UAS Landing with Deep Reinforcement Learning

Zhenhao Zhao
Dept. of Computer Science
Dept. of Biomedical Engineering
The George Washington University
Washington, DC, 20052, USA
zzhao98@gwu.edu

Jonathan Lee
Dept. of Computer Science
The George Washington University
Washington, DC, 20052, USA
jonathanlee@gwu.edu

***Abstract*—Small unmanned aircraft systems (UAS) have proven their usefulness for operations such as aerial cinematography, last-mile package delivery, and infrastructure monitoring. In order to achieve higher levels of autonomy and scale up operations without tele-operating human pilots, we investigate a method for autonomous UAS landing with deep reinforcement learning. Specifically in this paper, we train a Deep Q Network to learn control policies for landing a multirotor on a landing pad. We apply our method to an AirSim simulation environment with stationary obstacles and a landing pad.**

***Index Terms*—Unmanned Aircraft System (UAS), Autonomous Landing, Deep Reinforcement Learning (DRL), DQN**

I. Introduction

Autonomous landing requires precise control to both identify the landing pad and position the multirotor above the landing pad. Additionally, there is variability in each landing due to the aircraft approaching the landing pad from different directions or altitudes, and different situations on the ground. To address this problem, we formulated autonomous landing as a reinforcement learning problem where the drone must take an optimal set of actions to land on the landing pad.

Numerous approaches have been taken for autonomous landing of a drone, including both classical and deep learning approaches [1] [2]. The advantage of using deep reinforcement learning is that the drone can be trained to autonomously land without any labeled data. However, the main challenge is combating a large state space and exploring optimal policies. While we do not introduce a new deep reinforcement learning method, our contribution is building a deep reinforcement learning based solution in the AirSim simulation environment.

II. Related Work

A. Safety Awareness for UAS Landing

An accurate, fast, and reliable perception module is a key component for a fully autonomous landing stack, together with a prediction module and planning/control module. In our previous research [3], we demonstrated an algorithm for vision-based perception with safety awareness for UAS autonomous landing. In the paper, we addressed the issue of a UAS needing to recognize landing scene reconfiguration such as moving or static obstacles near or on the designated landing pad. In real-world landing situations, obstacles often include people, pets, cars, and bikes. Thus, we explored the feasibility of using computer vision based perception to (1) identify the landing pad and (2) detect the obstacles near or on the landing pad. The system would then alert a planning/control module that there is a hazard in the landing zone. We demonstrated real-time detection on video footage we collected from real world landings.

In this paper, we focus on a different aspect of autonomous landing, the development of a planning/control module that implements a policy to avoid obstacles and land on a landing pad. We kept in mind that our planning/control module should be inter-operable with our previous perception module. Therefore, our model uses only images taken from a downward facing camera on the UAS as the input. In future work the perception and planning/control modules could be linked together to provide a safer landing approach that pauses the landing when a hazard is nearby the landing pad.

B. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) was first introduced by DeepMind, an artificial intelligence research organization [4]. In this paper, they proposed the DQN algorithm to play Atari games through an agent learning solely from image input from video frames. Afterwards, DeepMind published an improved version of the DQN algorithm [5] on Nature, which combines deep learning with RL to implement a new end-to-end learning algorithm from perception to action. A high-level anecdote is the learning process of humans perceiving visual information, such as what the eyes see, and then directly making corresponding behaviors (output actions) through the brain (deep neural network). Subsequently, DeepMind proposed AlphaZero which used DRL and Monte Carlo Tree Search [6] to achieve a level of performance beyond the most competitive human for the game of chess.

III. Methods

A. Problem Formulation

The problem of landing a drone can be formalized as a Markov Decision Process (MDP) where an agent interacts with the environment. At each time-step, the agent can select a single action. This action results in a change in state in the environment.

The simulation keeps track of the vehicle's previous position, current position, and whether it has collided with an obstacle or not. This allows the simulator to determine the vehicle's velocity and reset the simulation when a collision occurs. All positions are measured from a global reference frame in Cartesian coordinates.

The entire internal state of the environment is not completely observable to the agent. In our situation, the agent is equipped with a downward facing color camera. It can only observe the world through images taken by the camera. The input of these images into the DQN model will be discussed in the next section.

