Learning to Fly a VTOL Drone

Project for Lecture "Neural Networks for Physics Students"
Technical University of Munich

Leonard Uscinowicz

February 9, 2023

Contents

1 Introduction

The goal of this problem is to achieve a minimum-time flight for a drone through a sequence of waypoints in the presence of obstacles while utilizing the full drone dynamics. Previous approaches have relied on simplified dynamics or polynomial trajectory representations that do not fully utilize the actuator potential of the drone, leading to suboptimal solutions. While recent methods have been able to plan minimum-time trajectories, they do not take obstacles into account during execution, making them prone to errors due to model mismatch and inflight disturbances.

To overcome this limitation, a combination of deep reinforcement learning and classical topological path planning is employed to train robust neural-network controllers for minimum-time quadrotor flight in cluttered environments. The resulting neural network controller demonstrates significantly improved performance of up to 19% compared to state-of-the-art methods. Furthermore, the learned policy simultaneously solves the planning and control problem online, accounting for disturbances and achieving much higher robustness. As a result, the proposed method achieved a 100% success rate of flying minimum-time policies without collision, while traditional planning and control approaches achieved only 40%. The proposed method has been validated through simulations and real-world experiments.

2 Neural networks literature research

2.1 Differences between conventional training and reinforcement learning

Conventional training, also known as supervised learning, is a method of training neural networks where the network is provided with a set of inputs and corresponding desired outputs, and the network's parameters are adjusted to minimize the difference between the network's actual outputs and the desired outputs. This is typically done using an algorithm such as backpropagation. The goal of conventional training is to make the network generalize well to new unseen data, by learning the underlying patterns and relationships present in the training data.

Reinforcement learning, on the other hand, is a method of training neural networks where the network learns to make decisions in an environment by maximizing a reward signal. The network learns to map observations of the environment to actions, through trial and error, and receives feedback in the form of rewards or penalties for its actions. This is typically done using an algorithm such as Q-learning.

Table 1: Different training methods

Learning method	conventional	reinforcement
training data	input and expected output	state and environment
goal of the training	reduce loss-function	maximize reward-function

In reinforcement learning, there is no explicit training dataset as in supervised learning, instead the agent interacts with the environment, receives a reward or penalty based on its actions and adjusts its actions accordingly. The agent's goal is to learn a policy that maximizes the expected total reward over time. Another key difference between the two is that in supervised learning, the network's parameters are adjusted based on the error between the network's output and the desired output, whereas in reinforcement learning, the network's parameters are adjusted based on the expected future rewards for different actions.

2.2 The actor critic method

Actor-Critic is a type of reinforcement learning algorithm that combines two separate neural networks, the actor and the critic, to learn to make decisions in an environment. The actor network is responsible for learning the

policy, which maps the current state of the environment to an action to be taken. The goal of the actor network is to learn a policy that maximizes the expected total reward over time. The actor network takes the current state of the environment as input and outputs the probability distribution over the actions that can be taken in that state.

The critic network, on the other hand, is responsible for evaluating the value of different states or state-action pairs. The goal of the critic network is to learn to predict the expected future rewards of different actions given a particular state. The critic network takes the current state of the environment and the action taken by the actor network as input and outputs a value that represents the expected future rewards. The two networks work together to improve the policy over time. The critic network provides feedback to the actor network by evaluating the value of different actions and the actor network uses this feedback to adjust its policy to select actions that are expected to lead to higher rewards. The combination allows the actor-critic method to have the advantages of both methods: it can learn from raw observations, similar to policy-based methods, and it can handle large and continuous action spaces, similar to value-based methods.

2.3 Proximal Policy Optimization

Proximal Policy Optimization is a type of policy gradient algorithm, which means that it uses the gradients of the policy with respect to its parameters to adjust the parameters in the direction that increases the expected total reward. One of the key features of PPO is its use of a "clipped" objective function that helps to prevent the policy from changing too much in one step. The objective function is a combination of the expected total reward and a term that measures the difference between the new policy and the old policy. The expectation is taken over a small batch of data collected from the environment using the current policy.

The term that measures the difference between the new policy and the old policy is called the "surrogate loss". This term is multiplied by a "clipping ratio" which is a hyperparameter that controls how much the policy can change in one step. The clipping ratio is typically set to a value between 0.1 and 0.3. If the surrogate loss is greater than the clipping ratio, it is clipped to the clipping ratio, preventing the policy from changing too much. This helps to stabilize the learning process and prevent the policy from oscillating or diverging.

