אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

Source Seeking in Nano Drones

Matan Kolpanitzky, Omri Tzur

Department of Industrial Engineering and Management

Ben-Gurion University of the Negev

A *final project*

presented for the degree of

Bachelor of Science

Supervised by: Dr. Armin Biess

Be’er Sheva 2020

Monday 10th August, 2020

Abstract

In the last decade, the use of Unmanned Aerial Vehicles (UAV) - or drones - has increased due to the development and improvement of technology and control systems. For outdoor tasks, stable and accurate flight control and navigation can be achieved by the use of the global navigation satellite system (GNSS). When GNSS is unavailable, for example, in indoor tasks or environments with broken satellite-receiver lines-of-sight (mines, tunnels, urban canyons, canopy of trees) control and navigations of drones becomes more challenging and need to be based on other sensors, such as IMUs and vision-guided control systems (optical flow, RGB/depth cameras). Further complexity can be introduced by navigation in stochastic environments, such as flights in the presence of dynamic obstacles (other drones, walking people or birds). With the recent progress in Artificial Intelligence and learning algorithms, classical control algorithms have been augmented or replaced by data-driven methods. Reinforcement learning (RL) is the branch of artificial intelligence, in which an agent learns a task through trial-and-error by interacting with the environment. The agent receives local feedback (reward) for each action taken and its objective is to maximize cumulative reward by learning the optimal policy. In this project we apply deep reinforcement learning, i.e., the combination of RL with deep neural networks, for source-seeking in a nano-drone in a cluttered environment. For this purpose, we implement a Deep Q-Network (DQN) for drone navigation and train the drone in simulation using the Unreal engine and the AirSim simulation platform (Microsoft). After successful training, we deploy the learned policy to a real nano-drone (Crazyfly 2.0, Bitcraze) without fine-tuning and test flight performance and generalization capabilities. We find that navigating a nano-drone using DRL is feasible, even without providing visual information or GNSS. We show that the deep neural network is suitable for handling noisy sensory inputs during real flights and can generalize to new cluttered environments. In conclusion, the application of RL may have the potential to provide drones with enhanced navigation skills, eventually converting them into fully autonomous machines.

***Keywords*—** UAV - Source Seeking - DRL - DQN

Table of Contents

List of Figures

List of Tables

List of Abbreviations

UAV	Unmanned Aerial Vehicle
DQN	Deep Q Network
COTS	Commercial off-the-shelf
DRL	Deep Reinforcement Learning
IMU	Inertial Measurement Unit
GLIE	Greedy in the Limit with Infinite Exploration
ReLu	Rectified Linear Unit

1 | Introduction

1.1 Motivation

Our motivation to the project is first of all the challenge in the implementation of a deep reinforcement learning algorithm. In order to do so, we had to gain knowledge in the fields of neural networks, deep learning, and robotics, as we had to integrate between them. This project can also be useful for several use-cases, such as finding a radiation/sound source in a collapsed building or an unknown environment. By working on this project we gained experience in working with robotics and programming DRL algorithms.

1.2 Aim and Objectives

Our main aim is to develop and train a neural network model in a simulation environment, transfer it so it matches the robot's hardware and measure its performance.

- Understanding the complexity of the project
- Purchase the hardware
- Getting to know the limitations of the quadcopter and how to work with it.
- Develop a deep reinforcement learning algorithm to seek a goal
- Train the algorithm in a simulation environment
- Test the algorithm on the drone

2 | Background

2.1 Dynamic Model

In the last decades the use of Unmanned Aerial Vehicles (UAV) has increased due to the development and improvement of technology and control systems. This increase can also be justified by the fact that these aircrafts are very versatile in contrast to their lower complexity. Within UAV hardware, quadcopters are being widely used for different purposes, such as educational, commercial or entertainment. This choice can be justified by the fact that this model presents a very low moment of inertia and six degrees of freedom, which results in great stability of the quadcopter.

