cdot 0 = 0 \tag{2.7}$$

Since all rotational movements null itself, there is only an upward thrust as aerodynamic effect. Depending on the upward force F and gravitational Force G the thrust produces the already mentioned movements:

- $F < G$: Fall
- $F = G$: Hover
- $F > G$: Rising

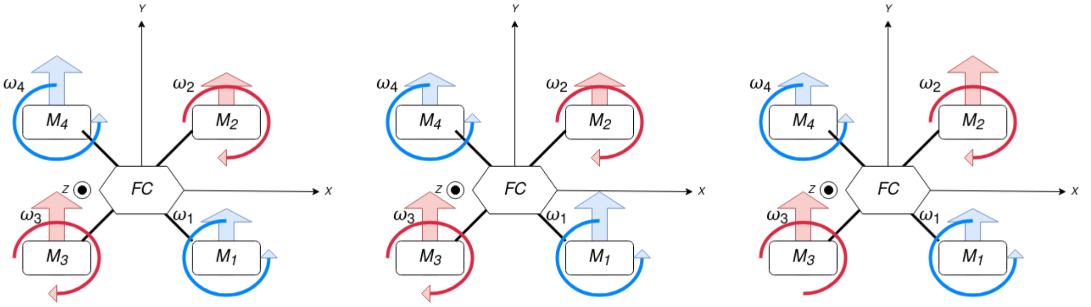


Fig. 2.2: Visualization of Roll, Pitch and Yaw rotational movement (from left to right) and the corresponding difference in the rotational speed ω_i

Roll

In order to *roll* (rotation around y axis) the thrust on one side has to be greater than on the other side. For example, increasing ω_3, ω_4 would lead to uneven distributed thrust. The left side of the aircraft experiences more thrust (Figure 2.2) and as a consequence the aircraft experiences a torque about the y axis. Equation (2.2) to Equation (2.4) also shows this aerodynamic effect:

$$\begin{aligned}
\omega_3^* &= \omega_4^* = n \cdot \omega_1 = n \cdot \omega_2 && n > 1 \\
u_\Theta &= b \cdot ((\omega_1)^2 - (\omega_2)^2 + (\omega_3^*)^2 - (\omega_4^*)^2) = 0 \\
u_\phi &= b \cdot ((\omega_1)^2 + (\omega_2)^2 - (\omega_3^*)^2 - (\omega_4^*)^2) < 0 \\
u_\psi &= b \cdot ((\omega_1)^2 - (\omega_2)^2 - (\omega_3^*)^2 + (\omega_4^*)^2) = 0 \\
u_f &= b \cdot ((\omega_1)^2 + (\omega_2)^2 + (\omega_3^*)^2 + (\omega_4^*)^2) > 0
\end{aligned}$$

Analogue a roll to the other side can be accomplished by increasing ω_1, ω_2, then $u_\phi > 0$. Since the pairs (ω_1, ω_2) and (ω_3, ω_4) each have one propeller that spins clockwise and one propeller that spins counterclockwise, the total torque of the quadrocopter does not change.

Equation (2.1) shows that there still is an upward thrust, but due to the the rotation of the aircraft the thrust force F has a component in x direction. This results in a translatational movement along the x axis.

Pitch

A *pitch* movement around x axis works similar to a roll movement, just with other motor pairs. In this case, the thrust at the front or at the back has to increase. For example, increasing ω_3, ω_1 would lead to a bigger thrust at the back and the corresponding torque (Figure 2.2). Equation (2.2) to Equation (2.4) also shows this aerodynamic effect:

$$\begin{aligned}
\omega_3^* &= \omega_1^* = n \cdot \omega_4 = n \cdot \omega_2 && n > 1 \\
u_\Theta &= b \cdot ((\omega_1^*)^2 - (\omega_2)^2 + (\omega_3^*)^2 - (\omega_4)^2) > 0 \\
u_\phi &= b \cdot ((\omega_1^*)^2 + (\omega_2)^2 - (\omega_3^*)^2 - (\omega_4)^2) = 0 \\
u_\psi &= b \cdot ((\omega_1^*)^2 - (\omega_2)^2 - (\omega_3^*)^2 + (\omega_4)^2) = 0 \\
u_f &= b \cdot ((\omega_1^*)^2 + (\omega_2)^2 + (\omega_3^*)^2 + (\omega_4)^2) > 0
\end{aligned}$$

Analogue a roll to the other side can be a done by increasing ω_2, ω_4, then $u_\Theta < 0$. The total torque also stays the same and the thrust force F gets a component in y direction. This results in a translatational movement along the y axis.

Yaw

A *yaw* movement around z axis is not implemented by using a difference in thrust like in the other rotational movements. Furthermore, differences in torque are used to accomplish yaw rotations (Figure 2.2).
For example, if ω_1, ω_4 increase, the quadrocopter rotates clockwise, because the sum of all torques has to stay the same. The external yaw rotation compensates the difference of the sum of the torques of the propeller. Besides, Equation (2.2) to Equation (2.4) shows this aerodynamic effect:

$$\begin{aligned}
\omega_4^* &= \omega_1^* = n \cdot \omega_3 = n \cdot \omega_2 && n > 1 \\
u_\Theta &= b \cdot ((\omega_1^*)^2 - (\omega_2)^2 + (\omega_3)^2 - (\omega_4^*)^2) = 0 \\
u_\phi &= b \cdot ((\omega_1^*)^2 + (\omega_2)^2 - (\omega_3)^2 - (\omega_4^*)^2) = 0 \\
u_\psi &= b \cdot ((\omega_1^*)^2 - (\omega_2)^2 - (\omega_3)^2 + (\omega_4^*)^2) > 0 \\
u_f &= b \cdot ((\omega_1^*)^2 + (\omega_2)^2 + (\omega_3)^2 + (\omega_4^*)^2) > 0
\end{aligned}$$

A counter clockwise yaw rotation can be executed by increasing ω_2, ω_3, then $u_\psi > 0$. The thrust force F has only a z component. A yaw rotational movement does not result in a translatational movement.

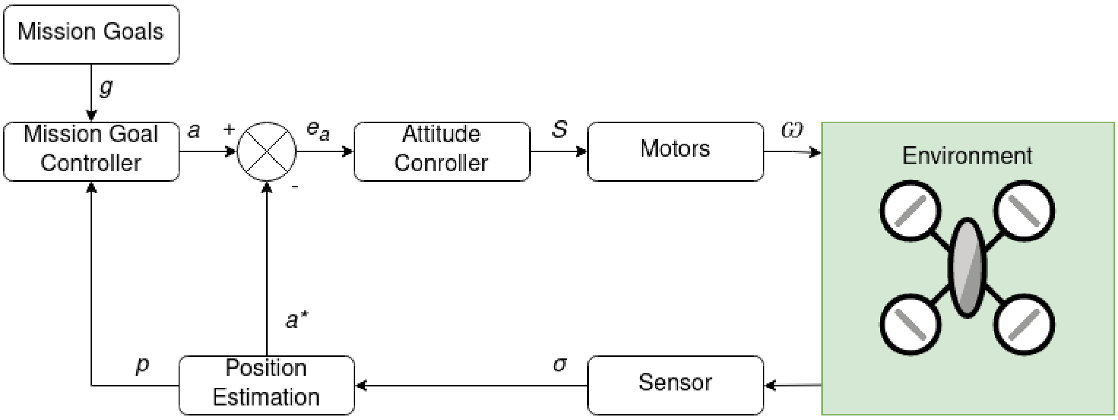


Fig. 2.3: Sketch of an autonomous quadrocopter flight control system

2.1.2 Autonomous Quadrocopter

Autonomus quadrocopter are especially interesting research topics, because it helps automating a lot of purposes. The general task can be in general defined as calculating the optimal motor signals $S = \{s_1, s_2, s_3, s_4\}$ based on the mission goals g and the current pose $p = \{p^*_{3D}, a^*\}$ in order to achieve a stable flight and satisfy the goals. Therefore, it consists of on *inner* and an *outer control loop.*

The inner loop controls the attitude based on the current desired attitude a and the estimated attitude a^*. By subtracting these the attitude error e_a can used to calculate the signals S, which are most commonly pwm signals. The signals are then applied by the motors, which act as actuators and apply a rotational force to the propellers. As a consequence the propeller get the rotational speeds $\omega = \{\omega_1, \omega_2, \omega_3, \omega_4\}$ which directly influence the position of the drone in the environment. The state of the drone is captured by the sensor and the corresponding sensor values σ, which are falsified by noise and offsets. They are then used to estimate the attitude.
The outer loop controls mission goals based on the prior defined goals g and the current estimated position p. These inputs are used in order to calculate the desired attitude a.

In general Control loops in software are always time discreet. For cascaded controllers like here the inner loop is running at a higher frequency than the outer loop. This is crucial, because attitude control is very time sensitive and has to be implemented in real time. A bad or slow attitude control could lead to an unstable flight or in worst case it could crash and therefore automatically erase the possibility to achieve mission goals.

Attitude Controller & Mixing

Common attitude controller often use *PID controller* as one of the most essential parts of nearly all commercial quadrocopter flight control. PID controller are linear feedback controller with a proportional part, an integral part and a differential part and can mathematically described in two different ways:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\ \ d\tau + K_d \cdot \frac{de(t)}{dt} \tag{2.8}$$

$$u(t)^* = K \cdot (e(t) + \frac{1}{T_n} \cdot \int_0^t e(\tau)\ \ d\tau + T_v \cdot \frac{de(t)}{dt}) \tag{2.9}$$

However, since both are mathematically the same just with other gain definitions Equation (2.8) will be the relevant one hereafter. The control signal $u(t)$ is calculated with the use of the configurable constant gains K_p, K_i, K_d, which weigh the different parts of the controller.
Within a PID controller the proportional term considers the current error $e(t)$, the integral term considers the history of errors by integrating about them and the differential term estimates the future error by considering change of the error. Since the attitude control is time discrete in software, Equation (2.8) must be transformed with the use of the sampling time period $\tilde{T}$:

$$u(t) = K_p \cdot e(t) + K_i \cdot \tilde{T} \cdot \sum_0^t e(\tau) + K_d \cdot \frac{e(t) - e(t-1)}{\tilde{T}} \tag{2.10}$$

In a quadrocopter there is a PID controller for each of the three axis(roll, pitch, yaw), which are all working parallel during an inner loop control cycle. Then mixing is used to translate the PID values of each axis to a pwm signal s_i for each motor. This process uses a table, which consists of constants that describe the geometry of the frame. The throttle coefficient f' and the mixer values of the motor M_i $(m_{i,\phi}, m_{i,\Theta}, m_{i,\psi})$.

$$s_i = f' \cdot (m_{i,\phi} \cdot u_\phi + m_{i,\Theta} \cdot u_\Theta + m_{i,\psi} \cdot u_\psi) \tag{2.11}$$

This implementation is modern state of the art attitude control. It is easy to implement, because Equation (2.10) and Equation (2.11) are simple equations and Equation (2.11) is just in need of some constants. In addition, there are not many calculations, so the process can be implemented in realtime. Also, this classic approach shows close to ideal performance in stable environments.
However, [KMWB19] shows that with RL an intelligent agent can be trained with the use of PPO (Section 2.2.5) that outperforms a classic PID Attitude controller in harsh, unpredictable environments.

Position Estimation

Position or state estimation can be accomplished in different ways. By the use of classic filters the sensor values are more accurately, although there still is a relevant error. *Kalman Filters* are the common state of the art solution for position estimation, because they are statistical optimal. Although the name suggests otherwise a Kalman Filter is not a classic filter, but a stochastic weighted combination of the state prediction and measurement.
At each step, the next position and the next measurement is predicted. Based on the prediction of the measurement and the real measurement an innovation is calculated and used in combination with the Kalman Filter Gain K to calculate the next state. At the same time, the state has a covariance and the covariance of the next state is calculated. Then the covariance of the innovation and the Kalman Filter Gain K is calculated and used to correct the covariance of the state.
The amount of calculations increases exponentially with the amount of used sensor values. As a consequence, the use of sensors for the Kalman Filter is limited in realtime tasks. In addition it is in need of linearization. As a consequence there is research to further enhance the filter. For example, [SZWJ22] shows Deep Kalman Filters, which combine the learning ability of deep learning method and the noise filtering ability of the classic Kalman Filter can further enhance trajectory estimation. Although it should be mentioned that the mentioned work uses external satellite images of moving targets. So, the use case is quite different, but in theory adaptable.

Pathfollowing Controller

Pathfollowing Controller work in the outer control loop as Mission Goal Controller (Figure 2.3) with the goal g typically defined as a set of waypoints $\Lambda = \{\lambda_i \quad i \in [0...n]\}$. These waypoints can be defined in different ways:

- $\lambda_i = \{x, y, z\}$: classic point in 3D space
- $\lambda_i = \{x, y, z, \Theta, \phi, \psi\}$: point in 3D space with attitude
- $\lambda_i = \{x, y, z, t\}$: classic point in 3D space with time when it should be reached
  → would lead to a Trajectoryfollowing Controller
- $\lambda_i = \{x, y, z, \Theta, \phi, \psi, t\}$: point in 3D space with attitude and time
  → would lead to a Trajectoryfollowing Controller

At the moment there is a wide range of different approaches for this task. Typically, either straight line paths or circular orbit paths are implemented, depending on the use case.
For example, [NBMB07] implemented a *Vector Field Pathfollowing* with promising results, although [SSS13] shows that this accuracy is combined with large control effort. But there is also the possibility to use a PID like implementation or a classic carrot chasing algorithm.