PPO also uses a technique called "value function regularization" to improve the stability of the learning process. The value function is an estimate of the expected future rewards for a given state. This value function is used to adjust the policy based on the expected future rewards. PPO regularizes the value function by adding a term to the objective function that measures the difference between the value function and the expected future rewards. This helps to keep the value function from deviating too far from the true value function, which can also help stabilize the learning process.

2.4 ADAM optimizer

The ADAM optimizer is a popular optimization algorithm used to train neural networks. It is a variant of the gradient descent algorithm, which is the most common optimization algorithm used to train neural networks. Like gradient descent, ADAM updates the parameters of the neural network by computing the gradients of the loss function with respect to the parameters and moving in the opposite direction. However, ADAM incorporates two additional features that make it more efficient and effective than vanilla gradient descent.

The first feature is the use of an adaptive learning rate. In vanilla gradient descent, the learning rate is a fixed hyperparameter that controls the step size of the updates. In ADAM, the learning rate is adapted on a per-parameter basis, which means that the step size of the updates is different for each parameter. The learning rate is adapted based on the historical gradient information of each parameter. This allows ADAM to take larger steps for parameters that are changing slowly and smaller steps for parameters that are changing quickly, which helps to converge faster and avoid overshooting the optimal solution.

The second feature is the use of momentum. Momentum is a technique that helps to smooth out the updates and prevent the optimization from getting stuck in local minima. ADAM uses a technique called "Adam momentum," which combines the gradient information with the historical gradient information of each parameter. This helps to smooth out the updates and prevent the optimization from getting stuck in local minima.

3 Understanding the Tools

3.1 The simulation

3.1.1 The environment

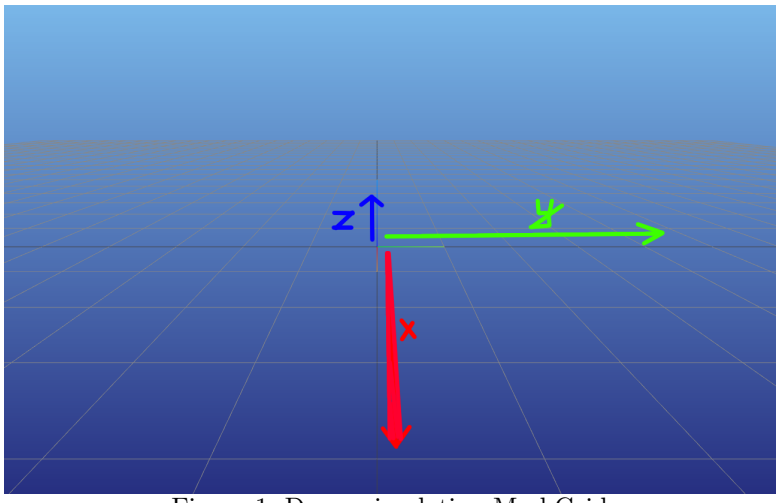


Figure 1: Drone simulation MeshGrid

The simulation is a MeshGrid which uses a 3D coordinate system.
The drone is a yellow drone simulated in the coordinate system.

3.1.2 The Drone

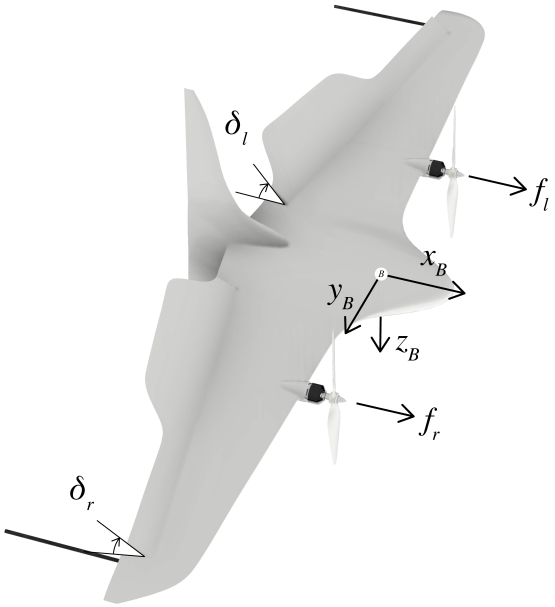


Figure 2: The Drone used in the simulation

The drone can execute up to 4 different actions in its action space. Both propellers can be accelerated with different speeds to exert different forces f_i and both flaps can have different angles δ_i.

$$\textbf{actions} = \begin{pmatrix} f_l \\ f_r \\ \delta_l \\ \delta_r \end{pmatrix}$$

$\Longrightarrow$ 4 possible actions

The drone has a neural networks and learns to apply actions according to the rewards received for applying said actions. That way, the drone learn to fly the desirable patterns.

