Delft University of Technology

Bio-Inspired Intelligence and Learning for Aerospace Applications
AE4350

Training a 2D Drone Model to Fly Using Reinforcement Learning

Author:

Matan Neumark 5920078

Delft University of Technology

August 31, 2024

Contents

Nomenclature

α	Angle relative to the vertical axis, positive counterclockwise
α	Learning rate
$\ddot{\alpha}$	Angular acceleration
$\ddot{x}$	Acceleration along the horizontal axis
$\ddot{z}$	Acceleration along the vertical axis
$\dot{\alpha}$	Angular velocity
$\dot{x}$	Velocity along the horizontal axis
$\dot{z}$	Velocity along the vertical axis
γ	Discount factor
μ	Mean value
π	Policy
σ	Standard deviation
θ	Policy weight vector
A	Action
A_t	Action at time t
G	Discounted sum of returns
I_{yy}	Moment of inertia due to rotation about the center of gravity in the XZ plane
m	Mass
R	Reward value
R_t	Reward at time t
S	State
S_t	State at time t
T_l	Thrust of Left motor
T_r	Thrust of right motor
x	Position is space along the horizontal axis
z	Position is space along the vertical axis
a	An action
r	A reward
s	A state
S'	New state

Chapter 1

Introduction

Up until a few years ago, watching a pilot fly a competitive racing drone or an aerobatic model helicopter would have led one to believe these pilots hold superhuman skills because of the precision and coordination they exhibit when performing complex and rapid manoeuvres. However, in recent times, researchers have been able to use artificial intelligence (AI) to train models to not only meet the performance of professional human pilots but to far, far exceed them as can be seen in publications by the University of Zurich and ETH Zurich [1], [2]. Inspired by how autonomous drones were able to fly around a gate racing track as if they were on rails, the chosen task to tackle in this assignment is that of a two-dimensional drone model which is trained to take off and fly into a target zone. Most commonly, the branch of AI used to tackle challenges of this nature is Reinforcement Learning (RL). RL is different from, for example, supervised learning because it generates its own 'training data' through interaction with the environment rather than being trained using external, pre-labelled data. While advancements in AI and RL in particular are becoming more and more evident in our daily lives, these technologies have roots dating back to the 1950s [3]. The process by which an agent interacts with the environment is shown in Figure 1.1. The idea is that an 'agent' takes an action in an environment and in return it gets a new state and a reward indicating whether the state is desired or not. Then, all kinds of algorithms can be used to assign values to states which would help the agent 'choose' the best action when he revisits this state in the future. The Environment can be represented by a set of rules. These can be based on physics and used to model a system such as a drone, or they can be made to dictate the dynamics of a game or describe a space such as a maze.

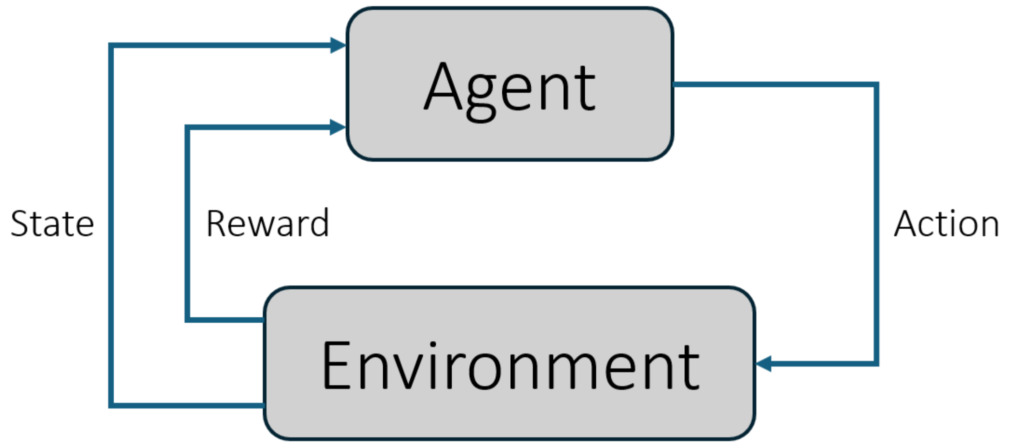


Figure 1.1: Reinforcement Learning system

Drones can of course fly without any form of AI. Traditional PID controllers do an outstanding job in stabilizing drones. They allow a human pilot to easily control an otherwise unstable system, and enable autonomous drones to precisely follow way-points and hold position (together with other systems such as GPS and RTK to estimate the position error). Yet, RL can be used to perform tasks that other control methods are not designed to do [4], and the almost magical way in which performance is improved over time makes it a fascinating field to explore and learn about, which is why the idea of letting a drone 'learn' to fly is intriguing and was chosen to be developed in this assignment.