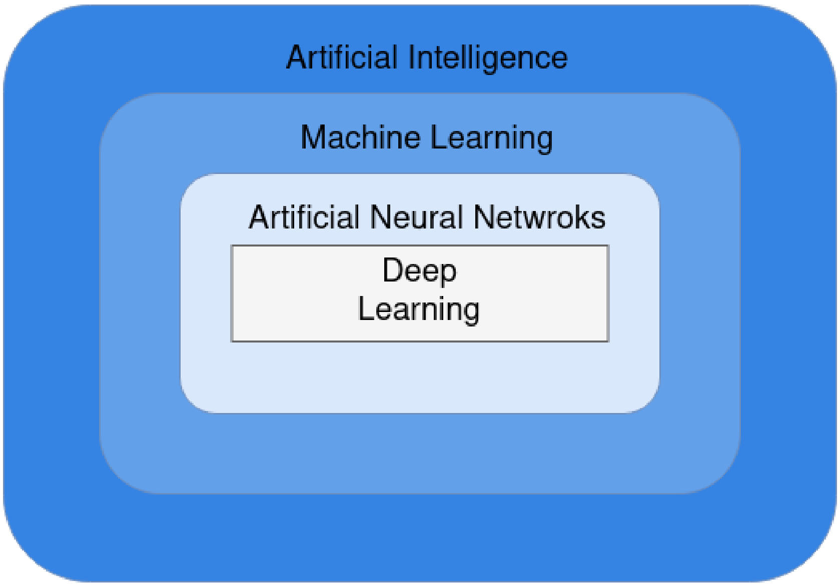


Fig. 2.4: Concept onion of Artificial Intelligence [DJS20]

2.2 Reinforcement Learning Basics

In modern society the field of artificial intelligence is increasingly recognized and used. Be it transformer technologies such as ChatGPT or classic deep learning, artificial intelligence has applications in a variety of different fields of science and everyday life.
Therefore, before explaining RL the different terms and aspects of the different methods should be distinguished.
Artificial Intelligence refers to an area in computer science, which deals with the simulation and application of behaviour that humans would define as intelligent. Although it is hard to define intelligence, there are some easy recognizable symptoms such as solving problems independently or human like interactions. The *Turing Test* [Tur50] represents an operative definition of intelligence. To sum it up, an algorithm passes the test, if a human, who previously asked written questions, can not determine whether the answer is given by the algorithm or a human. In order to pass this test, the algorithm is in need of different things:

- *Natural Language Processing (NLP)*: processing, interpretation, generation and output of natural language
- *Knowledge Representation*: understanding the content of the question
- *Logical Close*
- *Machine Learning (ML)*: adapt to new context, recognize structures

With these definitions and Figure 2.4 it is easy to locate RL, which is an area of machine learning and often uses artificial neural networks in order to satisfy the designated task.

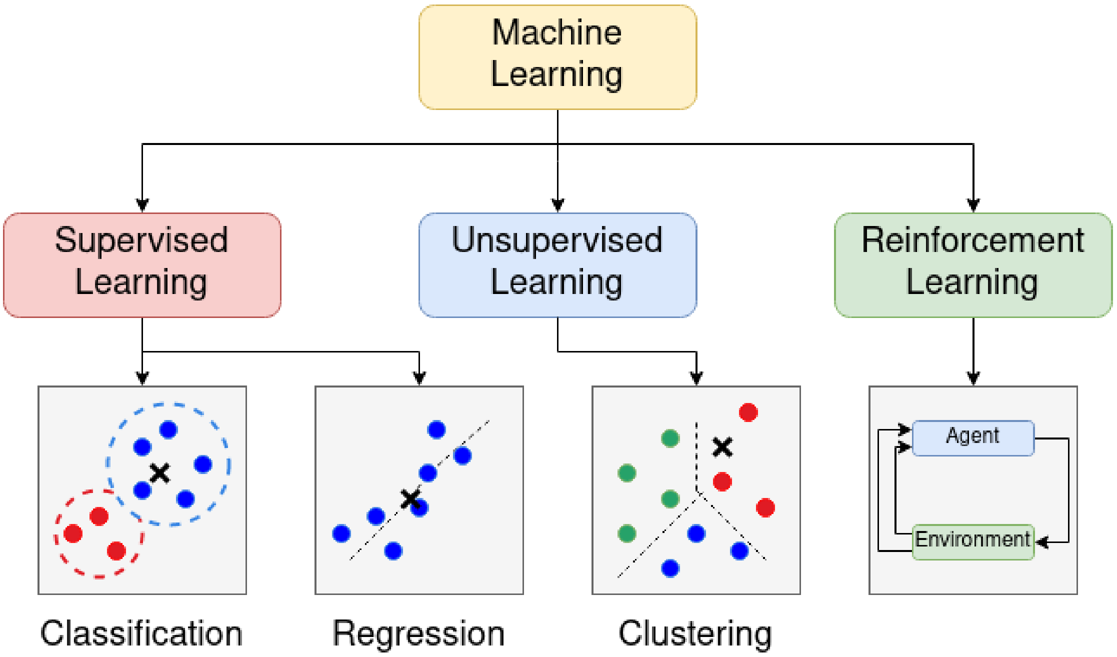


Fig. 2.5: The main learning strategies of machine learning including the related approaches like classification and regression under the Supervised Learning, clustering under the Unsupervised Learning and a sketch of the Reinforcement Learning concept

2.2.1 Classification of Machine Learning Areas

In general Machine Learning is seen as a subfield of artificial intelligence (Figure 2.4), that deals with generating knowledge based on experience. ML programs learn patterns based on training data and then can perform tasks without being explicitly programmed to do so by generalizing unknown examples.

Supervised Learning

Supervised Learning builds a model based on labelled data, which contains the input and the matching output. By iterative optimization supervised learning algorithms learn a function that can be used to predict the output associated with inputs. If learned correctly it also presents the output for inputs that were not a part of the training data set. In this case, the outputs should be analogue to the outputs of similar training data.
The two most common types of Supervised Learning algorithms are *Classification* and *Regression* (Figure 2.5).
Classification algorithms are in need of outputs that are restricted to a limited set of categories. After training the model can determine which category belongs to a certain input. Regression algorithms learn a model to estimate the relationship between a dependant and one ore more independent variable and are used when the outputs may have a numerical value within a defined range.

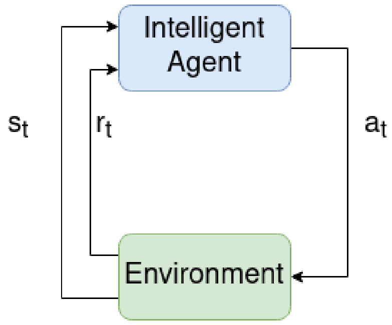


Fig. 2.6: The basic concept of RL: the agent takes an action based on the observed state and receives a numerical reward at each time step

Unsupervised Learning

Unsupervised Learning builds a model with unlabelled data, which only contains the input data. In contrary to Supervised Learning the data set does not contain a preferred output for the given inputs. Unsupervised Learning aims at discovering a distribution of the data in order to gain new knowledge.
Clustering (Figure 2.5) discovers clusters within the given data and can therefore predict for new inputs the related cluster.

Reinforcement Learning

Reinforcement Learning is an area of ML designated to modelling decision making by finding a way to take an action in en environment in order to maximize the sum of rewards. It has proven successful in a wide range of different problems like game theory, control theory or swarm intelligence. The main difference of RL to the already mentioned areas of ML is that the action affects the environment and therefore the observation it receives. Also, it learns by the use of a reward signal.
The *intelligent agent* is the decision maker and typically implemented with the use of a *Neural Network* (Section 2.2.4). Figure 2.6 shows the basic idea: at each time step t the agent takes an action a_t based on the prior observed state s_{t-1}. Based on how good the action was he receives a reward r_{t+1} and the new state s_{t+1}. The reward signal can either depend on the state (s_t), the state and the chosen action (s_t, a_t) or the state, the chosen action and the transitioned state (s_t, a_t, s_{t+1}). It is always numerical and used to optimize the agent. The environment ϵ receives at each time step the action a_t. Then, it simulates the state transition based on its implementation. It calculates the reward signal r_{t+1} and returns it alongside with the observed transitioned state s_{t+1}. It should be mentioned that the observed state s_t must not be the complete intern state of the environment s_t^*. Furthermore, it is only an observation and also can be defined as o_t. In general, the environment can be modelled with the use of a *Markov Decision Process*.

2.2.2 Markov Decision Process

A *Markov Decision Process* (MDP) is a time discreet stochastic control process, which constitutes a mathematical framework for modelling decision making. A MDP is formally defined as a 5-Tupel (S, A, R, P, p_0), where:

- S is the state space with the states $s_i \in S$
- A is the action space with the actions $a_i \in A$
- R is the reward function, mapping $S \times A \times S \rightarrow \mathbb{R}$
with:
$r_{t+1} = R(s_t)$, $r_{t+1} = R(s_t, a_t)$ or $r_{t+1} = R(s_t, a_t, s_{t+1})$
- P is the state transition probability function mapping $T(s_t, a_t, a_{t+1}) \sim Pr(s_{t+1}|s_t, a_t)$
- p_0 is the starting state distribution

The state space S is a set of states s and may be discreet or continuous. Analogue, the action space A may be discreet or continuous. If there are final states the MDP is called *finite MDP*, else *continuing MDP*.
The reward function always returns a numerical reward. Like previously mentioned the reward may only depend on the state s_t or state and action a_t, or state, action and transitioned state s_{t+1}.
When transitioning from one state to the next, the transitioning must not be deterministic. P maps a probability to a transition $T(s_t, a_t, a_{t+1})$ from state s_t to state s_{t+1} by choosing the action a_t. But there also may be deterministic transitions. Then there only is one possible state to transition to and taking the related action will result in this transition with a probability of 1. In addition, the transitions always support the Markov property, that defines that transitions only depend on the most recent state and action. So the prior states $s_0 ... s_{t-1}$ are irrelevant for the current transition.
In addition, the starting state distribution defines which states may be the first state s_0.

By modelling a RL problem as a MDP, there is a mathematical basis to optimize the decision making. Further, it defines the possible states and actions. Choosing a good reward function is the key to influence the decision making. For example, by penalizing the transition to a bad state the agent will learn to avoid to choose the matching action. Analogue, a big positive reward can influence the decision making.

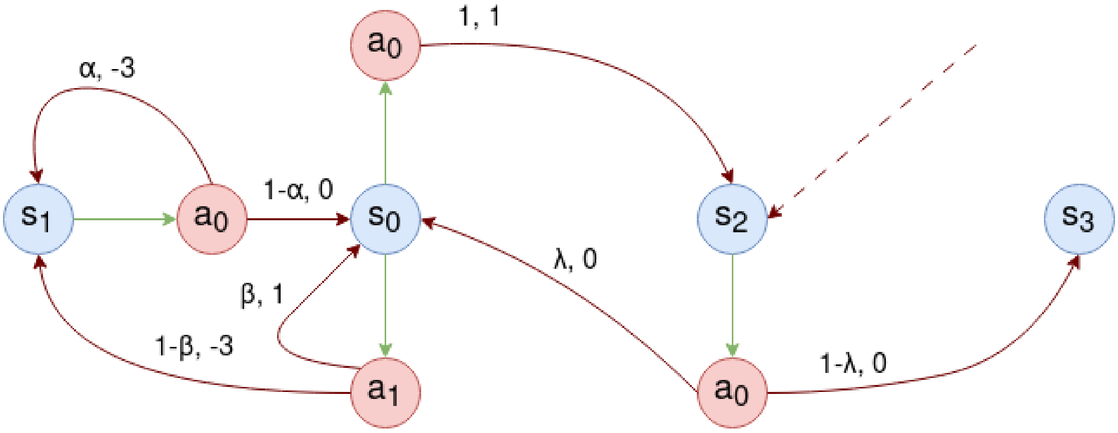


Fig. 2.7: Visualization of a simple MDP with 4 states (s_0, s_1, s_2, s_3) (blue nodes), 2 actions (a_0, a_1), state-action edges (green) and action-state edges (dark red) containing a tuple (possibility, reward)

Visualization

A Markov Decision Process can be visualized with the help of a directed transition graph. The graph posses state nodes and action nodes. Each state node is connected with $n \in \mathbb{N}_0$ actions with directed state-action edges. Each action node is connected with $k \in \mathbb{N}$ states with action-state edges, that posses a probability α_i. Since after taking in action there must be a transition, it follows that the sum of all the outgoing action-state edge probabilities must be equal to 1:

$$\sum_{i=0}^{k} \alpha_i = 1 \tag{2.12}$$

It is also possible that there is an state-action edge from state node s_k to action node a_n and an action-state edge from a_n back to s_k. It is equivalent to taking an action that may cause no transition at all and the state stays the same. In addition the graph may poses starting and ending states.
For example, Figure 2.7 shows a finite MDP with 4 states (blue nodes). S_2 is defined as starting state and since no action can be taken in state s_3, this is a ending state. If a state possesses multiple state-action edges like s_0 a decision has to be made. After choosing an action there may be multiple action state edges like action a_1 from state s_0. In this example there is a state transition to s_1 with a probability of $1 - \beta$ that succeeds in a numerical reward of -3. Also, there is the possibility of β to transition into s_0 which causes a reward of 1.

2.2.3 V,Q Function & The Bellman Equations

In order to further define how the agent learns, there are a couple of functions, equations and helpful definitions. The *policy* π defines which action should be taken based on the current state and the *trajectory* τ defines different possible sequence of transitions inside the MDP. In order to find the optimal trajectory there are different metrics defining an expected value to each state (*V Function* Section 2.2.3) and each state-action pair (*Q Function* Section 2.2.3). With these functions different policies can be evaluated.