There are three movements that describe all possible combinations of attitude: Roll (rotation around the X axis) is obtained when the balance of rotors 2,3 and 1,4 is changed (speed increases or decreases). Pitch movement (rotation around the Y axis) is obtained when the balance of the speed of the rotors 1,2 and 3,4 is changed. Yaw (rotation about the Z axis) is obtained by a simultaneous change of speed of the pair (1,3) or (2,4) [2].

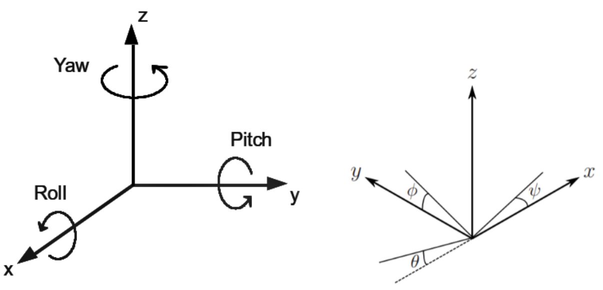


Figure 2.1: Quad-Copter orientation

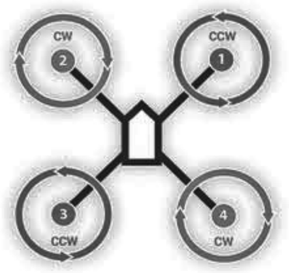


Figure 2.2: Rotors direction

The absolute linear position of the quadcopter is defined in the inertial frame x, y, z axes with ξ. The attitude, i.e. the angular position, is defined in the inertial frame with three Euler angles η. Pitch angle θ determines the rotation of the quadcopter around the y-axis. Roll angle ϕ determines the rotation around the x-axis and yaw angle ψ around the z-axis. Vector q contains the linear and angular position vectors [9].

$$\xi = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad \eta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}$$
$$\text{q} = \begin{bmatrix} \xi \\ \eta \end{bmatrix}$$

Figure 2.3: Orientation vectors

The origin of the body frame is in the center of mass of the quad-copter. In the body frame, the linear velocities are determined by V_B and the angular velocities by v.

$$V_B = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} \qquad \text{v} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Figure 2.4: Velocity vectors

To stabilize the quadcopter, a PID controller is utilized. Advantages of the PID controller are the simple structure and easy implementation of the controller. The general form of the PID controller is

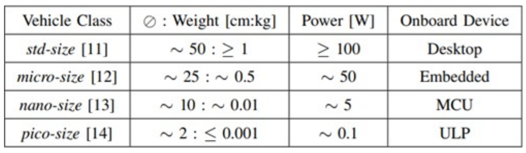
$$e(t) = x_d(t) - x(t) \tag{2.1}$$

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + k_D \frac{de(t)}{dt} \tag{2.2}$$

in which u(t) is the control input, e(t) is the difference between the desired state $x_d(t)$ and the present state x(t), and K_P, K_I and K_D are the parameters for the proportional, integral and derivative elements of the PID controller [3].

2.2 Quadcopters Nowadays

Commercial off-the-shelf (COTS) quadcopters have already started to enter the nano-scale, featuring only few centimeters in diameter and a few tens of grams in weight. However, commercial nano-UAVs still lack the autonomy boasted by their larger counterparts since their computational capabilities, heavily constrained by their tiny power envelopes, have been considered so far to be totally inadequate for the execution of sophisticated AI workloads, as summarized in Table 2.1 [13].



Vehicle Class	$\oslash$: Weight [cm:kg]	Power [W]	Onboard Device
std-size [11]	$\sim 50 : \geq 1$	≥ 100	Desktop
micro-size [12]	$\sim 25 : \sim 0.5$	~ 50	Embedded
nano-size [13]	$\sim 10 : \sim 0.01$	~ 5	MCU
pico-size [14]	$\sim 2 : \leq 0.001$	~ 0.1	ULP

Table 2.1: Quad-copters classes

Our system is based on ultra-low-power computing platform, and a 27 grams commercial, open-source Bitcraze Crazyflie 2.1 nano-quadcopter. Due to the limited capabilities of the Crazyflie, we are limited on the sensor payload. So we equiped the Crazyflie with two sensors for target-seeking: (1) a laser Multiranger to measure the distance in the front/back/left/right directions, which we use to avoid obstacles; (2) a flow deck to track motion on the z-axis and x-y plane to ensure stable flight. The microcontroller does all the processing required in real-time without any external assistance with only 192 KB of RAM for the full flight stack. Table 2.8 shows a comparison between the Nvidia computer (Jetson TX2), which is designed to perform AI tasks and the micro-controller (STM32) used by the Crazyflie [6].