Chapter 2

Simulation Environment

2.1 Drone and Environment Model

With a simple free-body diagram of a Quadcopter, its equations of motion in 2D can be derived. They are given by the equations below for motion in the XZ plane of an inertial frame of reference as defined in Figure 2.1. The angle α, representing the drone's in-plane roll angle, is defined with respect to the Z axis. The drone's moment of inertia I_{yy} is calculated by assuming a spherical mass distribution. This assumption is made (rather than assuming it to be a disc) because it allows the model to be easily extended to 3D without changing the inertia equation for yawing motion. The custom environment is based on the architecture of OpenAI's Gym environments such as the Lunar Lander and Pole Cart [5].

$$\ddot{x} = \frac{1}{m}\sin(-\alpha)(T_l + T_r) \tag{2.1}$$

$$\ddot{z} = -g + \frac{1}{m}\cos(\alpha)(T_l + T_r) \tag{2.2}$$

$$\ddot{\alpha} = \frac{r}{I_{yy}}(T_r - T_l) \tag{2.3}$$

$$I_{yy} = \frac{2}{5}mr^2 \tag{2.4}$$

The environment has the following characteristics:

1. Instantiating the environment requires the drone's mass, the diagonal motor distance, and render mode as inputs. Rendering is done with Pygame and is useful for verifying the trained and untrained models but it slows down the training considerably.

2. The environment can be reset with the drone starting from the 'floor' where its altitude and all of its velocity state components are zero while x and α are random and in the range 0 ± 0.01. Alternatively, it can be spawned mid-air with all random state values in the range mentioned above.

3. An action can be passed through to the environment which returns a new state, reward, a termination or truncation signal, and the cause of termination.

4. The $1x6$ state space is continuous and its components and corresponding ranges are specified in Table 2.1. The $1x2$ action space is also continuous and it represents the thrust of the left and right motors respectively.

5. The simulation truncates when it reaches 300 steps.

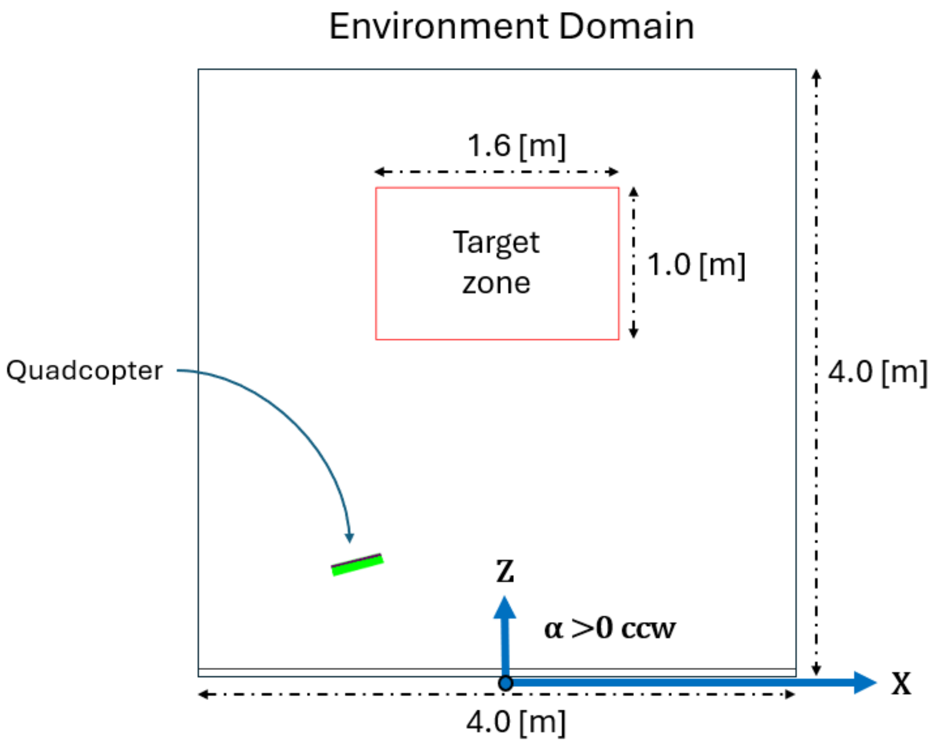


Figure 2.1: The agent (drone) and its training environment

$S_t[i]$	Domain bounds	Target zone bounds	unit
x	$[-2, 2]$	$[-0.8, 0.8]$	$[m]$
$\dot{x}$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[m/s]$
z	$[0, 4]$	$[2.2, 3.2]$	$[m]$
$\dot{z}$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[m/s]$
α	$[-60, 60]$	$[-60, 60]$	$[deg]$
$\dot{\alpha}$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[deg/s]$

Table 2.1: Definition of the continuous state space variables and bounds

2.2 Reward Definition

Several reward systems were investigated in pursuit of the one that encourages the desired behaviour most effectively. Eventually, the chosen reward function gives the agent a reward of 1 for each time step that hasn't terminated the episode, a reward of 10 for every time step the agent is within the target zone, and a reward of 0 when the episode terminates due to the agent exceeding its bounds. In an initial version of the reward function, the agent was given a reward only when it flew inside the target zone. However, the agent was never able to reach it due to the large amount of time steps required, meaning that during exploration the probability of it succeeding to fly straight up without first exceeding the angle limits or hitting the domain boundaries was virtually non-existent. Moreover, it has been observed that there had to be a substantial difference in the reward between the target zone and the rest of the domain. Otherwise, the policy parameters wouldn't be 'shaped' to manoeuvre the agent there within a reasonable training period.