3.2 The template

3.2.1 Reinforcement learning continuous

A sample code is provided for supporting the coding part of the project, in which the drone is taught to fly as high as possible. The implemented reward function for that purpose rewards the drone for staying alive, being oriented upwards and having an upwards velocity. Because it isn't necessary for the drone to maneuver in 3D, in order to reach big heights, the action space is reduced from 4 values to 2. The propeller speeds get set to the same values as each other and so do the flaps. That way, the drone needs only to learn to accelerate its propellers without paying attention to the different propeller speeds and to deflect both of its flaps acting as an aeronautic elevator.

3.2.2 Trajectory progress

The code is using the Flyonic module, which provides functions for generating a trajectory for the drone to follow, calculating progress along that trajectory, and visualizing the trajectory in a 3D environment using MeshCat. The script starts by generating a trajectory with the generate-trajectory function and then plots the x, y coordinates of the trajectory points. It then calculates the progress of a point along the trajectory and plots it on the same graph. It uses the MeshCat visualization library to create and display a 3D visualization of the trajectory and points.

3.2.3 Visualization

The module called Visualization exports functions that are used to create and manipulate a 3D visualizer using the MeshCat package. The functions can create and open a new visualizer, create and open a new visualizer but with a remote IP, select an object by its name and transforms it by position x and rotation matrix R, visualize actuator values such as flaps and motors, create a VTOL object with the specified name, create a sphere object with the specified name and radius, which is supposed to represent the waypoints, visualize a set of waypoints, and closes the MeshCat visualization. It also includes some other function like visualization of the arrow.

3.2.4 Utility tools

This julia code defines a module called Utils, that contains a set of utility functions for calculating angles, progress along lines, and generating trajectories. The functions can return the angle between two input vectors, return the progress of a given vector along a line between two points, return a line segment and progress of a given vector along a set of waypoints, return a set of waypoints for a trajectory, test if a given matrix is a valid rotation matrix, project a given matrix onto the special orthogonal group SO(3) and return the result.

In the function calculate-progress, the gates are of type **Vector{Vector{T}}**, which is the same as the waypoints of the VTOL. The waypoints of the VTOL act as the gates, necessary as the arguments in the calculate-progress function. That's why it's important that the types be the same.

4 Building and Training the Model

4.1 Steer to point in 2D

Teaching the drone how to fly a 3D-path along a trajectory with many waypoints is a big project. It requires a lot of time and consideration to plan and train a drone for that purpose. Therefore the problem is reduced to simply training the drone to fly to a single 2D-point. Like in the sample code, the action space was reduced to 2 variables. Because the drone only flew in the 2D-space, the flaps were unnecessary and so the only 2 executable

action were f_i changing the forces the propellers exerted. Given the specific orientation of the drone, that made it fly in the xz-plane.

After completing the phase of teaching the drone to fly to a single point in a 2D-plane, other solution focused on training the drone further to be able to fly a trajectory of many waypoints on a 2D-plane. The focus of this project was a little different. Because the paper for the project is called *"Learning Minimum-Time Flight in Cluttered Environments"* the optimization focused on teaching the drone to reach the point in the 2D-plane as quickly as possible. The hazards of the environment included the gravity which made the drone fall down instantly after initialization, had the drone not switched on its propellers and a missing limit on the propeller speed differences, which resulted in rapid, simulation-ending rotation speeds.