Name	STM32F405	Jetson TX2
CPU	1-core 168MHz	6-cores 2GHz+
GPU	None	256-core 1300MHz
RAM	192 kB	8GB
Storage	1 MB	32GB
Power	0.14 W (max)	7.5 W

Table 2.2: Hardware Capabilities

2.3 Location Estimation

Our quadcopter is equipped with an IMU and barometer, but how exactly will we be able to know the quadcopter's location during flight? On the quadcopter's hardware there is an implemented Kalman Filter, which takes all the information from the sensors and the dynamic model and makes a fusion of the information according to the variance of each information type [17].

2.4 Reinforcement Learning

With the rise of the Internet-of-Things (IoT) era and rapid development of artificial intelligence (AI), embedded systems ad-hoc programmed to act in relative isolation are being progressively replaced by AI-based sensor nodes that acquire information, process and understand it, and use it to interact with the environment and with each other. The "ultimate" IoT node will be capable of autonomously navigating the environment and, at the same time, sensing, analyzing, and understanding it [12].

Reinforcement Learning is about learning from interaction how to behave in order to achieve a goal. The learner and decision-maker is called agent, while the thing it interacts with, and therefore everything outside of the agent, is called the environment. The interaction takes place at each of a sequence of discrete time steps t [8]. At each time step, the agent receives a state S_t from the state space S and selects an action A_t from a set of possible actions in the action space $A(S_t)$. One time step later, the agent gets a numerical reward $R_{t+1} \subset \rm I\!R$ from the environment as a consequence of the previous action. Now, the agent finds itself in a new state S_{t+1} [11].

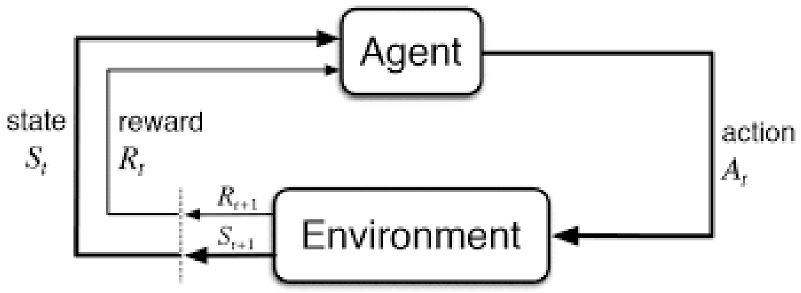


Figure 2.5: Agent-Environment Interaction

An important point to note - each state within an environment is a consequence of its previous state which in turn is a result of its previous state. However, storing all this information, even for environments with short episodes, will become readily infeasible. To resolve this, we assume that each state follows a Markov property, i.e., each state depends solely on the previous state and the transition from that state to the current state. [5]

2.5 Q-Learning

Q-learning is a policy learning algorithm that defines to the agent what action to take depending on his state, the algorithm does not require a model of the environment (i.e. model-free).The agent will seek to perform the sequence of actions that will eventually generate the maximum total reward. This total reward is also called the Q-value (state-action value) and we will formalise our strategy as:

$$Q(s,a) = r(s,a) + \gamma * \max_a(Q(s',a)) \tag{2.3}$$

In the above equation, r is the immediate reward for taking action a in state s, plus the highest Q-value possible from next state s', when γ is the discount factor, which controls the contribution of rewards further in the future. But with this way of calculation, we can see that the q-values at each state depend on the q-values of its next state, and since this is a recursive equation, we can start with making arbitrary assumptions for all q-values (e.g. $Q(s,a) = 0, \quad \forall s, a$).

With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * \max_a(Q(S_{t+1}, a) - Q(S_t, A_t))] \tag{2.4}$$

Where α is the learning rate/step-size, which will determine to what extent newly acquired information overrides old information [18].