Policy

The policy π is a mapping from the state space S to the action space A. Formally this mapping is described as

$$S \rightarrow \mathbb{P}(A, \pi(a_t, s_t))$$

and defines the probability of choosing the action a_t in state s_t.
In many RL Algorithms *Neural Networks* (Section 2.2.4) will be used to model the policy with the goal to approximate an optimal policy π^* that maximizes the expected reward:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

Trajectory

A sequence of states $s_0 ... s_{T+1}$ and the matching action $a_0 ... a_T$ is called a trajectory π with the amount of steps T. For example there could be the trajectory $\tau_1 = (s_2, a_0, s_0, a_1, s_1)$ for the visualized MDP (Figure 2.7) with $T = 2$ steps: Since s_2 is defined as a starting state it always is the first state of the trajectory. By choosing a_0 there is a state transition to state s_0. Then there is a transition to state s_1 by choosing the action a_1. This trajectory τ_1 could be achieved with the policy π_1 with $\pi_1(a_0|s_2) = 1, \pi_1(a_1|s_0) = 1$. This policy literally defines that in the state s_2 always the action a_0 should be taken and in state s_0 always the action a_1. Further there can be described a probability that this trajectory τ occurs under the policy π.

$$P(\tau|\pi) = p_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1} s_t, a_t|) \pi(a_t|s_t) \tag{2.13}$$

This is basically the the product of the probabilities that the policy π takes an action and that the action a_t transitions the state to s_{t+1}. In the above example, this would result in

$$P(\tau_1|\pi_1) = 1 \cdot \lambda \cdot 1 \cdot (1 - \beta) = \lambda \cdot (1 - \beta)$$

Tab. 2.1: Example of all two step trajectories with the matching probability, reward and expected reward of the MDP shown in Figure 2.7

τ	$P(\tau\|\pi)$	$R(\tau)$	$J(\tau)$
(s_2, a_0, s_3)	$(1-\lambda)$	0	0
$(s_2, a_0, s_0, a_1, s_0)$	$\lambda \cdot \beta$	1	$\lambda \cdot \beta$
$(s_2, a_0, s_0, a_1, s_1)$	$\lambda \cdot (1-\beta)$	-3	$-3 \cdot \lambda \cdot (1-\beta)$

Discounted & Undiscounted Reward

In *finite*, episodic MDPs with ending states, the sum of the rewards is easy to calculate for a trajectory τ with the use of the amount of steps T.

$$R(\tau) = \sum_{t=0}^{T-1} r_{t+1} \tag{2.14}$$

It was already discussed in Section 2.2.2 that a MDP may be infinite and continuing. As consequence the undiscounted reward, that was defined in the above may diverge, because there is no ending state. In order to counter the diverging of the sum a discount factor $\gamma \in [0...1]$ is used. The idea behind is to weigh future rewards. With increasing future step t the weigh factor decreases exponentially. As a consequence the sum of the rewards converges and is called discounted reward.

$$\begin{aligned} R(\tau)_t &= \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} && (2.15) \\ &= r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + ... && (2.16) \end{aligned}$$

Equation (2.15) shows how the discounted reward is calculated:
Future rewards are weighed with the factor $\gamma^n \quad n \in \mathbb{N}_0$ depending on how many steps they are in the future. The special case of $\gamma = 1$ means that the task should be episodic, the case of $\gamma = 0$ results in a greedy implementation, where future rewards are not considered at all and the agent learns to always choose the action that implies the highest reward in that step. With the before mentioned possibility of a transition τ under the policy π an expected reward $J(\pi)$ of a policy can be defined. For example in Figure 2.7 there can be a couple of $T = 2$ step trajectories τ_i with a matching probability and reward $R(\tau_i)$ (Table 2.1) that leads to an expected reward $J(\pi_1)$ of depending on the transition probabilities λ, β, α:

$$J(\pi_1) = \lambda \cdot \beta - 3 \cdot \lambda \cdot (1-\beta)$$

V-Function

A vast majority of *Reinforcement Learning Algorithms* try to estimate a *Value Function* in order measure how good it is to be in a state s_t under a matching policy π. The V Function calculates the expected return $J(\pi)$ with the starting state s_t and acting accordingly to the defined policy:

$$V^\pi(s) = \mathbb{E}_{\tau \sim P(\cdot|\pi)}[R(\tau)|s_0 = s] \tag{2.17}$$

$$= \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1|s=s_t}] \tag{2.18}$$

In addition, an *optimal Value Function* (Equation (2.19)) can be defined and used in order to calculate an optimal policy:

$$V^*(s) = \max_\pi V^\pi(s) = \max_\pi \mathbb{E}_{\tau \sim P(\cdot|\pi)}[R(\tau)|s_0 = s] \tag{2.19}$$

Q-Function

The *Q Function* (action-value function) is defined quite similar to the Value Function but instead of starting in a state, the Q Function starts in a state-action pair and then takes actions according to the chosen policy. Therefore, it is also mathematically defined as the expected reward:

$$Q^\pi(s,a) = \mathbb{E}_{\tau \sim P(\cdot|\pi)}[R(\tau)|s_0 = s, a_0 = a] \tag{2.20}$$

Analogue to the V Function, an *optimal Q Function* can be calculated by taking the maximal expected reward.

$$Q^*(s,a) = \max_\pi Q^\pi(s) = \max_\pi \mathbb{E}_{\tau \sim P(\cdot|\pi)}[R(\tau)|s_0 = s, a_0 = a] \tag{2.21}$$

With the help of the optimal Q Function the optimal action a for every state can be chosen by comparing their optimal Q values and used to construct an optimal policy π^*. If the MDP has discreet actions and states this is no problem and then called *Tabular Reinforcement Learning.* For more complex MDPs RL algorithms that use NN can be used. Probably the best known algorithm of this kind is *Deep Q Learning* [FWXY20] which estimates the Q Function with the use of a behaviour and a target policy. These policies are implemented with the use of Neural Networks.

The Bellman Equations

The *Bellman equations* are the central point of some RL algorithms. It parts the V and Q function into two separate parts consisting of the instant reward and the discounted future values. As an effect it simplifies the computation of the value function. Instead of building a sum over multiple time steps, an optimal solution of a complex problem is found by finding optimal solutions for simpler, recursive subproblems.

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(\cdot|S), s' \sim P(\cdot|S,a)}[R(s,a,s') + \gamma \cdot V^{\pi}(s')] \tag{2.22}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P(\cdot|S,a)}[R(s,a,s') + \gamma \cdot \mathbb{E}_{a' \sim \pi(\cdot|s')}[Q^{\pi}(s',a')]] \tag{2.23}$$

Equation (2.22) shows the instant reward $R(s,a,s')$ which is computed by calculating the expected value for all possible actions a and all possible transition states s' and the discounted future $\gamma \cdot V^{\pi}(s')$ reward of the values of the possible transition states. Analogue, Equation (2.23) shows the split up of instant reward and future reward for state-action pairs. As a consequence, iterative approaches can be implemented, that possesses the ability to calculate the values of all states or the values of all state-action pairs.

The Bellman Equations of Optimality

It was already discussed how optimal V and Q functions can be calculated and used to determine the optimal actions and create an optimal policy(Equation (2.19), Equation (2.21)). Analogue the *Bellman Equations of Optimality* can be defined with the use of Equation (2.22) and Equation (2.23):

$$V^{*}(s) = \max_{a} \mathbb{E}_{s' \sim P(\cdot|s,a)}[R(s,a,s') + \gamma \cdot V^{*}(s')] \tag{2.24}$$

$$Q^{*}(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}[R(s,a,s') + \gamma \cdot \max_{a'} Q^{*}(s',a')] \tag{2.25}$$

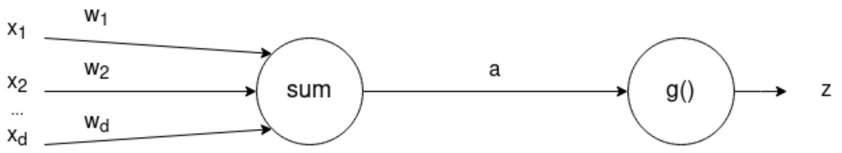


Fig. 2.8: The McCulloch-Pitts model of a single neuron uses a weighted sum with the inputs x_i and weights w_i and a non-linear activation function $g()$ in order to compute the output z [Bis94]

2.2.4 Neural Networks

A *Neural Network (NN)* can be described as an reactive computing system that consists of multiple simple *Neurons* that are interconnected. Therefore, the output does depend on the inputs and the defined dynamic state response of each neuron. NN are inspired by the network of neurons in the human brain which pass information to another with the help of synapses. While biological brains are dynamic and analog, NN tend to be static and symbolic.

NN have shown to be very useful, because they are universal function approximators. In addition, they are adaptive and poses the ability to change their structure based on external or internal information that flows threw it. They are non-linear and can be used to model complex relationships between inputs and outputs or find specific pattern in data.

Neuron

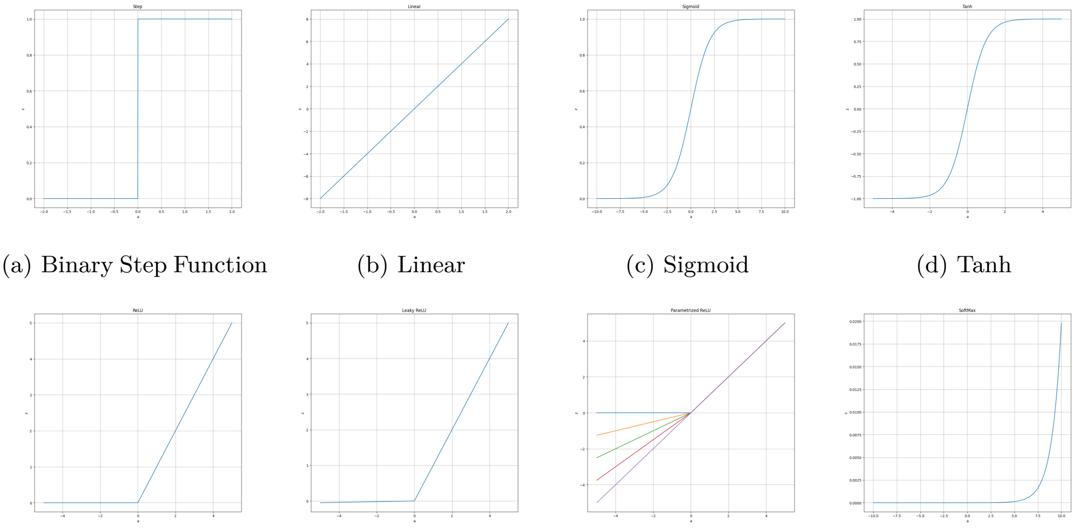
A Neuron is the basic element of a NN. Mathematically a neuron can be seen as a non linear function, which computes the output z based on a set of inputs $x_i, i \in [1, ..., d]$ [Bis94]. Further, [MP43] introduced the simple mathematical framework, called the McCulloch-Pits model (Figure 2.8):

$$a = \sum_{i=1}^{d} w_i \cdot x_i + w_0 \tag{2.26}$$

$$a = \sum_{i=0}^{d} w_i \cdot x_i \tag{2.27}$$

$$z = g(a) \tag{2.28}$$

Each input x_i is multiplied with its matching weight w_i, which is a parallel to synaptic strength in biological networks like brains (Equation (2.26)). In addition, there is a offset which is called *bias* and is analogue to the firing threshold of a biological neuron. By using the bias as additional input and setting it to 1, the sum can be further simplified (Equation (2.27)). The output of the neuron is then computed by giving it to a non-linear *Activation Function* $g()$.



(a) Binary Step Function (b) Linear (c) Sigmoid (d) Tanh

(e) ReLU (f) Leaky ReLU (g) Parametrized ReLU (h) SoftMax

Fig. 2.9: 8 different Activation Functions.

Activation Function

It was already discussed that NN are universal function approximators and that the neuron consists of a linear sum (Equation (2.27)). In order to represent nonlinear convoluted functional mappings between input and output, non linearity is added to each neuron with a *Activation Function*. It is important that these functions are differential in order to implement a back propagation optimization strategy [SSA17].
At the moment, there are different activation functions commonly used (Figure 2.9):

- Binary Step Function
- Linear
- Sigmoid
- Tanh
- ReLU
- Leaky ReLU
- Parametrized ReLU
- SoftMax

Each of this different functions has its advantages. For example, SoftMax is good at multiclass classification problems, while Sigmoid is good at binary classification problems. Also, in a NN with ReLU not all neurons are activated at the same time, which increases efficiency.

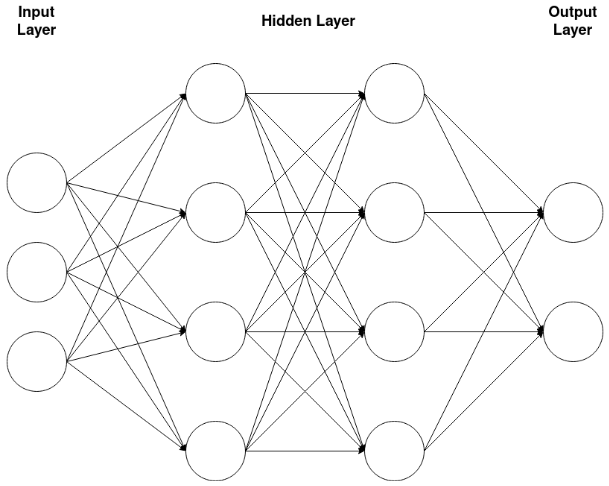


Fig. 2.10: Simple Feed-Forward Network with 3 neurons in the input layer, 2 hidden layers with 4 neurons and with 2 layers in the output layer

Simple Feed-Forward Network

A *Feed-Forward Network* is an artificial network wherein information directly flows forward in one direction from the input to the output. In contrast to *Recurrent Neural Networks*, it does not contain cycles of nodes and information can not flow in loops. It can be described as a directed graph $G = (V, E, w)$ with neurons as the set of nodes V and the set of edges E with the matching weight w (Figure 2.10). The neurons are grouped into different Layers:

- Input Layer: each node receives its input x and calculates the output z with the use of Equation (2.27) and passes it to each node of the next layer
- Hidden Layer: each node receives its inputs $x_i, i \in [1...d]$ from the d nodes of the previous layer and uses Equation (2.27) and the weights $w_i, i \in [1..d]$ in order to calculate the output z and pass it to each node of the next layer
- Output Layer: each node receives its inputs $x_i, i \in [1...d]$ from the d nodes of the previous layer and uses Equation (2.27) and the weights $w_i, i \in [1..d]$ in order to calculate the output z of the NN

The number of hidden layers can differ, likewise the amount of neurons in each layer. Due to the structure of neurons even simple Feed-Forward Networks can approximate functions $F : \mathbb{R}^j \rightarrow \mathbb{R}^k$, where the dimension j equals the amount of nodes in the input layer and k equals the amount of nodes in the output layer.