Chapter 3

Method

Deciding which RL algorithm to choose is no easy task considering the vast amount of different algorithms and variations that exist. They all have their strengths and weaknesses so choosing the right one requires consideration of the characteristics of the problem at hand. As described in the previous chapter, both the state and action spaces are continuous, which excludes algorithms such as SARSA and Q-learning. Having first tried to use Q-learning to train the drone, this method was indeed not suited for this task. Q-learning can be used with continuous state space if the states are aggregated such that discrete state values can be stored in an array. But in this case, the state space has dimension $\mathbb{R}^6$, which, if the states are to be discretized into sufficiently small groups, would result in an enormous, multi-dimensional array. Beyond the excessive memory required to store such a large array, the probability of revisiting a state aggregation enough times to successfully learn the policy becomes extremely small, causing the learning rate to be intolerably slow. Figure 3.1 shows an overview of RL algorithm classification. For the problem dealt with here, the bottom branch contains the relevant algorithms. Ultimately, REINFORCE is chosen because the algorithm answers the requirements and because it is well documented and much research has been done with it in literature.

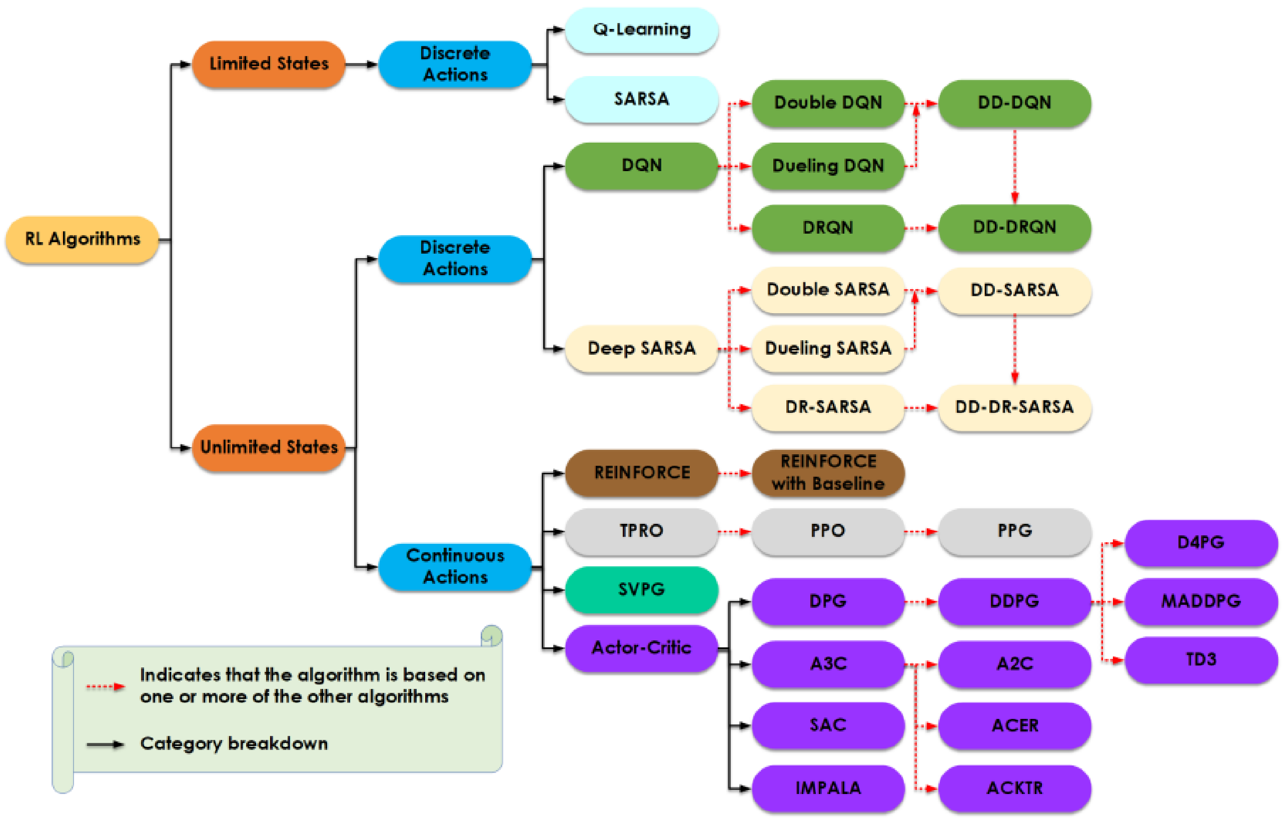
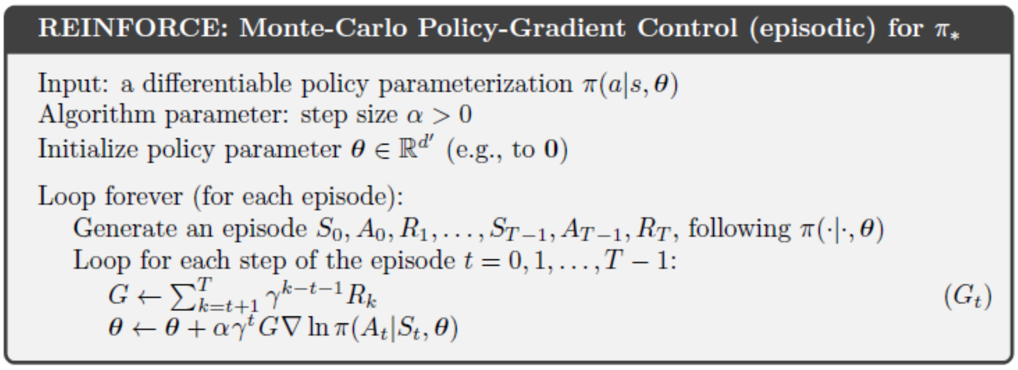