2.6 DQN

One of the primary goals of the field of artificial intelligence is to solve complex tasks from unprocessed, high-dimensional, sensory input. Recently, significant progress has been made by combining advances in deep learning for sensory processing with Q-Learning, resulting in the "Deep Q Network" (DQN) algorithm [10] that is capable of human level performance on many Atari video games using unprocessed pixels for input. To do so, deep neural network function approximators were used to estimate the action-value function.

However, while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. DQN cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

An obvious approach to adapting deep reinforcement learning methods such as DQN to continuous domains is to simply discretize the action space. However, this has many limitations, most notably the curse of dimensionality: the number of actions increases exponentially with the number of degrees of freedom. For example, quadcopter have 6 degree of freedom with the coarsest discretization $a_i \in \{0, 1\}$ for each motor leads to an action space with dimensionality: $2^6 = 64$. The situation is even worse for tasks that require fine control of actions as they require a correspondingly finer grained discretization, leading to an explosion of the number of discrete actions. Such large action spaces are difficult to explore efficiently, and thus successfully training DQN-like networks in this context is likely intractable. Additionally, naive discretization of action spaces needlessly throws away information about the structure of the action domain, which may be essential for solving many problems [8]. In this work we present a model-free, off-policy algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. Eventually, the action space is discretized into a discrete action space: $\mid A \mid = 3$.

3 | Methods

We aim to implement a new solution for navigation in cluttered and unknown environments with implementation of a deep reinforcement learning algorithm. Prior work [1, 15] has shown that all the implementations of the DRL algorithms have been using a camera input, and that there is no solution for navigating without the use of a camera.

3.1 The Environment

In this project we used AirSim, is a simulation platform from Microsoft for autonomous vehicles such as cars and quad-copters. For the graphics, it utilizes Unreal Engine [7] that provides many options to build the environment. AirSim is a very realistic simulator and the team that developed it published evaluation of a quad-copter model and find that flight tracks in the simulator are very close to the real world drone behaviour [14].

In particular, in this work we adjust the blocks environment to meet our requirements. The room set to 20m x 20m with five cylindrical obstacles, in fixed location (Figure 3.1). The drone starts in $(0, 0, 0)$ and the target is randomly chosen, before each episode, by the uniform distribution. The target is defined in the X-axis and the Y-axis and the flight altitude is constant and does not change during navigation. During a flight when the drone collides with an obstacle or wall, the environment resets and the drone returns to the $(0, 0, 0)$ point and a new episode begins.