Deep Neural Network

Deep Neural Networks differ from the previous mentioned NN in terms of complexity. Due to the large amount of neurons and a lot of hidden layers, the NN is able to approximate very complex mappings from the input to the output space.

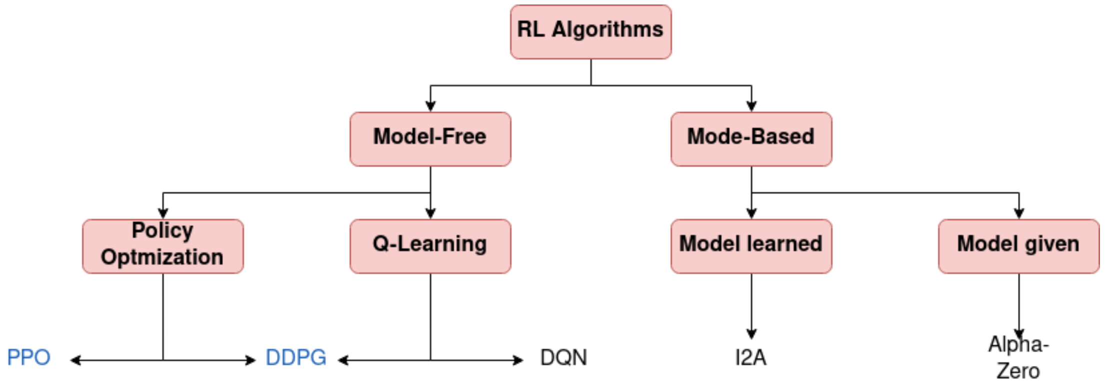


Fig. 2.11: Overview of the different categories of RL algorithms with examples: PPO and DDPG Algorithm coloured because they were used in this work.

2.2.5 State of the Art Algorithms

Modern state of the art RL algorithms can be divided into different categories (Figure 2.11):
Model-based Algorithms uses its experience in order to construct a model of the environment by modelling the MDP transitions. With the use of this model the optimal actions are chosen for the policy. This category can be further devided into *model learning algorithms* like I2A and *model given algorithms* like Alpha-Zero. Model learning algorithms observe the trajectory τ while following a policy π and use it in order to learn a dynamics model. Then the algorithms plan through the dynamics model in order to choose the actions. Model given algorithms work quite similar but the dynamic model is already given.
In *Model-free Algorithms* the agent constructs a policy based on trial-and-error experience and can be further divided into *Policy Optimization* and *Q-Learning.* Policy Optimization algorithms like *PPO* the policy is learned directly. In Contrast Q-Learning algorithms like *Deep Q-Learning* learns the Q-function and updates it in order to get an optimal Q-function. In addition there are hybrid algorithms like *DDPG* that combines learning the policy function and Q-function.
RL algorithms can be further categorized into *on-policy* and *off-policy* algorithms. On-policy algorithms use the current optimized policy in order to gain experience that is then used to further optimize the policy. In contrast, off-policy algorithms learn from actions that may not be according to the current policy. This experience is then used to construct the optimal policy.
The algorithms used in this work are part of *Stable Baselines 3*[RHG+21], which is a set of reliable implementations of RL algorithms in *PyTorch* that promise efficient learning and a good base to build projects on top of. Each of the main state of the art RL algorithms can be found there and used with custom policies, custom environments and custom callback functions. In addition, it offers *Tensorboard* support, which is a tool that can be used to visualize the training process.

PPO - Basics

PPO algorithms [SWD$^+$17] are a family of policy gradient methods for RL, they directly try to improve the policy π as much as possible in a single step without stepping to far and avoiding performance collapse. It is an *Actor-Critic Method* that approximates the value function V_ϕ and the policy π_Θ with the matching set of parameters ϕ, Θ. In each step the actor updates the policy parameters and the critic updates the value function parameters. In order to measure how good a policy π_Θ performs in relation to an old policy $\pi_{\Theta_{old}}$ a surrogate objective is used, that keeps the new policies close to the old one:

$$L^{PPO}(\Theta) = \mathbb{E}_{\tau \sim P(\cdot|\pi_{\Theta_{old}})}[\sum_{t=0}^{T} L(s_t, a_t, \Theta_{old}, \Theta)] \tag{2.29}$$

$$L(s_t, a_t, \Theta_{old}, \Theta) = \min(\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_k}(a_t|s_t)} A(s_t, a_t), g(\epsilon, A(s_t, a_t))) \tag{2.30}$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon) \cdot A & \text{if } A \geq 0 \\ (1-\epsilon) \cdot A & \text{if } A < 0 \end{cases} \tag{2.31}$$

This surrogate objective mainly depends on the hyperparameter ϵ which is an upper bound to the distance between the policies and the advantage function A, that estimates how good an action is compared to the average action for a specific state.
Equation (2.29) can be explained intuitively by observing a single state-action pair:
If the advantage of the state-action pair is positive Equation (2.30) is reduced to

$$L(s_t, a_t, \Theta_{old}, \Theta) = \min(\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_k}(a_t|s_t)}, (1+\epsilon)) \cdot A(s, a) \tag{2.32}$$

Because of the positive advantage the objective will increase with $\pi_\Theta(a|s)$, but is still limited to $(1+\epsilon) \cdot A(s, a)$ by the minimum expression.
If the advantage of the state action pair is positive Equation (2.30) is reduced to

$$L(s_t, a_t, \Theta_{old}, \Theta) = \max(\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_k}(a_t|s_t)}, (1-\epsilon)) \cdot A(s, a) \tag{2.33}$$

Because of the negative advantage the objective will increase with the decrease of $\pi_\Theta(a|s)$, but is still limited to $(1-\epsilon) \cdot A(s, a)$ by the maximum expression.
As a consequence the policy does not profit from changing a lot.
PPO is an on-policy algorithm, so the actions are selected according to the latest policy. Therefore, the randomness of the action selection decreases over time, which may causes the problem of being trapped in a local optima.

PPO - Algorithm

The PPO algorithm (Algorithm 1)first initializes the policy and value net with the matching parameters Θ_0 and ϕ_0.
Then the following steps are executed until convergence:
By running the current policy π_{Θ_k} a set of Trajectories D_k is collected and the rewards R_t are computed. Then the advantage A is computed based in the current value function approximation V_{ϕ_k}. With the use of this values the surrogate objective (Equation (2.29)) is calculated and used to update the policy π. Typically stochastic gradient ascent is used in order to update the policy. At last the value function is updated by using regression on the mean squared error.

Since the release of PPO it is known to be a state of the art algorithm with a good performance that is at least as good as other Policy Optimization methods on a variety of environments. In addition, the base algorithm is quite easy to implement.
Nevertheless there are still known problems:
The algorithm is quite sensitive to the initialization parameters Θ_0, ϕ_0. Like previously mentioned PPO can get stuck in local optima. This is the case if there are local optimal actions close to initialization.
PPO seems to be unstable when the reward function is not bounded on continuous action spaces.

```
Algorithm 1: Proximal Policy Optimization [SWD+17]
Initialize policy parameters $\Theta_0$ and value function parameters $\phi_0$
$k = 0$
repeat
    Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_{\Theta_k}$
    Compute rewards-to-go $R_t$
    Compute advantage estimates $A$ based on the current value function $V_{\phi_k}$
    Update the policy by maximizing the PPO objective
        $$\Theta_{k+1} = \arg\max_{\Theta}(\frac{1}{|D_k|} \sum_{\tau_i \in D_k} \sum_{t=0}^{T} \min(\frac{\pi_\Theta(a_t|s_t)}{\pi_{\Theta_k}(a_t|s_t)} A(s_t, a_t), g(\epsilon, A(s_t, a_t))))$$
     typically via stochastic gradient ascent
    Fit value function by regression on mean-squared error
        $$\phi_{k+1} = \arg\min_{\phi}(\frac{1}{|D_k|T} \sum_{\tau_i \in D_k} \sum_{t=0}^{T} (V_\phi(s_t) - R_t)^2)$$
    $k+ = 1$
until convergence
```

SAC - Basics

Soft Actor-Critic (SAC) [HZAL18] is an offpolicy Rl algorithm that optimizes a strochastic policy. It can be categorized as a mixed method between policy optimization and value based methods. A key feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy. The entropy of a policy can roughly be defined as a metric of measuring the randomness in a policy. Increasing the entropy results in more exploration, which accelerates training and can prevent the policy from converging to a bad local optimum. As a consequence, the RL problem changes to Equation (2.34) with trade-off coefficient α and entropy H.

$$\pi^* = \arg\max_{\pi}\ \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t \cdot (R(s_t, a_t, s_{t+1} + \alpha H(\pi(\cdot|s_t)))] \tag{2.34}$$

As a consequence the definition of the V function and Q function changes to Equation (2.35) and Equation (2.36).

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t \cdot (R(s_t, a_t, s_{t+1} + \alpha H(\pi(\cdot|s_t)))|s_0 = s] \\ &= \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t \cdot (R(s_t, a_t, s_{t+1} - \alpha \log \pi(a|s)] \\ &= \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a) - \alpha \log \pi(a|s)] \qquad (2.35) \\ Q^{\pi}(s,a) &= \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t \cdot R(s_t, a_t, s_{t+1} + \alpha \sum_{t=1}^{\infty}\gamma^t \cdot H(\pi(\cdot|s_t)))|s_0 = s, a_0 = a] \qquad (2.36) \end{aligned}$$

SAC solves this new RL problem by learning a policy π_Θ and two Q funcions Q_{ϕ_1}, Q_{ϕ_2}. The Q functions are learned by regressing to a single shared target with the loss function seen in Equation (2.37).

$$\begin{aligned} L(\phi_i, \mathcal{D}) &= \mathbb{E}_{(s,a,r,s',d)\sim D}[(y(r, s', d) - Q_{\phi_i}(s,a))^2] \qquad (2.37) \\ y(r, s', d) &= r + \gamma \cdot (1-d) \cdot (\min_{j=1,2} Q_{\phi_{targ_j}}(s', \tilde{a}') - \alpha \log \pi_\Theta(\tilde{a}'|s')) \qquad (2.38) \\ \tilde{a}' &\sim \pi_\Theta(\cdot|s') \end{aligned}$$

There, $\tilde{a}'$ is the next action. Unlike r, s' the next action is sampled from the policy and not from the replay buffer $\mathcal{D}$.

The policy should learn to maximize the value of each state $V^{\pi}(s)$ (Equation (2.35)). Therfore, the *reparameterization trick* is used, in which a sample from $\pi_{\Theta}(\cdot|s)$ is collected with the use of a squashed Gaussian policy. There, samples are collected according to Equation (2.39) and used in the V function (Equation (2.40)). Then, the policy is loss is given by choosing the minimum of the 2 Q function approximators as Q function (Equation (2.41)).

$$\tilde{a_{\Theta}}(s,\xi) = tanh(\mu_{\Theta}(s) + \sigma_{\Theta}(s) \odot \xi) \qquad \xi \in \mathcal{N}(0,I) \tag{2.39}$$

$$V^{\pi}(s) = \mathbb{E}_{\xi \sim \mathcal{N}}[Q^{\pi_{\Theta}}(s,\tilde{a}_{\Theta}(s,\xi)) - \alpha \cdot \log \pi_{\Theta}(\tilde{a}_{\Theta}(s,\xi)|s)] \tag{2.40}$$

$$= \mathbb{E}_{s \sim \mathcal{D}, \xi \sim \mathcal{N}}[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_{\Theta}(s,\xi)) - \alpha \cdot \log \pi_{\Theta}(\tilde{a}_{\Theta}(s,\xi)|s)] \tag{2.41}$$

The explore-exploit tradeoff is controlled with α. A lower α corresponds to more exploitation, a higher α to more exploration. During training it is optimized in order to avoid getting stuck in local optima.

SAC - Algorithm

The SAC algorithm (Algorithm 2) gets the policy parameters Θ, function parameters ϕ_1, ϕ_2 and an empty replay buffer $\mathcal{D}$ as input. First, the target parameters are set equal to the main parameters.
Then the following steps are executed until convergence:
The state s is observed and an action is taken with the use of the policy π_{Θ} and executed. Then, the next state s', reward r and the done signal d are observed and the 5-tupel (s,a,r,s',d) is stored in the replay buffer $\mathcal{D}$. If the done signal d indicated that s' is a terminal state, the environement is reset. If it is time to update an amount of updates are performed.
In the update a batch of transitions $\mathcal{B} = \{(s,a,r,s',d)\}$ is sampled from the replay buffer $\mathcal{D}$ and the targets are computed according to Equation (2.38) and used to update the two Q functions via gradient descent. Then, the policy is updated with the use of gradient ascent with the loss seen in Equation (2.41). At last, the target networks are updated with Polyak averging, where $\rho \in [0,1]$ is a hyperparamter.