4.2 The standard reward function

The standard reward function as suggested in the paper to the project is given by equation (1)

$$\underbrace{r_{\text{paper}}(t)}_{\text{reward}} = \underbrace{k_p r_p(t)}_{\text{partial progress}} + \underbrace{k_s s(p(t))}_{\text{total progress}} + \underbrace{k_{wp} r_{wp}}_{\text{waypoint proximity}} - \underbrace{r_T}_{\text{collision toll}} - \underbrace{k_\omega |\omega|}_{\text{rotation speed toll}} \tag{1}$$

However, using that standard reward function did not achieve the desired results. In order to teach the drone as quickly as possible to fly to the point as quickly as possible, the reward function had to be adjusted. There were many adjustment attempt along the way and explaining all of them would get very complicated very quickly. However, there were a lot of tendencies of the drone's behavioral patterns, depending on how the function was adjusted. Missing punishments to the drone for deviating to far prom the flight path or the waypoint, as fall as having a low z-coordinate often led to the drone falling down very often before learning to keep itself up. Local maxima in the reward function from ending the simulation resulted in the drone wanting to accelerate its actuators as quickly as possible to end the simulation so the punishment didn't grow too big.

During a first successfull implementation of the reward function there were 4 learning phases of the drone.

1. The drone had to learn to maintain a positive altitude $z \geq 0$. This however often resulted in rapid turns, aimed to end simulation before the punishments for low z coordinate got too big.
2. Stopping from constantly doing rapid turns $|\omega| \leq 2.0$ finally lead to some rational drone behavior. At first only small and poor corrections for path deviations were possible.
3. Once the drone was slowly learning to approach the target point, the simulation need a lot of time to learn to actually get within its proximity range. It was a slow learning process with strong erroneous behavior.
4. When the drone was able to reach the 2D-point it was time to optimize the flight time. That way the drone would learn to reach the target faster and faster. This also led to the drone frequently overshooting and missing the target point.

4.3 Improving the reward function

When adjusting the reward function, certain changes bore no fruit while others were really successful. The optimization attempts that either had a negative effect on the drone's behavior or none at all were f.e.:

- Badly conditioned changing of the coefficients k_i resulted in unexpected and unpredictable flight behavior.
- Extreme scaling of negative reward values or punishments didn't change the drone's behavior at all. The drone seemed not to notice the difference between punishments as high as -60000 or as low as -2. As long as they were negative, the drone tried to change them randomly and the reward kept swinging between different values for negative rewards.

- Limiting the propeller differences and therefore the rotational acceleration didn't have any effect and the drone's progress. This was probably largely due to the fact, that just because the actor didn't allow itself, to have high propeller speed differences, the critic didn't know that and still tried to teach it according to the values in the action space.

The successful optimization attempts were:

1. Resizing the network structure seemed to have made the drone slightly smarter. This seemingly accelerated the learning progress, though the reward function still needed to be condition so the drone could learn the right flight patterns.
2. Cautiously considering the orders of magnitude of the summands of the reward function turned out to be vitally important. If certain parameters were too high, they overwhelmed all the other and the drone concentrated on satisfying them and disregarded everything else. On the other hand, if they were too low, the drone didn't care enough about them to change them. Perfecting the parameters to all be in the same order of magnitude resulted in omnifunctional improvements.
3. Positive rewards worked better than punishments. An example of that, was that teaching the drone, it would get rewarded for keeping $z > 0$ had a much faster and better learning effect than teaching the drone it would get punished for $z < 0$.
4. Optimizing summands, which were supposed to be kept low using the exponential function worked better than using an anti-proportional function. I order to teach the drone to have as small a flight time as possible, it was seemingly better to apply the factor $f(t) = e^{-t}$ rather than $f(t) = \frac{1}{1+t}$

The improved reward function which ultimately worked vary well for a single 2D point is given by equation (2)