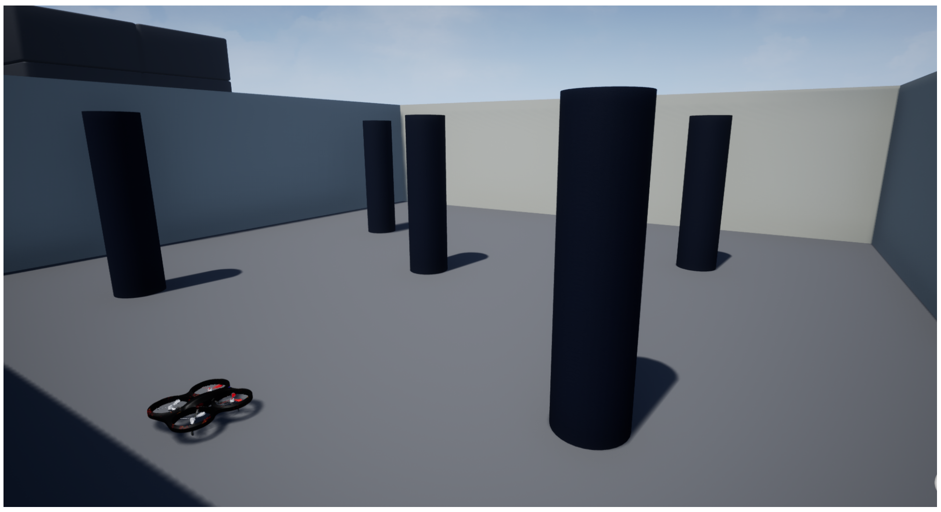


Figure 3.1: Simulation room

3.2 Reward Function

The reward function is presented in Equation 3.1; If the agent has reached its goal then α is set to 1. If the agent has collided with an obstacle or wall, then β is set to 1. β is also set to 1 if the agent stayed in its place for 8 steps or more, and if the agent took 300 steps in one episode. The Δd is the difference between the agent's distance to the goal in time step t and time step $t-1$, and that is in order to stimulate actions that get the agent closer to the goal and penalize for getting further away. The -1 exists in order to penalize delays [6].

$$Reward = 1000 \cdot \alpha - 100 \cdot \beta - 20 \cdot \Delta d - 1 \tag{3.1}$$

3.3 Challenges in RL

While in supervised learning we want the input to be independent and identically distributed - samples are randomize, therefore each batch has a similar distribution and samples are independent in the same batch. If this is not the case the model might be over-fitted and the solution will not be generalized.

As shown in equation 2.4, the target values for Q depends on Q itself, for this reason we chase a non-stationary target [18]. Such challenges can be overcome by two solutions; 1) Experience replay - each transition insert into a buffer and than the agent sample a mini-batch of samples to train the deep network. 2) Target network - we create two deep networks with the same architecture θ and θ^{-}. The first one includes all the updates in the training and the second one is used to retrieve Q values from s_{t+1}.

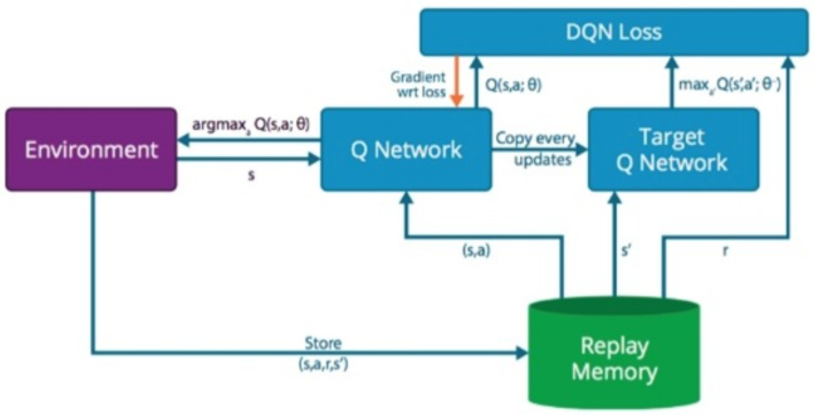


Figure 3.2: DQN Flow Chart

There are two possible ways of updating the target network; 1) Hard update - Every 10 episodes we synchronize θ^{-} with θ. 2) Soft update - Every time step the target network is updated according to the Equation3.2

$$\theta^{-} = \tau\theta + (1-\tau)\theta^{-}, \quad \tau \in [0,1] \tag{3.2}$$

Note - τ is usually close to 0 (e.g. 0.01).
The purpose is to fix the Q-value target so we don't chase the target. With both experience replay and the target network, we have more stable input and output to train the network with, so it behaves more like supervised learning.

One of the most difficult problems in RL is balancing the explroation/exploitation tradeoff [16]. On the one hand, the agent needs to explore the environment, i.e. try different actions in a given state so it will not get stuck in a local minimum, but on the other hand letting the agent choose randomly each time will cause the model not to converge. Therefore, we let it explore at the beginning, and with time, it will explore much less and start to exploit the Q-values.
This occurs in order to satisfy the GLIE properties [18]:
1) All state-action pairs are explored infinitely many times

$$\lim_{k\to\infty} N_k(s,a) = \infty \tag{3.3}$$

2) The policy converges on a greedy policy

$$\lim_{k\to\infty} \pi_k(a \mid s) = 1(a = argmax_{a\in A}Q(s,a)) \tag{3.4}$$

The GLIE function is presented in Equation 3.5.

$$\epsilon = \begin{cases} \epsilon \cdot \epsilon_{decay}, & if \quad \epsilon > \epsilon_{min} \\ \epsilon_{min}, & else \end{cases} \tag{3.5}$$