Sometimes, implementations use a trick to improve exploration at the start of the training by taking actions sampled from a uniform random distribution over valid actions for a fixed amount of steps. Afterwards, the algorithm works like previously defined.
SAC is known for stable performance.

Algorithm 2: Soft Actor-Critic [HZAL18]

```
Input: initial policy parameters $\Theta$, Q function parameters $\phi_1, \phi_2$, empty replay
       buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{targ,1} \leftarrow \phi_1, \phi_{targ,2} \leftarrow \phi_2$
repeat
    Observe state $s$ and select action $a \sim \pi_\Theta(\cdot|s)$
    Execute $a$ in the environment
    Observe next state $s'$, reward $r$ and done signal $d$ to indicate wether $s'$ is
      terminal
    Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
    if $s'$ is terminal then
        reset environment state
    if it is time to update then
        for j in range (however many updates) do
            Randomly sample a batch of transitions, $\mathcal{B} = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets for the Q functions:
                $$y(r, s', d) = r + \gamma \cdot (1 - d) \cdot (\min_{j=1,2} Q_{\phi_{targ_j}}(s', \tilde{a}') - \alpha \log \pi_\Theta(\tilde{a}'|s'))$$
                $$\tilde{a}' \sim \pi_\Theta(\cdot|s')$$
            Update Q functions by gradient descent:
                $$\nabla_{\phi_i} \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d) \in \mathcal{B}} (y(r, s', d) - Q_{\phi_i}(s, a))^2$$
              for $i = 1, 2$
            Update policy by one step of gradient ascent:
                $$\nabla_\Theta \frac{1}{|\mathcal{B}|} \sum_{s \in \mathcal{B}} (\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\Theta(s, \xi)) - \alpha \cdot \log \pi_\Theta(\tilde{a}_\Theta(s, \xi)|s))$$
              where $\tilde{a_\Theta}(s)$ is a sample from $\pi_\Theta(\cdot|s)$ which is differentiable wrt $\Theta$
            Update target networks with Polyak averging:
                $$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1 - \rho)\phi_i$$
              for $i = 1, 2$
until convergence
```

2.3 Pybullet Physics Simulator & Gym Pybullet Drones

Pybullet is a Python simulator that provides a fast and easy to use module for machine learning and robotics simulation. It provides dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection and ray intersection queries [CB21].
Gym Pybullet Drones [PZZ$^+$21] is an *OpenAI Gym environment* based on the previously mentioned Pybullet physics simulator as back-end. It aims at multi-agent reinforcement learning for quadrocopter simulations. Therefore, it provides a small variety of dronemodels as urdf files, for example based on the *Bitcraze's Crazyflie 2.x nano-quadrocopter* and gym environments.

2.3.1 BaseAviary Class

In Gym Pybullet Drones all aviary classes inherit from the *BaseAviary Class.* It provides a couple of useful methods, the most important ones named hereafter:

- init: initializes environment and loads the drone urdf file
- reset: resets the environment
- step: simulates one step and returns the observation, reward, a done information and an info
- render: renders the environment as textual output
- startVideoRecording: starts the recording of a mp4 video
- physics: the base PyBullet physics implementation
- dynamics: the drone dynamics implementation

2.3.2 BaseSingleAgentAviary Class

The *BaseSingleAgentAviary Class* is the most important environment class for this work, since it provides a framework for single agent RL problems. It inherits from the BaseAviary Class and defines typical action types, observation types and preprocesses the action that is passed to the step method of its mother class.
The implemented environment in this work inherits from this class.

3 Flight-Control Concept

It was already discussed in Section 2.1.2 how the flight control of modern UAV is structured. Since this work aims at using reinforcement learning to learn a robust control of a quadrocopter, the explained does not fit this structure. Figure 2.3 shows the use of a mission goal controller that shows some distant resemblance to the implemented intelligent agent. In Contrast, it does not output a desired attitude to achieve the defined goal, but directly outputs actions that are more or less the motor values. As a consequence there is no need for a classic Attitude Control but a need of controlling the motor signals to match the actions. This chapter proposes a flight control concept with the use of an intelligent agent. Also, it provides a general concept of the implementation in order to get a wider view at the different implemented classes, tools and scripts.

At each timestep t the agent receives a mission goal $\overrightarrow{g} = (g_x, g_y, g_z)$ and a normalization of the position/state estimation $p = (p_x, p_y, p_z, \Theta, \phi, \psi, \dot{p_x}, \dot{p_y}, \dot{p_z}, \dot{\Theta}, \dot{\phi}, \dot{\psi})$ and outputs a 4-tupel of actions $a = (a_0, a_1, a_2, a_3)$ with each $a_i \in [-1, 1], i \in [0, 1, 2, 3]$. This can be denormalized to a signal S_N. On a real UAV this would be most likely a pwm signal. With the use of a speed controller the signal is held at the specific value. With the use of the motors and props there is a thrust $\tilde{f}$ and a corresponding state transition that can be observed by the sensors. These sensor values σ can be used in order to estimate the position and normalize it to the range $[-1, 1]$. This concept is visualized in Figure 3.1.

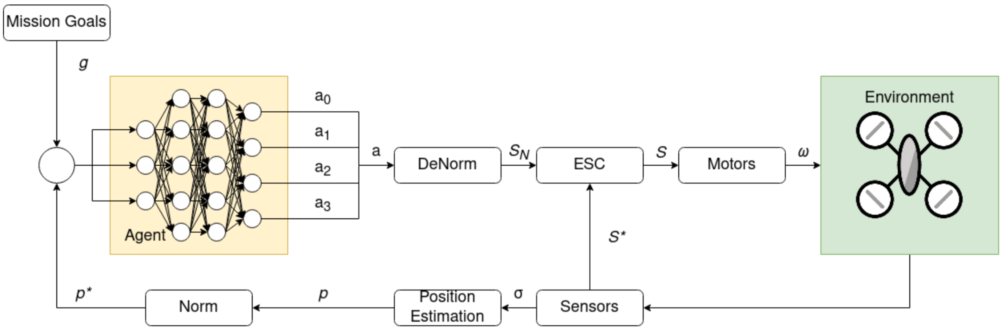


Fig. 3.1: Proposed Concept of autonomous flight control with the use of a intelligent agent implemented with the use of a NN. The output of the NN is denormalized in order to translate them to motor signals. With the use of the sensors and position estimation the input for the next control step are calculated.

4 Implementation

The concept of implementation can be divided into the used packages (*Stable Baselines3, Gym Pybullet Drones, ...*), the scripts, the environment classes and different support classes (Figure 4.1).
Stable Baselines3 is used as implementation of the PPO algorithm (Algorithm 1) and SAC algorithm (Algorithm 2) in order to achieve the tasks. Gym Pybullet Drones has already some environments and is the foundation of the simulation. It already provides a suitable step function and a well designed physics engine.
The scripts (Section 4.2, lila) are used to either learn the agent on a given environment or evaluate it.
The environment classes(Section 4.1, green) inherit from a *Gym Environment* and models a MDP. By modelling this MDP precisely, it is defined what is learned later by the intelligent agent. It uses the wind class in order to model a harsh environment. In addition, there are a couple of evaluation tools (Section 4.2.3, red), that supports the implemented classes.

The whole concept is implemented in Python3.x with the use of a *conda environment*.

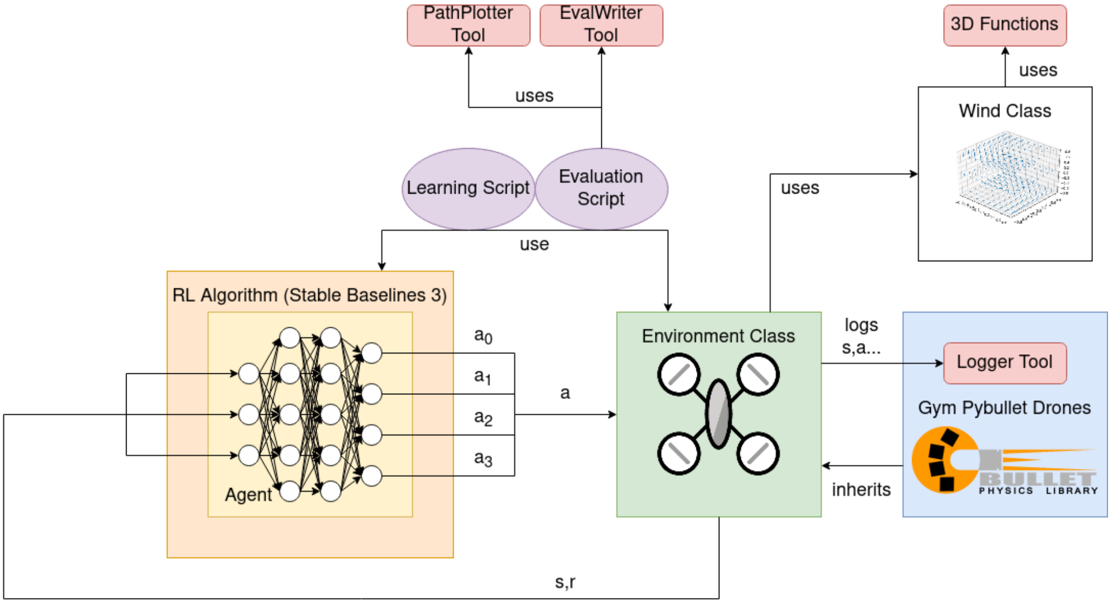


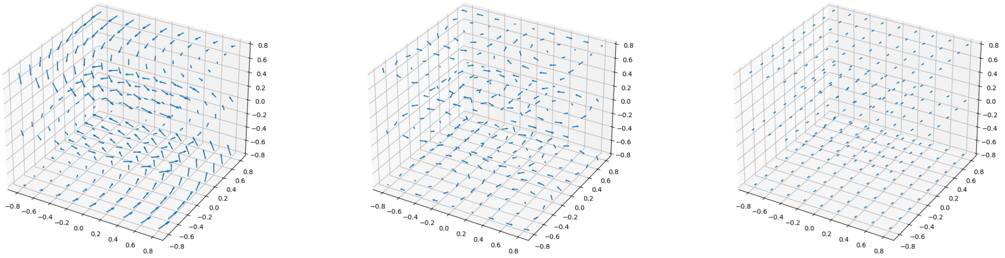
Fig. 4.1: Concept of the implemented software: the different tools(red), scripts(violet) that are used in order to learn the intelligent Agent robust flight control with the use of RL.

4.1 WindSingleAgentAviary Environment Class

The *WindSingleAgentAviary Class* is an implemntation of the environment that models the defined RL problem. It is highly adaptable and has the ability to model a range of MDPs with different action spaces and non-deterministic transitions caused by a random wind field (Section 4.1.1). Independently of this, the class still always models a goal environment, although the goal might be randomly sampled within a defined space (Algorithm 3). These flexible parameters are summed up in different operating modes (Section 4.1.2). Nevertheless, this environement class only models learning to hover in goals. Theoretically, it is compatible with every RL algorithm that allows continous action spaces. On initialization different parameters can be parsed to the environment class.

Tab. 4.1: Overview of the initialization parameters of the WindSingleAgentAviary environment class.

Name	Symbol	Type	Default	Explaination
drone_model		DroneModel	$DroneModel$ $.HB$	The model of the used drone
physics		$Physics$	Physics.PYB	The pybullet physics dynamics
frequency	f_s	Integer	240	The step frequency of the physics engine
aggregate_phy_steps	$\aleph$	Integer	5	The number of physics steps within a step
gui		Boolean	$false$	enables gui
act		ActionType	$ActionType$ $.RPM$	The type of the action
total_force	ω_w	Float	0.0	The wind force bound
mode		Integer	0	The mode of the environment
episode_len	T	Integer	5	The length of an episode
radius	R	Float	0.0	The radius of the half goal sphere
debug		Boolean	$false$	Enables debug mesages



(a) (b) (c)

Fig. 4.2: Visualization of three different wind fields. Vectors represent a force vector that impacts the drone in the matching position. (a) is a random vector field made with a 3D function, (b) a predefined trigonometric vortex vector field and (c) with a random linear vector field.

4.1.1 Wind Class

The *Wind Class* is an essential class in order to simulate a turbulent condition. Basically the class returns a 3 dimensional force vector W based on the x, y, z position of the drone. This force vector is then applied in the environment and pushes the drone in the matching direction. On initialization the wind is given a force bound $\omega_w[N]$ and a type. All vectors are being clipped to this amount of force in order to simulate wind fields of different strengths. In addition, the clipping method adds some Gaussian distributed randomness to each coordinate with the clipped value as mean and a standard deviation of 0.003.