Figure 3.1: Classification of RL algorithms based on environment characteristics. Taken from [6]

3.1 The REINFORCE Algorithm

The REINFORCE algorithm is a model-free, episodic, policy gradient method [3]. In policy gradient methods actions are chosen through a parameterized policy instead of using action values, which is the case for action value methods. Figure 3.2 shows the REINFORCE algorithm as presented in the book [3]. However, it isn't easy to interpret and not particularly useful if one wishes to implement it in code. Thus, the following will describe the implementation and relate it to the algorithm provided by Sutton and Barto. The code was developed with the aid of various documentation repositories, including those of OpenAI [5] and PyTorch [7].



```
REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

Figure 3.2: The REINFORCE RL algorithm. Taken from [3]

3.2 Implementation

As mentioned above, REINFORCE is episodic, meaning that learning only takes place at the end of an episode. At the beginning of an episode, the environment is reset with an initial stochastic state S_0. This state (and every consequent state) is fed into a neural network which outputs a mean μ and standard deviation σ. These parameters are used to create a normal distribution from which an action a is sampled. This action, which is still normalized, is multiplied by the weight of the drone to give thrust. The thrust is then fed into the environment which returns a new state s and reward r (and a truncation signal). The reward is appended for later use, and the cycle repeats as shown in Equation 3.2. Figure 3.3 visualizes the overall structure of the implemented algorithm and information flow between classes. The environment is blank to highlight that it is invisible to the agent who can only interact with it through the exchange of actions, states and rewards.

Once the episode is done learning can begin. Because neural nets are used, they are the object that requires training, leading to a slight deviation from the formal definition shown in Figure 3.2. First the discounted sum of rewards G is computed. The discount factor parameter γ influences the relative importance of future rewards. Its effect on performance will be evaluated later. After G is computed, it is multiplied by the gradient of the natural logarithm of the policy evaluation when taking action A_t at state S_t and policy parameterization θ. In other words, evaluating the policy $\pi(A_t|S_t, \theta)$ returns the natural logarithm of probability $P \in [0, 1]$ of taking action A_t at a given state S_t and parameters θ Equation 3.1. Now the term $G\nabla\pi(A_t|S_t, \theta)$ can be calculated by taking the natural logarithm of the previously found probabilities, multiplying them by $-G$ and numerically calculating the gradient using `torch.tensor.backward()` command. The reason for taking the negative of G is that the neural net's optimizer uses gradient descent whereas the equation in the algorithm calculates gradient ascent. The policy update cycle finishes with the optimizer taking a step in the direction of the (negative) policy gradient. The size of the step is determined by α, often referred to as the learning rate. Its magnitude influences the optimizer's convergence speed and ability to find the true minimum [8].

$$\pi(A_t|S_t, \theta) = P\{A_t|S_t, \theta\} \tag{3.1}$$

$$S_1 \;\rightarrow\; A_1(\mu, \sigma) \;\rightarrow\; S_2,\; R_2 \ldots \rightarrow\; A_t(\mu, \sigma) \;\rightarrow\; S_{t+1}\; R_{t+1} \;\ldots \rightarrow\; A_{t+1}(\mu, \sigma) \;\rightarrow\; S_T\; R_T \tag{3.2}$$

3.2.1 Neural Nets