Hyperparameters	Value	Notes
episodes	3000	
epsilon start ϵ	1.1	
epsilon end ϵ	0.01	
epsilon decay	0.95	
batch size	64	
replay memory	100,000	
discount factor γ	0.99	
learning rate α	0.0025	Adam optimizer
target update period	10	relevant for hard update

Table 3.1: Hyperparameters for the training

3.4 Work Process

At each time step the agent performs an action according to the network (Figure 3.4) or chooses a random action with a certain probability, in order to prevent it from staying at a local minimum. After performing the action, the environment outputs: the reward that the agent received for performing the action, the new state it reached and whether it completed the current episode or not (reached the goal or collided with an obstacle). This information is stored at every step the agent performs, and forms a database for the training of the neural network.

After a large enough number of steps the network should converge and so the algorithm knows for each state of the agent - what is the action that will bring it the highest value and ultimately direct it to the target.

The real-life drone uses the neural network in the same way, and in fact the network receives the information (agent state) from it and outputs which action is the best to perform.

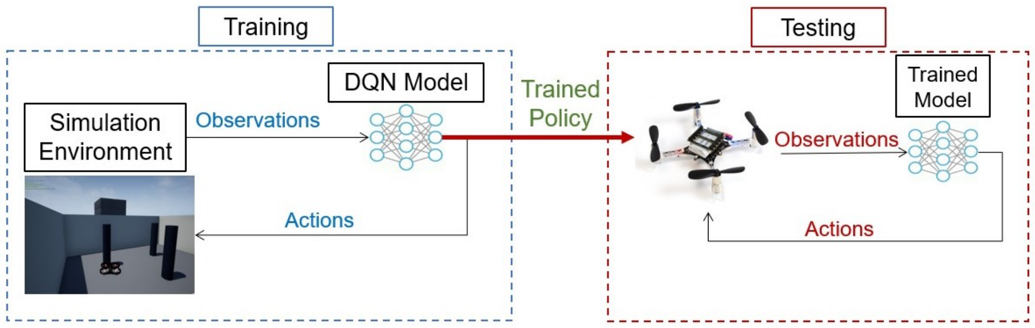


Figure 3.3: Methodology

The observations are modified in a way that will let the neural network learn the most out of it. An observation is defined like that:

Agent's X axis position, Y axis position, euclidean distance to the goal, angle to the goal, and the reading of the front distance sensor (IR sensor). Eventually, the network outputs which action to take: Go straight, turn right or turn left.