Type 0 simulates a random, constant wind field that applies the same random force vector at each position (Equation (4.1)). The random coordinates r_i are chosen with respect to a Gaussian distribution with the mean of $\frac{\omega_w}{2}$ and a standard deviation of 0.03. As a consequence, the length of the force vector $|\overrightarrow{W}|$ is distributed with a mean of $0.866w$. A visualization of type 0 wind fields can be seen in Figure 4.2c.

$$\overrightarrow{W} = clip(\begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix}) \tag{4.1}$$

$$r_i \sim \mathbb{N}(\frac{\omega_w}{2}, 0.03) \tag{4.2}$$

$$\mathbb{E}|\overrightarrow{W}| = \sqrt{3 \cdot \frac{\omega_w^2}{4}} \approx 0.866\omega_w \tag{4.3}$$

Type 1 is a trigonometric wind field with central vortex. A visualization can be seen in Figure 4.2b.

$$\overrightarrow{W} = clip(\begin{pmatrix} \sin(\pi \cdot x) \cdot \cos(\pi \cdot y) \cdot \cos(\pi \cdot z) \\ -\cos(\pi \cdot x) \cdot \sin(\pi \cdot y) \cdot \cos(\pi \cdot z) \\ \sqrt{\frac{2}{3}} \cdot \cos(\pi \cdot x) \cdot \cos(\pi \cdot y) \cdot \sin(\pi \cdot z) \end{pmatrix}) \tag{4.4}$$

Type 2 is a wind field that is linear in each axis. Like seen in type 0 wind fields a random Gaussian distributed factor r_i is used that indicate how steep each of the linear functions is.

$$\overrightarrow{W} = clip(\begin{pmatrix} r_0 \cdot x \\ r_1 \cdot y \\ r_2 \cdot z \end{pmatrix}) \tag{4.5}$$

$$r_i \sim \mathbb{N}(\frac{\omega_w}{2}, 0.03) \tag{4.6}$$

Type 3 simulates a basic, random wind field with a central vortex. Therefore, it uses random signs.

$$\overrightarrow{W} = clip(\begin{pmatrix} \pm y \\ \pm x \\ \pm z \end{pmatrix}) \tag{4.7}$$

Type 4 is a random wind field with a central vortex, that shows a little more complexity.

$$\overrightarrow{W} = clip(\begin{pmatrix} x \pm y \\ z \pm x \\ y \pm z \end{pmatrix}) \tag{4.8}$$

Type 5 is a completely random wind field based on three random 3D functions $f, f', f'' : \mathbb{R}^3 \to \mathbb{R}$. As a consequence, a lot of different, complex wind field can be created like seen in Figure 4.2a that possesses different functions in each coordinate.

A *3D function* can have a lot of different forms and should be described inductive. There are a base set of functions mapping from $\mathbb{R}^3 \to \mathbb{R}$ (Equation (4.9)). In addition, there are two rules that inductively form the whole set of functions. If g already is a 3DFunction then also $\sin(g), \cos(g), 2 \cdot x \cdot \sin(g), \sqrt{g}, e^g$ are 3DFunctions (Equation (4.10)). If g and h are 3DFunctions then $g + h$ is also a 3Dfunction (Equation (4.11)). Since there could be an endless regression, the induction is closed after only one step in order to avoid a stack overflow in implementation.

Base set:

$$F^0 = \{0, 1, x + y + z, x + y, x + z, y + z, x \cdot y \cdot z, x \cdot y, x \cdot z, y \cdot z\} \tag{4.9}$$

Rules:

$$\forall n < 2 \quad g \in F^n \wedge h \in F^n \rightarrow F^{n+1} = F^n \cup \{g + h\} \tag{4.10}$$

$$\forall n < 2 \quad g \in F^n \rightarrow F^{n+1} = F^n \cup \{\sin(g), \cos(g), 2 \cdot x \cdot \sin(g), \sqrt{g}, e^g\} \tag{4.11}$$

If no type is specified, then the wind field is of a chosen random type. Since type 5 is the most complex type of wind field, the others might not be really needed, because similar wind fields like a vortex can still be approximated with the use of the 3D functions. However, it is still preferable to be able to choose a simple wind field at first before increasing complexity. In addition, there is the possibility to evaluate agents in different wind fields.

4.1.2 Modes

Since the WindSingleAgentAviary class is meant as flexible class that more complex classes can inherit from, it has different modes that influence the position of the goal and the type of the wind (Table 4.2). The use of modes is helpful in order to debug the environment end slowly increment the complexity of the RL problem.

Tab. 4.2: The different Modes of a WindSingleAgentAviary environment.

Mode	Goal	Starting State	Wind
0	$\overrightarrow{g} = \begin{pmatrix} 0 \\ 0 \\ 0.5 \end{pmatrix}$	$\overrightarrow{p_0} = \begin{pmatrix} 0 \\ 0 \\ 0.5 \end{pmatrix}$	no wind
1	$\overrightarrow{g} = \begin{pmatrix} 0 \\ 0 \\ 0.5 \end{pmatrix}$	$\overrightarrow{p_0} \begin{pmatrix} 0 \\ 0 \\ z_{min} \end{pmatrix}$	no wind
2	$\overrightarrow{g} \sim G_{HS}(R)$	$\overrightarrow{p_0} \begin{pmatrix} 0 \\ 0 \\ z_{min} \end{pmatrix}$	no wind
3	$\overrightarrow{g} \sim G_{HS}(R)$	$\overrightarrow{p_0} \begin{pmatrix} 0 \\ 0 \\ z_{min} \end{pmatrix}$	specified type

```
Algorithm 3: Evenly distributed Sampling from a half sphere $G_{HS}$
Input: radius $R$
Output: goal $\overrightarrow{g} = \begin{pmatrix} g_x \\ g_y \\ g_z \end{pmatrix}$
repeat
    $\overrightarrow{\nu} = \begin{pmatrix} \nu_x \\ \nu_y \\ \nu_z \end{pmatrix} \qquad \nu_x, \nu_y \in [-R, R], \quad \nu_z \in [0, R]$
until $|\overrightarrow{\nu}| \leq R$
return $\begin{pmatrix} 0 \\ 0 \\ 0.5 \end{pmatrix} + \overrightarrow{\nu}$
```

Mode 0 and *Mode 1* are simple modes with a static goal $0.5m$ over the ground and no wind, which only differ in the starting state distribution. In Mode 0 the drone is already in the goal and only has to stay there. In Mode 1 it starts on the ground and has to fly to the goal and hover there. Since the coordinate frame of the drone might not be on the lowest point, the drone starts at the height z_{min}.
Mode 2 evenly samples the goal vector from a half sphere that is shifted about 0.5 in positive z direction. This random point sampling in a half sphere is implemented with Algorithm 3. It samples a random vector from inside a cupoid with the side lengths of $2 \cdot R, 2 \cdot R, R$ until it is inside the defined half sphere. As a consequence, the vector in evenly distributed. At first, this algorithm seems pretty inefficient, because in theory the break condition of the loop might never be violated. Hower, since the probability of $|\overrightarrow{\nu}| > R$ is small (Equation (4.12)), the expected amount of loops is approximately 1.738 (Equation (4.14)).

$$P(|\overrightarrow{\nu}| > r) = 1 - \frac{V_{HS}}{V_Q} = 1 - \frac{\frac{2}{3} \cdot \pi R^3}{4 \cdot R^3} = 1 - \frac{1}{6} \cdot \pi \approx 0.476 \tag{4.12}$$

$$\begin{aligned} \frac{d}{dq} \sum_{i=0}^{\infty} q^i &= \frac{d}{dq} \frac{1}{1-q} \qquad \forall |q| < 1 \\ \leftrightarrow \sum_{i=0}^{\infty} i \cdot q^{i-1} &= \frac{1}{(1-q)^2} \qquad \forall |q| < 1 \\ \leftrightarrow \sum_{i=0}^{\infty} i \cdot q &= \frac{q}{(1-q)^2} \qquad \forall |q| < 1 \end{aligned} \tag{4.13}$$

$$\mathbb{E}(n) = \sum_{i=0}^{\infty} (1 - \frac{1}{6} \cdot \pi)^i \cdot i = \frac{(1 - \frac{1}{6} \cdot \pi)}{(1 - (1 - \frac{1}{6} \cdot \pi))^2} \approx 1.738 \tag{4.14}$$

Mode 3 samples the goal with Algorithm 3, but also implements a wind field. The type of the wind field is either given in initialization or random. Also the force bound is given in initialization.

4.1.3 Observation Space & State Space

The state space S defines the state vector of the drone in the environment and possesses a dimensionality of 23. This state is only used inside the environment and consists of the current position x_p, y_p, z_p in each axis, the roll, pitch and yaw angles Θ_p, ϕ_p, ψ_p as well as represented as quaternion q, the velocities $\dot{x_p}, \dot{y_p}, \dot{z_p}$, the angular velocities $dot\Theta_p, \dot{\phi}_p, \dot{\psi}_p$, the goal position x_g, y_g, z_g and the last clipped action a_t. With the use of this drone state space the observations are calculated.

The observation space $\daleth$ is a subset of the state space S with the dimensionality of 15. The observation space is implemented as a *spaces box* of type *float32* which are mainly ranged within $[-1, 1]$ with the exception of the z coordinate of the position z_p and goal z_g. Because there is floor defined as a plain at the height of 0, which inherits a collision body, the drone is not able to reach a negative z coordinate. As a consequence, these are ranged to $[0, 1]$.

$$\daleth_t = (p_x, p_y, p_z, \Theta_p, \phi_p, \psi_p, \dot{p_x}, \dot{p_y}, \dot{p_z}, \dot{\Theta}_p, \dot{\phi}_p, \dot{\psi}_p, g_x, g_y, g_z) \tag{4.15}$$

Each observation σ_t consists of the current position x_p, y_p, z_p in each axis, the roll, pitch and yaw angles Θ_p, ϕ_p, ψ_p, the velocities $\dot{x_p}, \dot{y_p}, \dot{z_p}$, the angular velocities $\dot{\Theta}_p, \dot{\phi}_p, \dot{\psi}_p$ and the goal position g_x, g_y, g_z (Equation (4.15)). These observations are given to the NN in order to approximate the optimal action a that satisfies the defined RL problem.

Like previously mentioned, all observations are ranged in order to prohibit inputs of different magnitude that could disrupt the learning process. This is done with the use of the method *_clipAndNormalizeState* which gets the current state s_t and normalizes it to the defined range (Equation (4.16)). First, it clips it to predefined values v and then normalizes it by dividing with the matching predefined value $v_i, i \in [0, 11]$. If wanted, a warning can be printed each time a state parameter has to be clipped. By clipping x and y to a value in $[-20, 20]$ there is a predefined limit of maximal distance, in which there is a reasonable option to learn robust flight. Analogue the z component is clipped to $[-10, 10]$, roll and pitch to $[-\pi, \pi]$, the translation velocities to $[-3, 3]$ in x, y and to $[-2, 2]$ in z direction.

$$\daleth_t \leftarrow clip(s_t)/v \tag{4.16}$$

Tab. 4.3: The different ActionTypes with the corresponding dimensionality of the action, its range and how it is processed.

ActionType	dim	range	processing
one_d_rpm	$\|a\| = 1$	$a_i \in [-1, 1]$	$rpm = (hover\_rpm \cdot (1 + 0.05 \cdot a)) \cdot (1, 1, 1, 1)$
rpm	$\|a\| = 4$	$a_i \in [-1, 1]$	$rpm = hover\_rpm \cdot (1 + 0.05 \cdot a)$
vel	$\|a\| = 4$	$a_i \in [-1, 1]$	$rpm = pid(S, vel = limit \cdot \|a_3\| \cdot \frac{(a_0, a_1, a_2)}{\|(a_0, a_1, a_2)\|})$

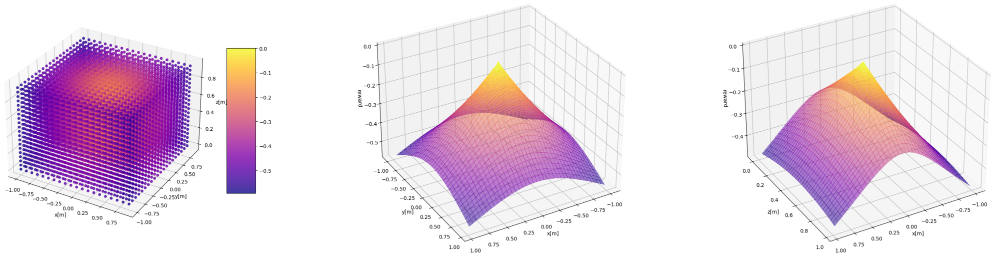
4.1.4 Action Space

The WindSingleAgentAviary Environment possesses three different type of action spaces, that are processed in different ways to the *rpm (rotation per minute)* of the four motors (Table 4.3). Nevertheless, all action types are continuous and are ranged in $[-1, 1]$.

one_d_rpm is a one dimensional action space. The chosen action a is processed to range of 0.05 about the *hover_rpm* to a 4-tupel of rpms. The hover_rpm is defined as the rpm that corresponds to hovering (Section 2.1.1). The 4-tupel is then forwarded to the motors. As a consequence of this limitation, the drone can only perform hovering, rising and falling movements and can not influence its x, y position or Θ, ϕ, ψ. Also, the translational speed v_z is limited by the size of 0.05.

rpm is a 4 dimensional action space. The actions are processed within a range of 0.05 about the hover_rpm to a 4-tupel of rpms. The drone is not limited in any dimensionality, but the task increases in complexity. Due to Equation (2.1), Equation (2.2), Equation (2.3), Equation (2.4) even a small difference in rpms can lead to an unstable flight or even a crash, because the roll or pitch angle is too high. Also, all translational and rotational speeds are limited by the size of 0.05. Because of the higher complexity, it is expected, that the training takes noticeable more time.

vel is a top level, 4-dimensional action space. The action consists of a velocity vector $v = (a_0, a_1, a_2)$ and its size a_3. Since it is a top level action space, the actions are not corresponding directly to the rpms, but a pid controller (Section 2.1.2) is used in order to control the rpms. The basic pid controller is part of Gym Pybullet Drones [PZZ$^+$21] and must be tuned for the used quadrocopter. It mainly receives the state S of the drone, as well as the targeted velocity, which is calculated with the use of the actions. Therefore, a_3 is multiplied with speed limit of the drone in order to derange the action. Also, the velocity vector $\overrightarrow{v}$ is normalized to a length of 1.
The drone is not limited in any dimensionality and the task is less complex than setting rpm directly. Stability of the flight is now mainly controlled by the pid controller, so it is bounded by the typical pid constraints in harsh environments and not adaptable.



(a) (b) (c)

Fig. 4.3: Visualization of the used reward function with the goal $(0, 0, 0.5)$ and a color scale for different positions in space.