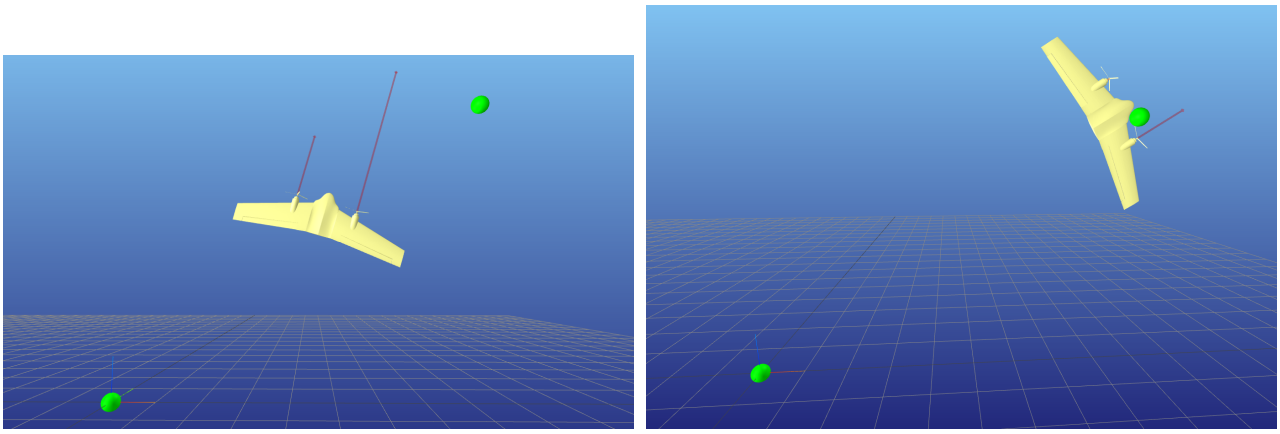
$$\underbrace{r_{\text{new}}(t)}_{\text{new eeward}} = \underbrace{k_p r_p(t) + k_s s(p(t)) + k_{wp} r_{wp} - k_\omega |\omega|}_{\text{standard reward without collision toll}} + \underbrace{r_{\text{extra}}(t)}_{\text{additional optimizations}} \tag{2}$$

$$\underbrace{r_{\text{extra}}(t)}_{\text{additional optimizations}} = \underbrace{k_{su} r_{su}}_{z \geq 0} + \underbrace{k_{pr} r_{pr}}_{|\dot{\omega}| \text{ small}} + \underbrace{k_{hit} r_{hit}(t)}_{\text{waypoint hit}} - \underbrace{k_d d_{wp}}_{\text{distance from waypoint}} - \underbrace{k_{gd} r_{gd}}_{\text{distance from guiding path}} \tag{3}$$

5 Evaluation of Results

5.1 Waypoint in close proximity

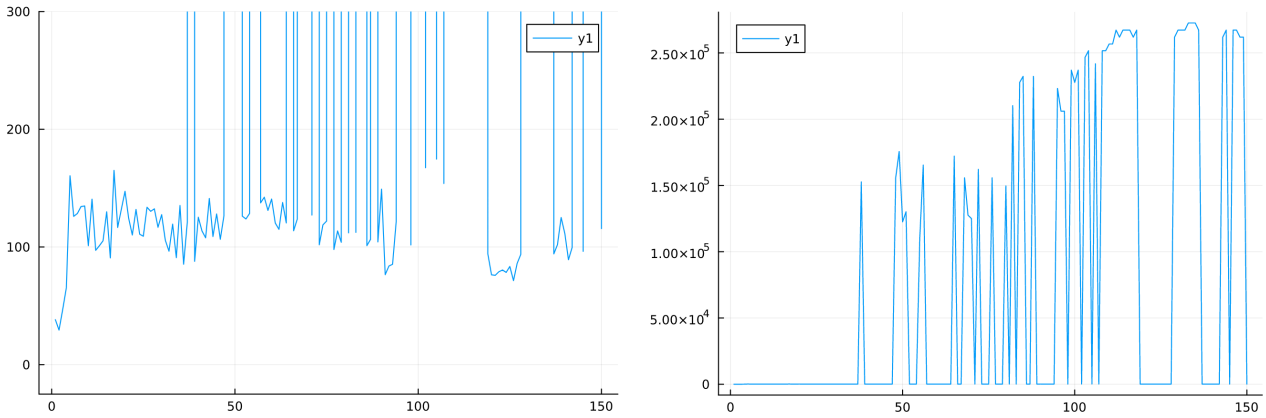
Thanks to the better implemented reward function the drone was able to learn to fly to a 2D point very quickly.



(a) Drone to close waypoint

(b) Drone is at close waypoint

The graphs in figure 4 show the reward function on the y-axis plotted against the epoch on the x-axis for the drone learning to fly to a point in close proximity to the origin.