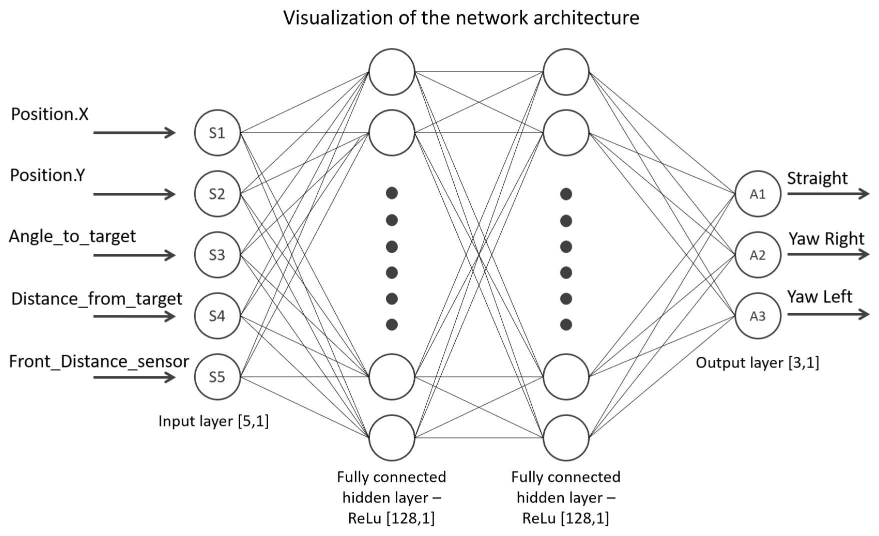


Figure 3.4: Network Architecture

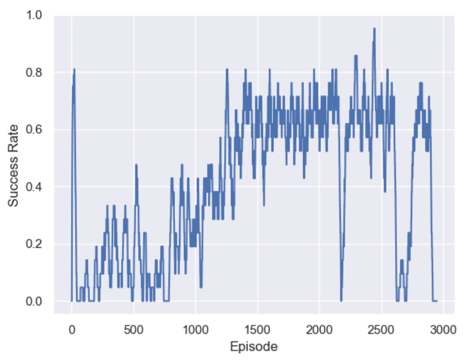
We use the ReLu activation function. The activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.

The rectified linear activation function is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it, is easier to train and often achieves better performance [4].

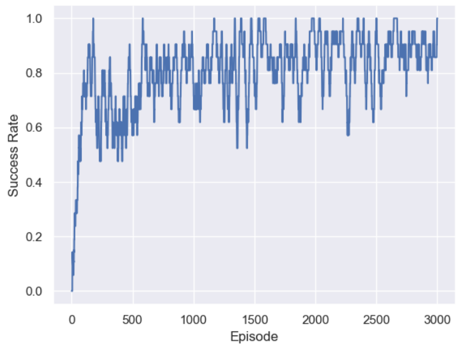
4 | Results

4.1 Training

Two different kinds of training were executed. One is without using a target Q network, and is shown in Fig 4.1(a). The other is with the use of a target Q network as shown in Fig 4.1(b)



(a) Without Target Q network



(b) With Target Q network

Figure 4.1: Training results

As we can see in Fig 4.1, the target Q network has led the model to convergence and high success rate, while the model that didn't use the target Q network did not converge, as the model led to low success rates after a large number of episodes. It seems that without the target Q network the model really is "chasing its own tail". After training, we evaluate our model in the simulation and we let the agent perform 50 episodes with different targets. As we can see in Fig 4.2, our model leads us to a 92% success rate.

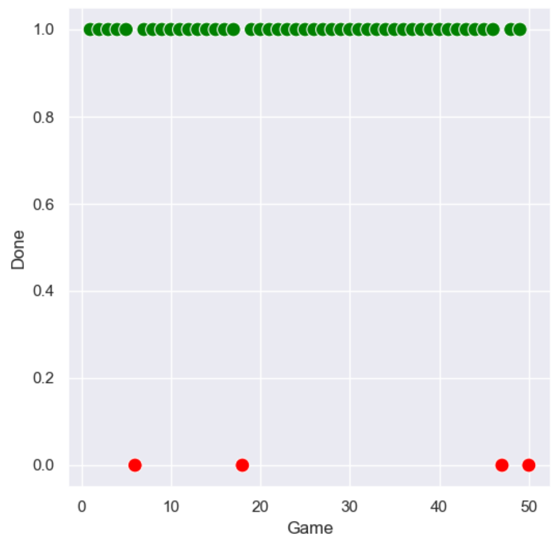


Figure 4.2: Evaluation of the trained policy in the simulation

4.2 Testing

In order to test the model in a real environment, the commands are sent to the drone from a computer in RF communication. The computer gets a noisy sensory input from the drone, put it into the model, processes it and sends the output (i.e. the best action according to its state). For flight test we use a room that is approximately 4m x 3m, the flight altitude was set to $0.6m$ to avoid severe crashes. We count a successful run as the drone gets into a $1m$ radius from the target.

The flight test includes 5 flights that 3 of them were successful (60% success). The drop in success rate is explained by the fact that in reality the front distance sensor of the drone, during flight, shows lower values than they would show in the simulation, when it comes to high distances. This problem causes the drone to stay in its position, even if there are no obstacles in its way, until it gets the real value from the front distance sensor.