4.1.5 Reward Function

The reward mainly assesses how the agent learns and which action are benefitial in which state. In this environment the reward consists of a classic reward function, mainly depending on the distance to the goal $dist_t$ (Equation (4.17)), and a set of constraints. The reward function is an exponential function with a negative exponent. The factor 0.6 defines the steepness of the function. Figure 4.4 shows a plot of the reward function over a small field of distances up to $2m$. Due to the main exponential part, the reward function is also bounded to the range $[-1, 0]$ with the gradient increasing close to a distance of 0. This is helpful in order to get a precise behavior. At hughe distances the gradient is small, but this does not really matter that much because so hughe distances are not expected.

$$r_t = e^{-0.6 \cdot dist_t} - 1 = e^{-0.6 \cdot |\overrightarrow{g} - \overrightarrow{p_t}|} - 1 \tag{4.17}$$

$$\lim_{dist_t \to \infty} r_t = -1 \tag{4.18}$$

$$\lim_{dist_t \to 0+} r_t = 0 \tag{4.19}$$

As a consequence, the reward function only depends on the current relative position to the goal. The action with the highest reward is the action that concurrs with the optimal velocity vector towards the goal while still guaranteeing a safe flight. Figure 4.3 also shows this relation with the standard goal $g = (0, 0, 0.5)$ of mode 0 and mode 1. The reward towards a fixed goal is point semetric with the global maximum when $dist_t = 0$.

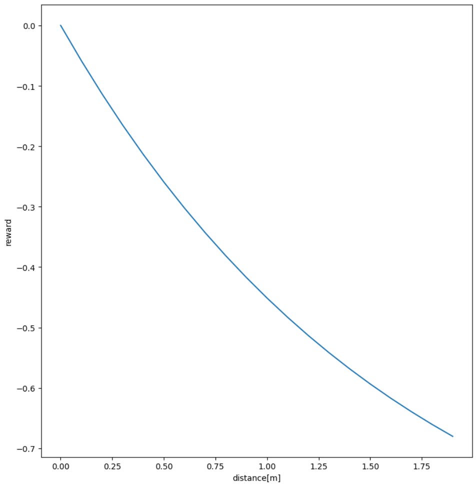


Fig. 4.4: Visualization of the reward functions over the total distance[m]

4.1.6 Constraints

Besides, the reward function is also unfluenced by a set of constraints, which are logical rules that define a safe flight. If one of the rules is broken, the episode is ended by returning a done d value that is true. In addition, there is a punishement reward of -200 that is added to the classic reward function.

This constraints can be benefitial for training time, because the episodes are ended early. Also, unsafe flight is highly unattractive for the agent because of the punishement reward.

$$p_z \leq z_{min} \wedge \frac{t_s}{f_s} \geq 0.1s \rightarrow r_t = e^{-0.6 \cdot |goal-pos|} - 1 - 200 \wedge d = True \tag{4.20}$$

$$\Theta_p > 1 \vee \phi_p > 1 \rightarrow r_t = e^{-0.6 \cdot |goal-pos|} - 1 - 200 \wedge d = True \tag{4.21}$$

The first constraint (Equation (4.20)) is checking if the UAV crashed. Therfore, it compares the height of the coordinate frame with the starting height z_{min} and checks that already some time has passed. This is needed, because otherwise the constraint would fire directly when starting. If this rule fires, 200 is subtracted from the reward as punishement and the episode is ended. As a consequence the agent learns to avoid crashing and to start fast.

The second constraint (Equation (4.21)) is checking the roll and the pitch angle. If one of these angles is bigger than $\approx 57,296°$ the rule fires and leads to a terminal state and a punishement. As a consequence the agent learns stable flight and avoids this undesired states. The roll and pitch angle should be kept as close as possible to zero. However, in order to change the x, y coordinates a roll and pitch angle is needed. Also, this might be benefitial countering wind.

4.1.7 Optimal Rewards

Based on the modes (Section 4.1.2) different optimal rewards (Equation (4.17)) can be defined, that are helpful in order to determine how good the learned decision making is. *Mode 0* has an optimal reward of $r_{opt} = 0$, which is basically just giving the action 0 to all motors. This leads to hovering and staying in the goal. *Mode 1* is due to its fixed goal position g rather easy to define:

$$r_{opt} = \sum_{t=0}^{f_c \cdot T} r_{opt_t} \tag{4.22}$$

$$d_0 = 0.5 \tag{4.23}$$

$$dist_{t+1} = dist_t - |\overrightarrow{v_{opt}}| \cdot \frac{1}{f_c} \tag{4.24}$$

The distance at start d_0 is always 0.5. An optimal decision of the agent corresponds to an optimal velocity vector v_{opt} that minimizes the distance to the goal with the given bounds of the maximum speed at the α of the action processing. The reward (Equation (4.17)) can then be used for every control step and summed up to the total optimal reward r_{opt} (Equation (4.22)).

Since every mode apart from mode 0 and mode 1 does not possess a fixed goal position, only an expected optimal reward $\mathbb{E}(R_{opt})$ can be defined. Therefore, an expected starting distance $\mathbb{E}(d_0)$ to a goal that is equally distributed in a half sphere:

$$\mathbb{E}(d_0) = \sqrt{\mathbb{E}(d_x)^2 + \mathbb{E}(d_y)^2 + \mathbb{E}(d_z)^2} = \mathbb{E}(d_z) \tag{4.25}$$

$$\mathbb{E}(d_x) = \mathbb{E}(d_y) = 0 \tag{4.26}$$

$$\begin{aligned} \mathbb{E}(d_z) &= 0.5 + \mathbb{E}(z^*) = 0.5 + P(z) \cdot \int\int\int z dx dy dz \\ &= 0.5 + \frac{1}{V} \int_0^{2\cdot\pi} d\phi \int_0^{\frac{\pi}{2}} \sin(\Theta) d\Theta \int_0^R r^2 r \cos(\Theta) dr \\ &= 0.5 + \frac{\frac{\pi}{4} R^4}{\frac{2}{3}\pi R^3} = 0.5 + \frac{3}{8} R \end{aligned} \tag{4.27}$$

The expected distance in each axis $\mathbb{E}(d_0)$ depends linear on the chosen maximum radius R. The expected distance in x and y direction is equal to zero due to the symmetrie of the half sphere.

Also mode 3 possess a wind class that tries to disrupt the agent. Since there are random wind fields and the corresponding force is not normally distributed, the optimal reward can just be estimated downwards. Therefore, the used maximum wind force ω_w is used in order to approximate the maximum of disruptive distance and use it in order to update Equation (4.24).

$$dist_{t+1} = dist_t - |v_{opt}| \cdot \frac{1}{f_c} + \frac{\omega_w}{m} \cdot \frac{1}{f_c} \tag{4.28}$$

4.2 Scripts & Evaluation Tools

In order to learn a policy π based on the implemented environment a learning script was implemented that offers parsing different arguments ξ. These arguments define the learning process and different environment arguments $\tilde{\lambda}$, that are used on initialization of the WindSingleAgentEnvironment class. Also, the evaluation script offers a similar script with similar arguments ξ in order to evaluate the policy π. Therfore, it uses two different evaluation tools and the gym-pybullet-drones logger.

4.2.1 Learning Script

The learning script (Algorithm 4) receives a set of parsed arguments ξ (Table 4.4) and checks them on contradictions. If a predefined contradiction is found a *ParsingError* is raised stating the contradicting parsed arguments with a message. Then, the training environment ϵ_t is created with the parsed environment args $\tilde{\lambda} \subset \xi$ and the policy is defined based on the parsed algorithm. The policy mocel π is then either created or loaded from a specified file. Next, a evaluation environment ϵ_e is created with the same arguments and the evaluation callbacks are defined. Then depending on the parsed curriculum parameter ζ the policy is either trained commonly or with a curriculum method (Algorithm 5). At last, the policy model π is safed with the parsed name.

Tab. 4.4: Overview of the Arguments ξ parsed to the Learning Script

Name	Symbol	Type	Default	Explaination
–cpu	η	Integer	1	Amount of parallel training environments
–drone		DroneModel	$Dronemodel.HB$	The drone model
–steps	t_{total}	Float	1e7	Training steps
–mode		Integer	0	Environment mode
–load	ι	Boolean	$false$	Load an existing policy
–total_force	$\omega[N]$	Float	0.0	The upper force bound
–radius	$R[m]$	Float	0.0	The default radius
–debug_env		Boolean	$false$	Enables debug print statements in the environment
–episode_len	$T[s]$	Integer	5	Amount of seconds of each episode
–act		ActionType	$ActionType.RPM$	The type of action of the NN
–name		String	”$results/success\_model.zip$”	The name of the model after training
–curriculum	ζ	Boolean	$false$	Use curriculum learning
–algo		String	”*ppo*”	The used algorithm

```
Algorithm 4: Learning Script
Input: parsed arguments $\xi$
Check the parsed arguments $\xi$ on contradiction and raise ParsingError if needed
Create training environment $\epsilon_t$ with the parsed environment args $\tilde{\lambda} \subset \xi$
Define relevant attributes for the policy network
if parsed load parameter $\iota \in \xi$ then
    Load policy model $\pi$ with matching algorithm
else
    Create policy model $\pi$ with matching algorithm
Create evaluation environment $\epsilon_e$ with the parsed environment args $\lambda \subset \xi$
Define evaluation callbacks
if parsed curriculum parameter $\zeta \in \xi$ is parsed as false then
    Learn the model with the parsed amount of time steps
else
    Learn the model with the curriculum method (Algorithm 5)
Save the model $\pi$ with the parsed preferred name
```

Linear Curriculum Learning

Linear Curriculum Learning (LCL) is a method that changes environment parameters during training process. In contrast, the classic approach uses the parsed algorithm in order to optimize the policy on a firm environment. LCL is a simple adaption that changes an environment parameter after a specified amount of steps. In this case, the radius parameter R of the environment is adjusted. In Theory, the learning starts with a rather simple approach with a fixed goal and then steadily increases the difficulty of the RL problem, which may benefit training performance.
The LCL algorithm (Algorithm 5) receives the total steps within a unchanged environment t_total, a set of environment args $\tilde{\lambda}$, the policy model π, the amount of parallel environments η and the specified name of the model. Then the maximum radius R_{max} is saved from $\tilde{\lambda}$ and the current radius R is initialized with a value of 0. Until the current radius is bigger or equal to the maximum radius, the radius in $\tilde{\lambda}$ is set to the current radius. Also, the training environment ϵ_t with the new arguments is initialized and set, the evaluation environment ϵ_e is initialized and evaluation callbacks are defined. Then, the policy model is learned in a common way with t_total steps. Due to the evaluation callbacks it may be that the learning starts earlier if the policy outperforms a defined performance threshold. At last, the policy is saved, the best policy π_{best} is renamed and the current radius R is linear increased.

Algorithm 5: Linear Curriculum Learning Algorithm

```
Input: total steps $t_{total}$, set of environment args $\tilde{\lambda}$, model $\pi$, number of parallel
       environments $\eta$, name
Get radius $R_{max}$ from $\tilde{\lambda}$
$R = 0.0$
while $R \leq R_{max}$ do
    Change radius in $\tilde{\lambda}$ ro R
    Initialize training environment $\epsilon_t$ with $\tilde{\lambda}, \eta$ as parameters
    Set $\epsilon_t$
    Initialize evaluation environment $\epsilon_e$ with $\tilde{\lambda}, \eta$ as parameters
    Define evaluation callback
    Learn the model $\pi$ with the amount of steps $t_{total}$
    Save $\pi$ and rename best policy $\pi_{best}$
    $R = R + 0.2$
```

4.2.2 Evaluation Script

The evaluation script (Algorithm 6) is structured similar to the algorithm script and also receives a set of parsed arguments ξ (Table 4.5). First, it checks ξ on contradiction and raises a ParsingError if needed. Then, the evaluation environment ϵ_e is created with the parsed set of environment args $\tilde{\lambda}$ and the policy model π is loaded. The *EvalWriter* class (Section 4.2.3) is then instatiated and used for evaluation.
If the gui parameter $\tilde{\rho}$ is parsed a single episode is visualized with the pybullet gui. Therefore, a test environment ϵ_t, a logger and a pathplotter class is instantiated. At the start, the environment is reset and returns an initial observation $\neg_0$. In order to sync the simulation time to the real word time the current real time t^* is safed. Until the episode is over a fixed amount of commands are executed:

First π is used in order to decide the action a. The action is then given to ϵ_t and a step is executed that returns a reward r_t, a done value d and an info. Then, the current pose vector $\overrightarrow{p_t}$ is added to the pathplotter class and the current drone state s_t is logged with the logger. At last, the simulation time t_{sim} is synced to the real time t^*.

After this visualization the environment is closed, the plotted path and the logs are shown.

Tab. 4.5: Overview of the Arguments ξ parsed to the Evaluation Script

Name	Symbol	Type	Default	Explaination
–drone		DroneModel	*Dronemodel.HB*	The drone model
–episodes		Float	100	Evaluation Episodes
–mode		Integer	0	Environment mode
–total_force	$\omega[N]$	Float	0.0	The upper force bound
–radius	$R[m]$	Float	0.0	The default radius of the goal sphere
–debug_env		Boolean	*false*	Enables debug print statements in the environment
–episode_len	$T[s]$	Integer	5	Amount of seconds of each episode
–init	p_0	List[Float]	*none*	Starting state
–gui	$\tilde{\rho}$	Boolean	*true*	Show gui
–act		ActionType	*ActionType.RPM*	The type of action of the NN
–name		String	”*results* */success* *_model.zip*”	The name of the model after training
–algo		String	”*ppo*”	The used algorithm

Algorithm 6: Evaluation Script

```
Input: parsed arguments $\xi$
Check the parsed arguments $\xi$ on contradiction and raise ParsingError if needed
Create evaluation environment $\epsilon_e$ with the parsed environment args $\tilde{\lambda} \subset \xi$
Load the policy model $\pi$
Instantiate an EvalWriter and evaluate the model $\pi$
if parsed gui parameter $\tilde{\rho} \in \xi$ then
    Instantiate a test environment $\epsilon_t$, logger and a pathplotter
    Reset environment and safe an observation $\daleth_0$
    Safe current real time $t^*$
    repeat
        Get action $a$ from $\pi$ based on $\daleth_t$
        Step $\epsilon_t$ and receive an observation $\daleth_t$, reward $r_t$, a done value $d$ and a info
        Add current pose $\overrightarrow{p_t}$ to pathplotter and log state $s_t$
        Sync simulation time $t_{sim}$ to the real time $t^*$
    until episode is done
    Close test environment, show the plotted path and logs
```

Tab. 4.6: The evaluation metric.