A policy maps a state to an action by following predefined rules. Greedy policies for example always choose the action that yields the highest reward at a given state, even if it is less beneficial in the long run (such as in the mountain car problem where an agent must first perform actions that yield less rewards in order to maximize future rewards). In tubular methods, a state value function can be used as a look-up table from which an action is chosen based on the type of policy (Greedy, random, ϵ-greedy). However, in continuous state and action spaces like those dealt with here, the policy must be embodied in a different way. This can be achieved by making use of artificial neural networks which learn to vary the weights of each of the neuron's inputs such that eventually they can map inputs to the desired outputs. The design and architecture of neural nets require consideration of many variables and is a research topic of its own. These include the number of hidden layers, the number of neurons in each layer, the type of activation functions and the type of optimizer. Since the focus of the assignment is not on the net itself, but rather on Reinforcement Learning, a basic net configuration was chosen after reviewing documentation provided by OpenAI and Pytorch [5], [7]. As explained in the last subsection, an action is sampled from a normal distribution, which in turn requires the values of the mean and standard deviation. To approximate those, three neural nets are employed:

1. **Shared net:** The first net takes a state as an input and outputs the weights of a certain number of neurons. This net has two hidden layers, each with 32 neurons and tangent hyperbolic activation functions Equation 3.3.

2. **Mean value net:** The second net takes the output of the shared net as input, processes it through its single layer and outputs a mean value. The challenge with this net is that the mean value should not be smaller than zero because that would mean that the drone's motors are producing negative thrust. While this could be physically possible for a variable pitch quadcopter for example, this isn't what is assumed here, so $\mu => 0$ is a requirement. While this is easy to achieve by passing the layer's output through an activation function such as ReLU Equation 3.5, this wouldn't solve the next problem which is that the net's output is only the mean of the normal distribution, so the sampled action may still take on a negative value even if the mean is larger than zero. Despite many attempts to solve the issue without 'killing' the learning process, no appropriate solution was found. During one attempt, the action value was set to zero if it was negative, but it seemed to throw off the loss function and sabotage learning progression. Another possibility which is further elaborated on later, is that the issue isn't related to the net, but rather to the reward function.

3. **Standard deviation net:** Lastly, the third net also takes in the output of the shared net as input and outputs the standard deviation σ. Because the standard deviation must be larger than zero, the net's output goes through a Softplus activation function Equation 3.4.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (3.3) \qquad f(x) = \ln(1 + e^x) \quad (3.4) \qquad f(x) = 0 \;\; if \; x \leq 0, \; else \; x \quad (3.5)$$

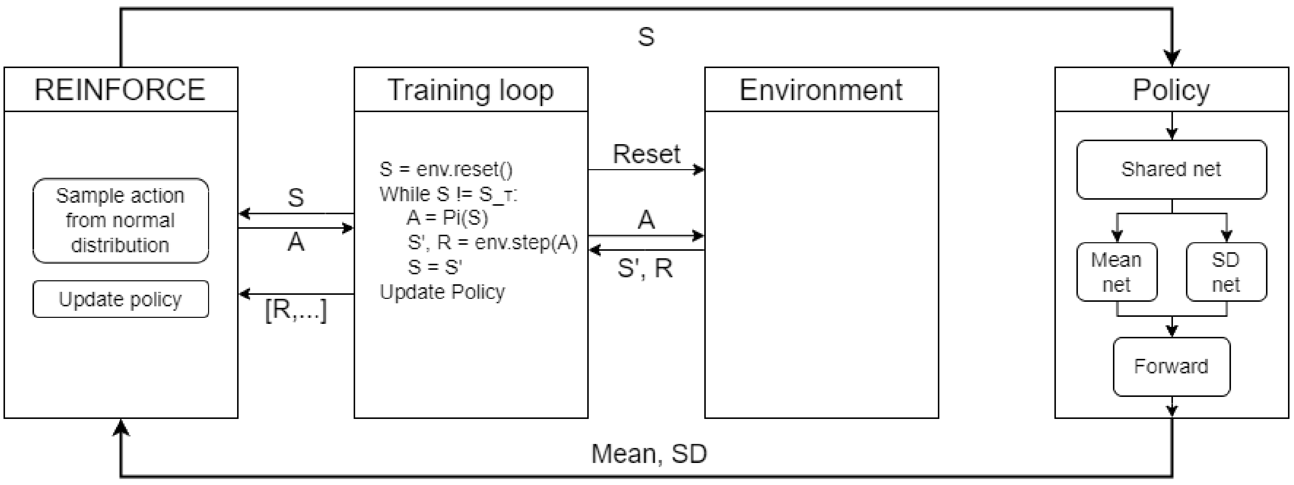


Figure 3.3: Diagram of information flow

Chapter 4

Results

4.1 Influence of Hyperparameters on Performance

The learning rate α and the discount factor γ are varied independently to study the influence of the hyperparameter's value on the agent's performance. Due to the stochastic nature of RL, results can have large variations between runs and therefore paint an entirely different picture each time. Moreover, the sample variance can also vary greatly within each run resulting in a very noisy trend. To smooth out the learning curves, a running average is computed with a window size of 100 episodes. Then, to reduce uncertainty between runs, each hyperparameter value is processed 4 times. Finally, the per-episode mean of all runs is calculated to produce the final curve. Ideally, the number of runs would have been greater, but given the amount of time training takes, a balance had to be met between mitigating uncertainties and exploring enough hyperparameter values. In total, processing all runs took about 10 hours. Note that the maximum theoretical return per episode is just under 3000 (episode length is 300 time steps and the reward in the target zone is 10 per step).