5 | Discussion

In this work we present a fully autonomous goal seeking nano drone, based on a deep reinforcement learning policy trained in simulation. AirSim, the simulation platform we used has an option only for a front distance sensor, while CrazyFlie has a five-direction distance sensors. This gap led us to use the network in a different way than what we first aimed to - reading from the front, back, left and right distance sensors plus the distance to the goal and the relative angle to the goal. Other than that, the AirSim platform simulates the environment in a very realistic way, which makes the simulation-to-reality gap pretty small, so that the implementation of the model on the real drone didn't require any fine tuning.

The application of a deep reinforcement learning algorithm is a big challenge, as training in simulation is a necessity, because the drone that we used is only capable of flying indoors, as it is very light-weighted so it can fluctuate and crash even when a weak wind blows. Additionally, the lighting conditions are very important because of how its stabilization works (i.e. the optic flow deck). Despite all that being said, navigation of a drone in a cluttered environment (with no GPS signal) using DRL is feasible.

In this project, we integrated between a Python API, a simulation engine and theoretical knowledge and comprehension. We had to learn how to use the Unreal Engine from scratch, including installation of several software programs, as documented in [19].

6 | Future Work

We anticipate an expansion of the applicability of our work - a sensor that captures light intensity, radiation or gas leaks can be installed on the drone, thus converting its target to an external source.
It is possible to combine the flight of several CrazyFlie drones to divide the problem into sub-problems - for example, letting every drone search separately for gas leaks in collapsed buildings.
In order to get even better results with harder environmental conditions such as a darker room or a more cluttered environment, usage of better sensors is required, and a simulation engine that allows it. One major thing that can be done, is to embed the trained policy on the drone's firmware, what will require a compression (i.e. quantization) of the model, due to the limited computational capabilities of the drone.

REFERENCES

[1] Aqeel Anwar and Arijit Raychowdhury. Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning. *IEEE Access*, 8:26549–26560, 2020.

[2] Lucas M Argentim, Willian C Rezende, Paulo E Santos, and Renato A Aguiar. Pid, lqr and lqr-pid on a quadcopter platform. In *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 1–6. IEEE, 2013.

[3] Karl Johan Åström and Tore Hägglund. *PID controllers: theory, design, and tuning*, volume 2. Instrument society of America Research Triangle Park, NC, 1995.

[4] Jason Brownlee. A gentle introduction to the rectified linear unit (relu). *Machine Learning Mastery. https://machinelearningmastery. com/rectified-linear-activation-function-fordeep-learning-neural-networks*, 2019.

[5] Ankit Choudhary. A hands-on introduction to deep q-learning using openai gym in python. *Retrived from https://www. analyticsvidhya. com/blog/2019/04/introduction-deep-q-learningpython*, 2019.

[6] Bardienus P Duisterhof, Srivatsan Krishnan, Jonathan J Cruz, Colby R Banbury, William Fu, Aleksandra Faust, Guido CHE de Croon, and Vijay Janapa Reddi. Learning to seek: Deep reinforcement learning for phototaxis of a nano drone in an obstacle field. *arXiv*, pages arXiv–1909, 2019.

[7] Epic Games. Unreal engine. https://www.unrealengine.com.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with

deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] Teppo Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 22, 2011.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[11] Guillem Muñoz, Cristina Barrado, Ender Çetin, and Esther Salami. Deep reinforcement learning for drone delivery. *Drones*, 3(3):72, 2019.

[12] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini. A 64-mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 6(5):8357–8371, 2019.

[13] Daniele Palossi, Jaskirat Singh, Michele Magno, and Luca Benini. Target following on nano-scale unmanned aerial vehicles. In *2017 7th IEEE international workshop on advances in sensors and interfaces (IWASI)*, pages 170–175. IEEE, 2017.

[14] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[15] Sang-Yun Shin, Yong-Won Kang, and Yong-Guk Kim. Obstacle avoidance drone by deep reinforcement learning and its racing with human pilot. *Applied Sciences*, 9(24):5571, 2019.

[16] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

[17] Shu-Li Sun and Zi-Li Deng. Multi-sensor optimal information fusion kalman filter. *Automatica*, 40(6):1017–1023, 2004.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] Omri Tzur and Matan H Kolpanitzky. Drl-crazyflie. https://github.com/omritz/DRL-Crazyflie, 2020.