(a) Learning curve #1 scaled in the learning phase

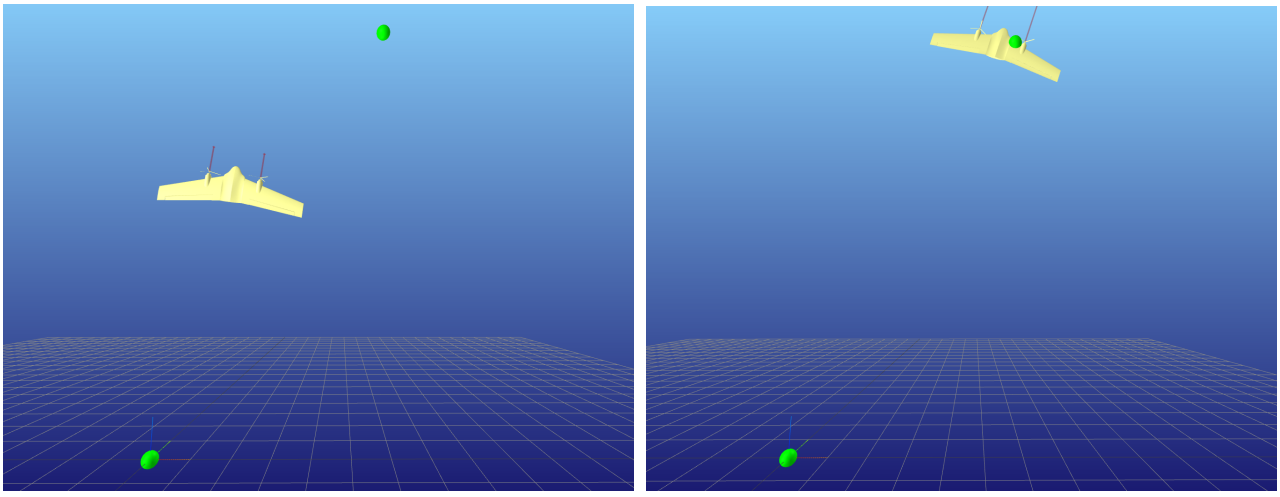
(b) Learning curve #1 scaled in the optimizing phase

Figure 4: Learning curves for a close waypoint

The interesting this about this is, that the improved reward function worked so well, that, the during the best iteration the drone actually managed to fly to the waypoint and lightly scratch it with one of its wings after only the 70000^{th} epoch. Similar behavior is shown in figure 4a, where the reward function increases from roughly 30 to over 150 withing the first 90000 training epochs or so. After about epoch 380000 the drone made its first hit on the target. It then began learning to optimize its flight time and received higher and higher rewards for better and better timing as seen in figure 4b.

5.2 Waypoint in far proximity

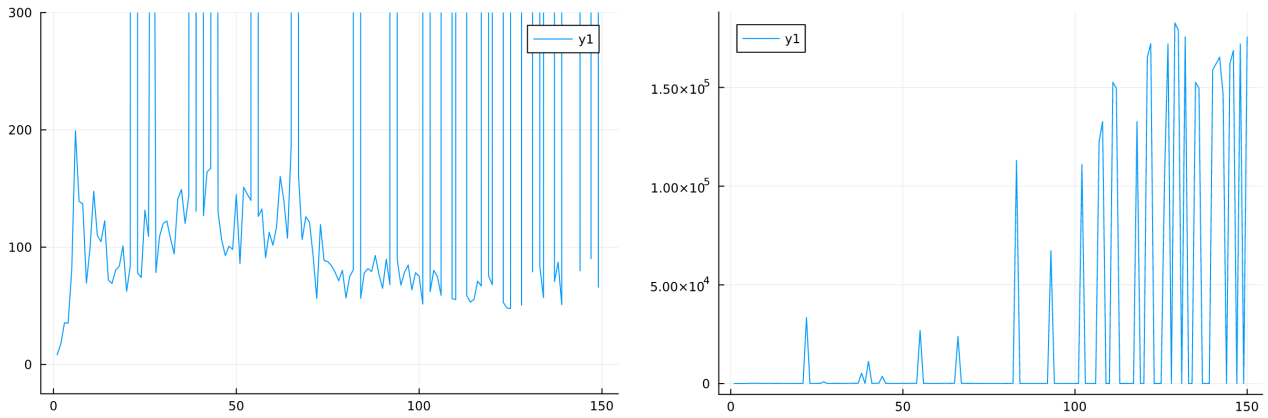
For an unknown reason, it was much harder for the drone to learn to fly to a point that was further away. It was as if there was a threshold for how far a point could be for the drone to learn to fly there.



(a) Drone flying to a far waypoint

(b) Drone hitting far waypoint

The graphs in figure 6 show the reward function on the y-axis plotted against the epoch on the x-axis for the drone learning to fly to a point further away from the origin.