4.1.1 Learning Rate α

The learning rate α works differently in different algorithms. Here, it is used as a step size parameter in the optimizer's search of a minima. A larger step size means that fewer iterations are required to descend a certain amount, shortening the learning period. But, a too large of a step size will prevent the optimizer from finding the true minima thus reducing the maximum obtainable performance and leading to instabilities. Too small of a step size will extend the learning period and is inefficient. This is illustrated in Figure 4.1. Figure 4.2 shows the learning curves for different step size values. Figure 4.2a clearly shows that larger step sizes indeed increase the learning rate but come at the cost of large oscillations. After 5000 episodes, both the green and blue curves in Figure 4.2a demonstrate that the agent was able to last for the entire duration of an episode (300 steps), but in Figure 4.2b the blue curve shows further improvement in the remaining 1000 episodes, whereas the green curve oscillates around the same value it had before. Performance could be increased by using an adaptive learning rate which changes its value based on the gradient, or another predefined function for example [8]. This would improve performance by allowing fast learning at the start, and a tamed search for the minima once most of the learning has been done.

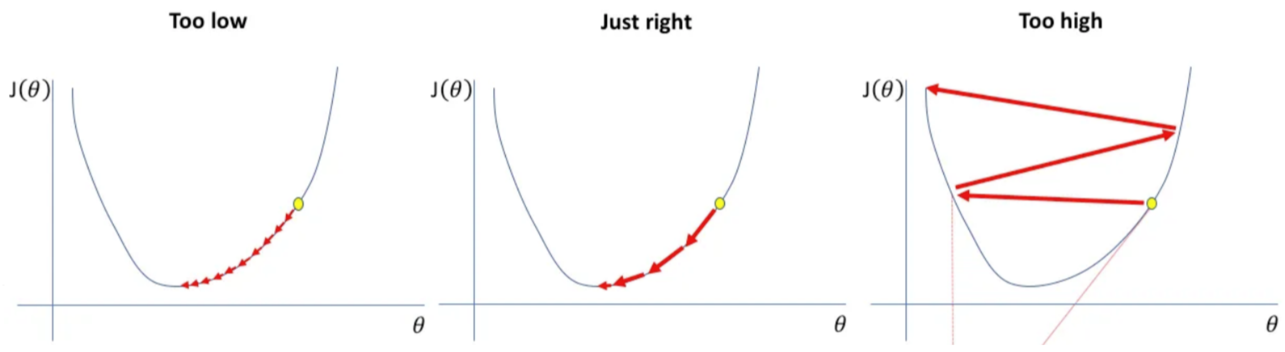
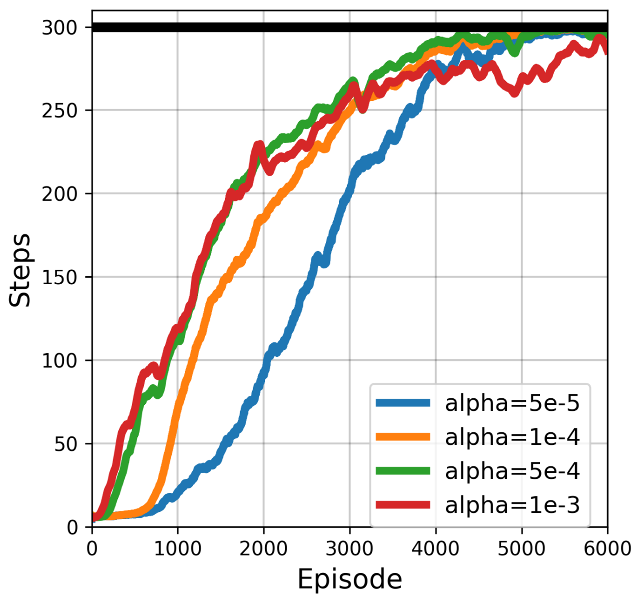
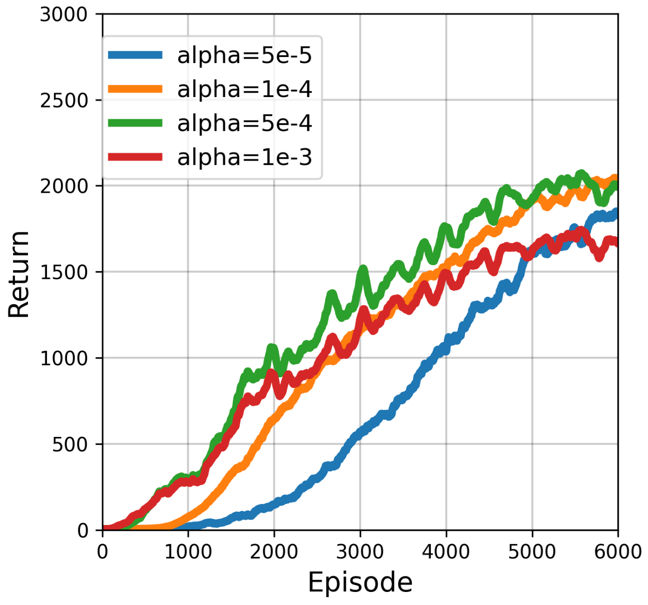


Figure 4.1: Illustration of gradient descent and the influence of step size α. Taken from [9]



(a) Steps per episode for various learning rate values



(b) Returns per episode for various learning rate values

Figure 4.2: Influence of the learning rate hyperparameter α on performance. $\gamma = 0.99$ and is constant

4.1.2 Discount factor γ

As Sutton and Barto put it; "*The discount rate determines the present value of future rewards*" [3, p.55]. What this means is that if γ has a small value (where $\gamma \in [0 : 1]$), the agent will tend to take actions which yield the maximum immediate rewards, rather than actions that bring higher future rewards and lead to higher overall returns. An agent following a policy which is updated with a low discount factor can be described as being shortsighted, and one following a policy with a high discount factor, farsighted [3]. This way of looking at it gives a nice intuition for the factor's influence and is visualized in Figure 4.3. The figure shows how the value of the factor decreases as a function of the number of steps according to an exponential decay, $y(x) = \gamma^x$. For $\gamma = 0.9$, the agent doesn't see the future beyond 50 steps while for $\gamma > 0.99$ the entire episode is within the agent's horizon.

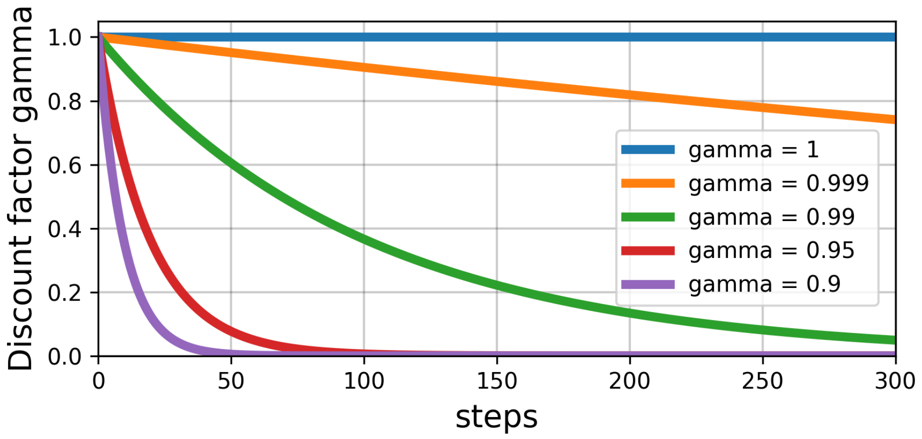
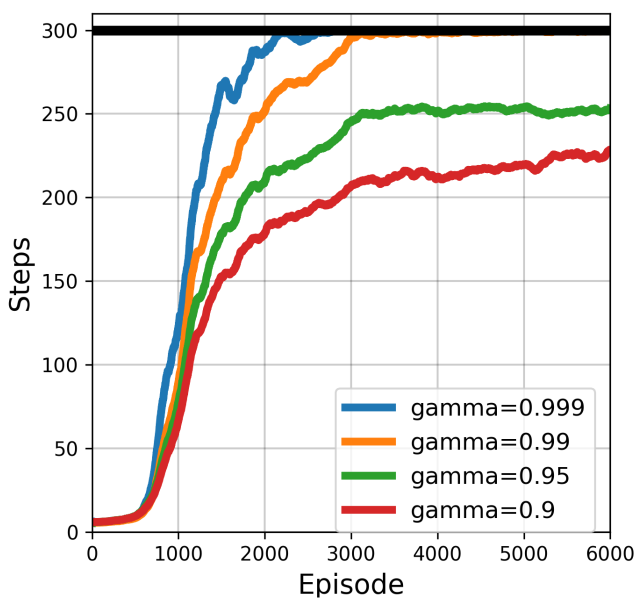
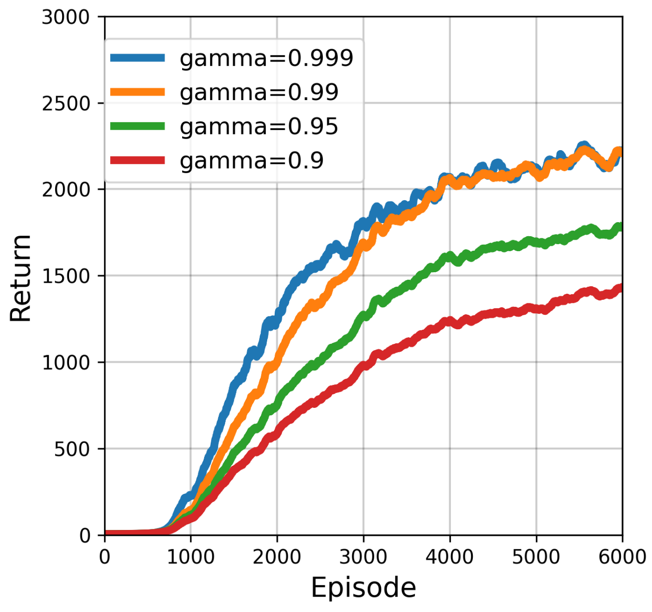


Figure 4.3: Illustration of the influence of γ on the value of future rewards

When looking at Figure 4.4, it seems like the best performance is obtained with $\gamma \geq 0.99$. According to the blue and orange curves in Figure 4.4a the agent was able to reach the maximum number of steps after only 3000 episodes. But learning didn't stop there because as visible in Figure 4.4b, the return continues to increase past the 3000 episode mark, meaning the agent spends a larger portion of the episode inside the target zone, thus maximizing returns. However, there do seem to be some instabilities at those high γ values, so a higher value is not necessarily better. This also shows that the system is rather sensitive to variations in this parameter, much more so than it was for the learning rate.



(a) Steps per episode for various discount factor values



(b) Returns per episode for various discount factor values

Figure 4.4: Influence of the discount factor hyperparameter γ on performance. $\alpha = 1 \times 10^{-4}$ and is constant

4.2 Discussion