The action space contains 7 discrete actions $a \in \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$. Each of these actions in Equation (1) map to a vector $(x, y, z)^\top$ corresponding to a the components of a velocity vector in 3D space. This velocity vector is added to the drone's current velocity. Then flight controller is instructed to fly with the new desired velocity components V_{t+1} in Equation (2).

$$\begin{aligned} a_0 &= (v, 0, 0)^\top \\ a_1 &= (0, v, 0)^\top \\ a_2 &= (0, 0, v)^\top \\ a_3 &= (-v, 0, 0)^\top \\ a_4 &= (0, -v, 0)^\top \\ a_5 &= (0, 0, -v)^\top \\ a_6 &= (0, 0, 0)^\top \end{aligned} \tag{1}$$

$$V_{t+1} = V_t + a = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} + \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \tag{2}$$

Additionally, the agent receives a reward, $R(s, a)$ at each time step. The reward is related to the state of the environment, s, and the action of the drone, a. Positive rewards correspond to desirable results, such as descending towards the landing pad, and low rewards correspond to poor actions, such as crashing into obstacles. The goal of the agent is to select actions that will maximize its future rewards.

Therefore, our reward function should incentivize drone behavior that moves the drone closer to the landing pad and disincentive the drone from flying away from the landing pad. Hence, the reward is a function of the distance $d(s)$ between the drone's position, s, and the landing pad position, p in Equation (3).

$$d(s) = \sqrt{(p_x - s_x)^2 + (p_y - s_y)^2 + (p_z - s_z)^2} \tag{3}$$

We implement the reward in Equation (4) as a piecewise function. If the drone is right above the landing pad, it receives a reward of 100. If the distance to the landing pad is greater than a threshold τ, then the reward is -10. We set τ to a value greater than $d(s_0)$ to penalize the drone from flying farther than its starting position. If the distance is less than τ, then the drone will receive a reward proportional to the square root of $d(s)$. We use this function because when the drone is approximately $\tau/3$ from the landing pad, it starts receiving a large positive reward. Finally, if a collision with an obstacle is detected, the reward is -100. The episode ends immediately after receiving a reward for landing or colliding with an obstacle.

$$R(s) = \begin{cases} 100 & d(s) \le 1 \\ 4 - \sqrt{d(s)} & 1 < d(s) \le \tau \\ -10 & d(s) > \tau \\ -100 & \text{if collision} \end{cases} \tag{4}$$

The optimal action-value function $Q^*(s, a)$ is defined as the maximum expected reward achievable by following any strategy. Once the optimal action-value function is determined, the optimal strategy is simply the action that maximizes $Q^*(s, a)$. However, in practice, it is impractical to compute the action-value function due to the large state space. Instead, we apply deep Q-learning to best approximate this function.

B. Deep Q Network

The DQN algorithm is a method of passing Q-learning through a neural network to approximate the value function. It has achieved results beyond human-level players in classic Atari 2600 games. We take Break Out Bricks as an example. It has a high-dimensional state input (original image pixel input), low-dimensional action output (discrete actions: up, down, left, right, firing shells, etc.). A screenshot of Break Out Bricks as shown in Figure (1).

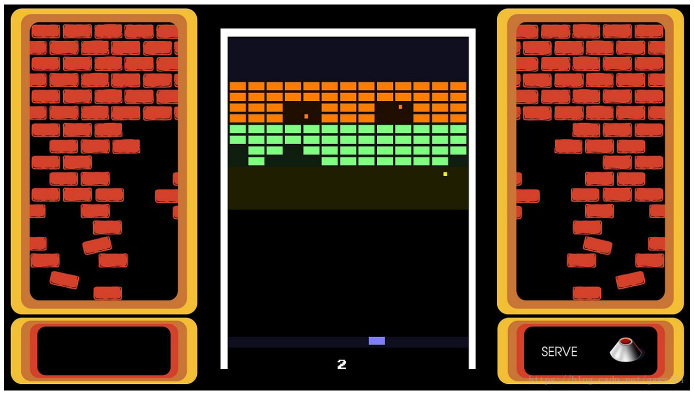

Fig. 1. Break Out Bricks game

The input of our DQN algorithm is 4 frames of the game displays. We first pre-process the images by cropping and converting the original RGB image to grayscale images such that the images are now single channel. The size of each pre-processed frame should be 84 x 84. After that, we stack 4 such pre-processed frames to make a 3D array, the size of of which should be 84 x 84 x 4. Then this 3D array is passed as input to the deep convolutional neural network to calculate the q value of each action so we can decide the optimal action to choose. The whole structure is shown in Figure 2. The model first uses two convolutional layers to extract the features of the input. And then analyze it and output the q value of each actions by using two full connected layers. The output state