Name	Form	Explanation
rate	$x\%$	percentage of succeeding episodes
time rate $\beth$	$x\%$	percentage of time in goal sphere
settled rate	$x\%$	percentage of settled episodes
dist($\frac{T}{2}$)	$(x \pm y)m$	average distance half way through episode
dist(T)	$(x \pm y)m$	average distance at the end of the episode
overshoot	$(x \pm y)m$	average overshoot
reward	$x \pm y$	average accumulated episode reward

4.2.3 Evaluation Tools

In order to evaluate the policies and visualize the flight path, the *EvalWriter* and the *PathPlotter* class were implemented. The EvalWriter class aims at providing a static metric that emphasises how good a chosen policy π is. Therefore, it evaluates different classic RL metrics like the average accumulated reward and also some basic control theory metrics like overshoot (Table 4.6). The PathPlotter class aims at visualizing a single episode in a 3D plot.

EvalWriter Class

The EvalWriter class is intialized with a name, a number of evaluation steps $\gimel$, the path to the xlsx file it writes to, the environment ϵ, the episode length T and the threshold $\Omega[m]$ that defines a sphere around the goal. If the drone is inside this sphere and the distance is lower than Ω it counts a beeing in the goal. In theory this parameter could be fixed to 0, but it is easier to allow a minimum distance in order to compare the policies better.The EvalWriter has an update function (Algorithm 7), that is used in order get the new distances from ϵ and update the metric parameters and is used in the evaluation algorithm (Algorithm 8). Then, the data can be visualized with the *write* method.

Algorithm 7: Update Algorithm of EvalWriter

```
Get current dist and $t_{sim}$ from environment $\epsilon$ and save it
if $e = 1$ then
    Add pose of the drone from $\epsilon$ to pathplotter
if $t_{sim} = \frac{T}{2}$ then
    Save current dist to matching set of distances
if $t_{sim} = T$ then
    Save current dist to matching set of distances
if episode already reached goal and overshoot then
    Save current dist to overshoot set
Check if the episode succeded in this time step
```

Algorithm 8: Evaluation Algorithm of EvalWriter

```
Input: policy $\pi$
Output: tupel $(\mu_r, \sigma_r)$
Get mean reward $\mu_r$ and the std of the reward $\sigma_r$ from evaluate_policy$(\pi, \epsilon, e)$
reset $\epsilon$ and receive $\daleth_t$
for $j = 0 \to \gimel$ do
    repeat
        Get action $a$ from $\pi$ with input $\daleth_t$
        Step $\epsilon$ and get the new observation $\daleth_t$, reward $r_t$, a done signal $d$ and an
          info
        Update EvalWriter
        if $d$ then
            Reset $\epsilon$
    until done
    if not $j = \gimel$ then
        Use Housekeeping of the EvalWriter class
Close EvalWriter and $\epsilon$.
return $(\mu_r, \sigma_r)$
```

PathPlotter Class

The PathPlotter class is a basic class that visualizes the path taken by drone under a policy π. It is mostly used in the EvalWriter class when only a single episode is evaluated, but also can be used seperatly. It mainly helps understanding the current decision making of a policy. On fixed goal modes it literaly sums up the complete performance in one plot.

It is initiliaized with a set of goals $\overrightarrow{g}$. With the use of the *addPose* method the position of the drone at different time steps can be added. After the episode, the drone path can be visualized with a *Matplotlib 3D plot* by using the *show* method, which creates plots like

5 Evaluation

5.1 Drone Model

5.2 Setup

5.3 Results

5.3.1 Fixed Goal Modes

Policies

Tab. 5.1: Overview of the evaluated Policies on fixed goal modes

Name	Algorithm	ActionType	$\frac{f_s}{\aleph}$	t_{total}	Mode
PPO4D_0.zip	PPO	rpm	$48Hz$	$5e6$	0
SAC4D_0.zip	SAC	rpm	$48Hz$	$5e6$	0
PPO1D_1.zip	PPO	one_d_rpm	$48Hz$	$5e6$	1
SAC1D_1.zip	SAC	one_d_rpm	$48Hz$	$5e6$	1
PPO4D_1.zip	PPO	rpm	$48Hz$	$5e7$	1
SAC4D24_1.zip	SAC	rpm	$24Hz$	$5e7$	1
SAC4D48_1.zip	SAC	rpm	$24Hz$	$5e7$	1
π_{opt}		$rpm/$ one_d_rpm			

Training Comparison

Overall Comparison

$$r_{opt} = \begin{cases} 0 & \text{if } mode = 0 \\ -4.874 & \text{if } mode = 1 \wedge \frac{f_s}{\aleph} = 48Hz \\ -2.39 & \text{if } mode = 1 \wedge \frac{f_s}{\aleph} = 24Hz \end{cases} \tag{5.1}$$

$$\beth_{opt} = \begin{cases} 1 & \text{if } mode = 0 \\ 0.83750 & \text{if } mode = 1 \end{cases} \tag{5.2}$$

Tab. 5.2: Evaluation of the Policies with mode 1

	PPO1D_1	SAC1D_1	PPO4D_1	SAC4D24_1	SAC4D48_1
Reward r_e					
Optimality$_r$ $\frac{r_{opt}}{r_e}$					
Sucess					
Time Rate $\beth$					
Optimality$_\beth$ $\frac{\beth}{\beth_{opt}}$					
dist$(\frac{T}{2})$					
dist(T)					
Settled					
Overshoot					

Comparison for different episode lengths

Hovering in Goal

Comparison of control frequencies

5.3.2 Random Goal Modes

Policies

Tab. 5.3: Overview of the evaluated Policies on random goal modes

Name	Algorithm	ActionType	$\frac{f_s}{\aleph}$	t_{total}
SAC4D__2.zip	SAC	rpm	$48Hz$	$1e8$
SAC4Dcurri__2.zip	SAC	rpm	$48Hz$	$5e7$
SAC4Dsp__curri__2.zip	SAC	rpm	$48Hz$	$5e7$

Training Comparison

Overall Comparison

Comparison for different radii

5.3.3 Wind Disruption

6 Conclusion & Future Work

6.1 Of Migrating to a real Drone

6.2 Of Hovering, Pathfollowing & Pathplanning

6.3 Of Improvments in Self-Pased Curriculm Learning

6.4 Of Policy Hierarchy RL for Sub-Problem-Solving

List of Symbols

Θ	Roll Euler Angle
ϕ	Pitch Euler Angle
ψ	Yaw Euler Angle
$M_i \quad i \in [1,4]$	Motor of a Quadrocopter
$\tilde{f}$	Upward Thrust of a UAV
$\omega_i \quad i \in [1,4]$	Rotational speed of a propeller of a Quadrocopter
$\omega_i^* \quad i \in [1,4]$	Changed Rotational speed
b	Quadrocopter constant
u_Θ, u_ϕ, u_ψ	Rotational movement
p_{3D}	Pose in 3D space
x_p, y_p, z_p	Coordinates in 3D space
F	Upward Thrust Force
G	Gravitational Force

g	Mission Goal
a	Attitude
a^*	Estimated Attitude
e_a	Attitude Error
S	Set of Motor Signals
p	Estimated Pose
p_{3D}^*	Estimated 3D Pose
$(K_p, K_i, K_d), (K, T_n, T_v)$	Tuple of PID controll gains
$u(t)$	Control Signal

$e(t)$	Error Signal
$\tilde{T}$	Sampling Time Period
f'	Throttle Coefficient
$m_{i,\Theta}, m_{i,\phi}, m_{i,\psi}$	Mixer Values of the Motors
Λ	Set of Waypoints
$\lambda_i \qquad i \in [0...n]$	Waypoint

S	Set of states
A	Set of actions
R	Reward function
P	State Transition probability
p_0	Starting State Distribution
s_t	Observed State at timestep t
s_t^*	Real State at timestep t
a_t	Action at timestep t
r_t	Reward at timestep t
π	Policy
π^*	Optimal Policy
τ	Trajectory
T	Amount of steps in a trajectory
γ	Discount factor
$V^\pi(s)$	Value Function under a policy
$V^*(s)$	Optimal Value Function
$Q^\pi(s,a)$	Q Function under a policy
$Q^*(s,a)$	Optimal Q Function
x_i	Input to a NN
w_i	Weight of edge in a NN

$g()$	Activation Function
Θ	Policy Parameters
ϕ	Value Function Parameters in PPO
$L(s_t, a_t, \Theta_{old}, \Theta)$	Policy Loss in PPO
$g(\epsilon, A)$	Surrogate Objective
ϵ	Learning Rate
A	Advantage Function
α	Trade-Off coefficient in SAC
H	Entropy
$\mathcal{D}$	Replay Buffer
$L(\phi_i, \mathcal{D})$	Loss Function in SAC
d	Done Value
$y(r, s', d)$	Target in SAC
$\tilde{a}$	Next Action sampled from the policy
$\mathcal{B}$	Batch of Transitions
ρ	Polyak averging hyperparameter
$\overrightarrow{g}$	Goal Vector
g_x, g_y, g_z	Goal Coordinates
p	Position Tupel
$\overrightarrow{p_t}$	Position Vector at time step t
$\dot{p}_i \quad i \in \{x, y, z\}$	Linear Velocities
$\dot{\Theta}, \dot{\phi}, \dot{\psi}$	Angular Velocities
σ	Sensor Values
f_s	Simulation Frequency
$\aleph$	Number of physics steps within a step
ω_w	Wind Force Bound

T	Episode length
R	Radius
$\overrightarrow{W}$	Wind Force Vector
$\overrightarrow{p_0}$	Starting State Vector
z_{min}	Minimum height of Coordinate Frame
$\daleth$	Observation Space
$\daleth_t$	Observation at time step t
$v_i \quad i \in [0, 11]$	Predefined Clipping Values
$dist_t$	Distance at time step t
t_s	SImulation time step
r_{opt}	Optimal Reward
f_c	Control Frequency
m	Mass of UAV

ξ	Set of parsed agruments to a script
$\tilde{\lambda}$	Set of environment arguments
ν	Amount of parallel training environments
t_{total}	Training steps
ι	Parsed load parameter
ζ	Parsed curriculum parameter
ϵ_t	Training environment
ϵ_e	Evaluation environment
ϵ	Environment
R_{max}	Maximum radius in LCL
$\tilde{\rho}$	Parsed gui parameter
t^*	Real time
t_{sim}	Simulation time

Σ	Threshold of EvalWriter
$\gimel$	Evaluation episodes
$\beth$	Time Rate
μ_r	Mean reward
σ_r	Std reward

List of Abbreviations

DOF	Degree of Freedom
FC	Flight controller
PID controller	Proportional-Integral-Differential controller
RL	Reinforcement Learning
NLP	Natural Language Processing
ML	Machine Learning
MDP	Markov Decision Process
NN	Neural Network
PPO	Proximal Policy Optimization
SAC	Soft Actor-Critic

Bibliography

[Bis94] Chris M. Bishop: Neural networks and their applications. *Review of Scientific Instruments*, 65(6):1803–1832, June 1994, 10.1063/1.1144830, ISSN 0034-6748. `https://doi.org/10.1063/1.1144830`.

[CB21] Erwin Coumans and Yunfei Bai: Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[DJS20] Dr. Roland Schwaiger Dr. Joachim Steinwendler: *Neuronale Netze programmieren in Python*. Rheinwerk Verlag, 2020.

[FWXY20] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang: A theoretical analysis of deep q-learning. In Alexandre M. Bayen, Ali Jadbabaie, George Pappas, Pablo A. Parrilo, Benjamin Recht, Claire Tomlin, and Melanie Zeilinger (editors): *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pages 486–489. PMLR, 10–11 Jun 2020. `https://proceedings.mlr.press/v120/yang20a.html`.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[KMWB19] William Koch, Renato Mancuso, Richard West, and Azer Bestavros: Reinforcement learning for uav attitude control. *ACM Transactions on Cyber-Physical Systems*, 3(2):22, 2019.

[MP43] Warren S McCulloch and Walter Pitts: A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[NBMB07] Derek R Nelson, D Blake Barber, Timothy W McLain, and Randal W Beard: Vector field path following for miniature air vehicles. *IEEE Transactions on Robotics*, 23(3):519–529, 2007.

[PZZ$^+$21] Jacopo Panerati, Hehui Zheng, SiQi Zhou, James Xu, Amanda Prorok, and Angela P. Schoellig: Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[RHG+21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann: Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. `http://jmlr.org/papers/v22/20-1364.html`.

[SSA17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya: Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[SSS13] P.B. Sujit, Srikanth Saripalli, and J.B. Sousa: An evaluation of uav path following algorithms. In *2013 European Control Conference (ECC)*, pages 3332–3337, 2013, 10.23919/ECC.2013.6669680.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov: Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[SZWJ22] Zhong Shi, Fanyu Zhao, Xin Wang, and Zhonghe Jin: Deep kalman-based trajectory estimation of moving target from satellite images. In *2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, volume 10, pages 71–75. IEEE, 2022.

[Tur50] A.M. Turing: *The Imitation Game*. 1950.