Every method has its strengths and weaknesses and it's important to consider them when evaluating the results and determining their validity. As shown by the results, the agent achieved impressive performance by learning to take off, fly to, and hover in the target zone. However, the maximum returns it obtained were only about 76% of the theoretical maximum. This could be due to several limitations of RL, the REINFORCE algorithm, and of the specific implementation method.

4.2.1 Limitations

1. A common challenge in RL and other fields of AI is the choice of learning parameters. As evident in the results, they have a considerable effect on performance and learning rate. Optimizing them is a challenge which requires considerable time and effort. Not only due to the number of variables to tune but also due to the time it takes to observe meaningful differences that are statistically valid.

2. A limitation which concerns the specific implementation is that similar to tabular Monte-Carlo methods, this implementation of REINFORCE doesn't learn the policy online, but only once an episode ends. Depending on a typical episode length, this can substantially slow down the learning process. Choosing the type of algorithm is of course problem dependent, but it is reasonable to think that the agent could indeed learn faster had it been trained using an online learning method.

3. Since the focus of the assignment was not as much on the neural net's architecture, there is likely room for improvement by optimizing it. Limited experimentation showed that increasing the number of neurons (from 32 to 64 per layer) had a positive effect on the amount of learning per episode but came with substantial computational cost. So this avenue could be explored further to find the best balance between the time it takes to learn versus the time it takes to optimize the cost function.

4.2.2 Unexpected Behaviour and Faulty Reward Function

Going back to the issue of negative thrust- while experimenting with different methods to prevent the agent from learning this behaviour, it was evident that the agent's performance was negatively affected when restricting it to positive thrust values. This can be explained by considering when this behaviour was used; During the vertical acceleration to the target zone, where the reward is high, the agent learnt to use negative thrust to slow down its ascent. It also seemed to use negative thrust to make quick roll motions to correct for deviations in its x position. This behaviour allowed it to accelerate up more quickly, spending less time in the 'low' reward region and more time in the precious target zone. This unexpected behaviour can be attributed to a 'faulty reward function' [10] in which the agent learns to achieve a high reward by exploiting the reward structure rather than by completing the implicit task. To disincentivize this behaviour, the reward function was modified such that if the action vector had an element with a negative value the reward would be zero and the episode would terminate. However, this modification had a detrimental effect on learning and the agent was not even able to hover. So this issue requires further investigation.

4.2.3 The Issue of Generalization

While REINFORCE, like many other RL algorithms, is model-free, a model is still required to generate states in response to the agent's actions. But even if the agent behaves optimally in the simulated environment it was trained in, it may not perform so well once deployed in a 'real' environment. For our problem, factors such as aerodynamic drag, wake vortices or the inertia of the motor's rotor are all neglected in the simulated environment. This can lead to performance issues if the policy was to be deployed in a real quadcopter. This touches on the issue of generalization, which according to A. Nichol, V. Pfau and others can be described in the following way: *"Ideally, intelligent agents would also be able to generalize between tasks, using prior experience to pick up new skills more quickly"* [11, p.1].

4.3 Conclusions