(a) Learning curve #1 scaled in the learning phase

(b) Learning curve #1 scaled in the optimizing phase

Figure 6: Learning curves for a close waypoint

This time the drone needed slightly more epochs to learn to fly in the direction of the waypoint. The figure 6b. also shows that even after learning to hit the waypoint, the dron estruggled much more to improve the flight time in comparison to a waypoint in close proximity to the origin.

5.3 eccentric behavioral patterns

The first eccentric behavior happened during the learning phase. Because of the reduced reward for high rotational acceleration, the drone learned not to rotate itself out of the simulation. However, it still had the instinct to rotate as much as it could for corrections towards the guiding path. Because mid-rotation it noticed that it was getting punished for it, it often fired its propellers in the opposite style. This lead to a weird behavioral pattern that looked like rocking. Sometimes the drone was rocking so hard, it accidentally left the bounds of the guiding path and rapidly ended the simulation.

The other thing that happened was in the speed optimization phase. It isn't clear why, but eventually the drone learned to try to hit the waypoint from the side rather than had on. The drone wasn't flying straight from the origin to the waypoint, but was swinging an arc around the guiding path and hit the waypoint sideways. Interestingly enough that did not impede the results of the reward function nor the speed and the flight time much.

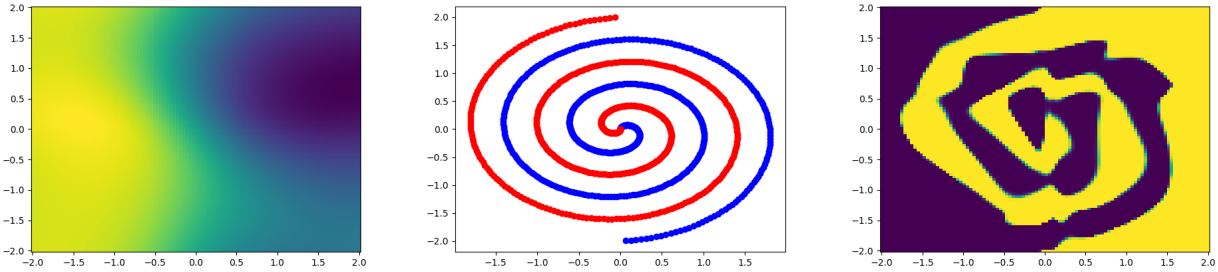
5.4 Findings

During the course of the study, some fascinating findings were made. The precise adjustments of rewards and punishments, taking into account all orders of magnitude, was vitally necessary. If they weren't made, that might have resulted in unpredictable behavior. The learning progress was very strongly dependent on the reward. It was hard to do but perfecting the cooperation of the reward's summands in order to achieve the best conditioning of the drone's neural network was crucial was therefore very effective.

The project itself was very fascinating and incredibly interesting. In truth, I didn't really like the concept of reinforcement learning all that much in comparison to conventional training. However, this is just my opinion. Other than my reduced fondness of reinforcement learning, the project itself fittedf nearly perfectly to my future goals. I have been fascinated with VTOL, especially eVTOL technologies for a while and can very well imagine one day working at a company like Lilium of Velocopter. This project was the perfect choice and putting as much effort into it as I did was definitely worth it.

It is also noteworthy that the entire course on neural networks was very interesting as well. I even extended the work on exercise sheet number 6 to teach the 2D-point-mapper to map the points onto 2 spirals integrated

into each other. My friend dubbed it **"Leo's stupid spiral-AI"**



(a) Output before the training (b) Training data (c) Output after the training

Figure 7: The spiral-AI

It was easier top teach the neural network to learn using training data rather than reinforcement learning, which is why I liked conventional training better. However, I can see that reinforcement learning has some advantages. It would be extremely confusing and complex to come up with training data to teach a drone to fly trajectories. How would one even collect the training data? Reinforcement training might just be the solution for AI-problems where coming up with training data is extremely hard. It was one of the best if not the best modules to do this semester.

References

[1] Robert Penicka et al. "Learning Minimum-Time Flight in Cluttered Environments". In: *IEEE Robotics and Automation Letters* 7.3 (July 2022). Conference Name: IEEE Robotics and Automation Letters, pp. 7209–7216. ISSN: 2377-3766. DOI: 10.1109/LRA.2022.3181755.

[2] *LarissaRickler/nn_project_drone*. URL: https://github.com/LarissaRickler/nn_project_drone (visited on 01/22/2023).