In this assignment a model of a two-dimensional drone was given the task of taking off, flying to, and hovering in a target zone placed in a bounded domain. To achieve this, REINFORCE was used, which is an on-policy, episodic Reinforcement Learning algorithm. because the state and action spaces are continuous, the policy is represented by the weights of three neural nets instead of utilizing tables and explicit state values. As shown by the results, the agent successfully learned to perform the task and was able to gain up to 76% of the maximum theoretical returns. To study the solution's dependence on the hyperparameters, several values for the learning rate α and the discount factor γ were tested and compared. The conclusion is that while there are clear trends that point to specific values as being the best, it is very much problem-dependent and choosing the right hyperparameters requires striking a balance between maximizing returns and maintaining stability. It has also been shown that the algorithm is very sensitive to variations in the discount factor, but less so to the learning rate. An interesting discovery was that the agent is capable of learning unintended behaviours. It was observed that the agent used negative thrust to perform swift accelerations, improving its ability to maintain its position and thus gain higher returns. While there are drones that can physically achieve quick thrust reversal, this is not what is assumed here, making this an undesired behaviour. Investigation of the issue suggested it might be related to the so-called 'faulty reward function' which allows the agent to behave in a way that maximises returns rather than completing the desired task. However, despite attempts to modify the reward rules to demotivate the agent from using negative thrust, a proper solution has yet to be found. So this point is worthy of further development. As for the neural network, a more thoroughly designed net could contribute to an increase in performance and or decrease in computational cost and is another point to study in the future. Although, its performance has been adequate within the scope of this RL assignment. To conclude, while the model developed here will most certainly not outperform human drone pilots, it demonstrates the power of AI and RL in particular when it comes to controlling complex dynamical systems and that this technology has many benefits, especially in situations where the system is a 'black box'.

Bibliography

[1] Robotics and Perception Group. URL https://rpg.ifi.uzh.ch/research_drone_racing.html.

[2] Ismail Geles, Leonard Bauersfeld, Angel Romero, Jiaxu Xing, and Davide Scaramuzza. Demonstrating Agile Flight from Pixels without State Estimation. URL https://youtu.be/a1MSkTD-T18.

[3] Richard S.. Sutton and Andrew G.. Barto. *Reinforcement learning : an introduction*. The MIT Press, 2020. ISBN 9780262039246.

[4] RL_for_Flight_Control - AE4350 Bio-inspired Intelligence and learning for Aerospace Applications (2023/24 Q4). URL https://brightspace.tudelft.nl/d2l/le/content/593473/viewContent/3407543/View.

[5] Gymnasium Documentation. URL https://gymnasium.farama.org/.

[6] Fadi AlMahamid, Senior Member, and Katarina Grolinger. Reinforcement Learning Algorithms: An Overview and Classification. URL https://www.ieee.org/publications/rights/copyright-.

[7] PyTorch documentation — PyTorch 2.4 documentation. URL https://pytorch.org/docs/stable/index.html.

[8] NeuralNetworkLecture - AE4350 Bio-inspired Intelligence and learning for Aerospace Applications (2023/24 Q4). URL https://brightspace.tudelft.nl/d2l/le/content/593473/viewContent/3407532/View.

[9] How to apply Gradient Descent from Scratch for any ML problem | Medium. URL https://bhatnagar91.medium.com/how-neural-networks-learn-using-gradient-descent-f48c2e4079a6.

[10] Faulty reward functions in the wild | OpenAI. URL https://openai.com/index/faulty-reward-functions/.

[11] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman Openai. Gotta Learn Fast: A New Benchmark for Generalization in RL. 2018. URL https://www.libretro.com/index.php/api.