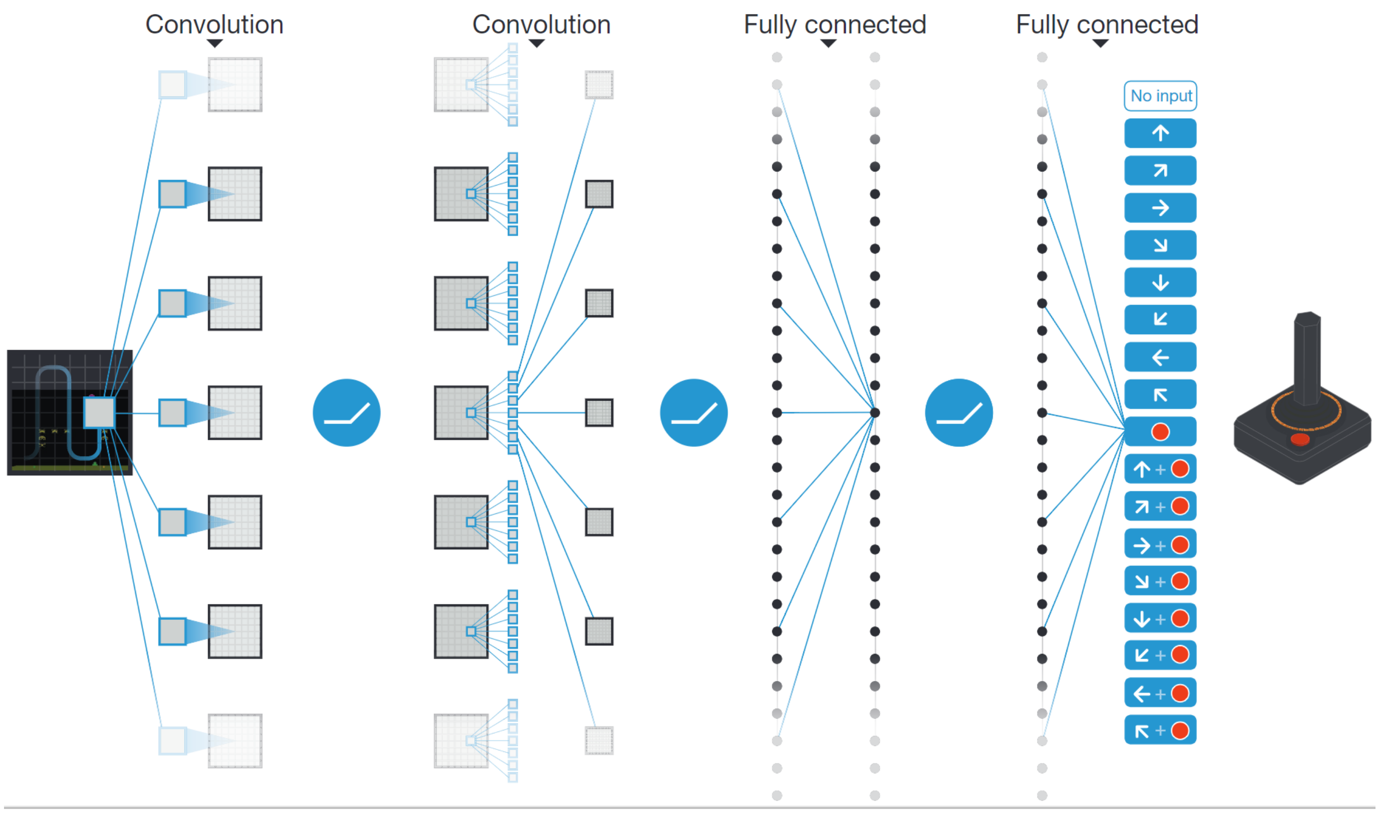

Fig. 2. The structure of the DQN

number will change depending on the different games. There are 7 different actions to control our multirotor.

The loss function is based on the reward and it uses two networks. The evaluation network will output the prediction of current state: $\hat{Q}(s_{j+1}, a'; \theta)$ directly, and the target network will output next state q value: $\hat{Q}(s_{j+1}, a'; \theta^-)$. The ground truth q value y_j of current state is calculated by adding the output of target network to the instant reward r_j.

$$y_j = \begin{cases} r_j & \text{terminate} \\ r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \quad (5)$$

Therefore, based on the model prediction q value, the loss is calculated by the following error:

$$E_i(\theta_i) = y_j - \hat{Q}(s_{j+1}, a'; \theta) \quad (6)$$

We can keep the target Q value constant for a period of time, which reduces the correlation between the current Q value and the target Q value and improves the stability of the algorithm.

Another advantage of DQN is replay buffer. DQN will store the transitions (s, a, r, s') in an array as a buffer and then train the model by randomly selecting transitions from this buffer. The principle is very similar to that of stochastic gradient descent (SGD). Training samples should be independent to each other, but neighboring transition are highly correlated. The replay buffer will store in memory all the transitions in a period of time to make the training data distribution more steady. Additionally, it will forget transition that occurred too far in the past to make sure the distribution of the replay buffer can simulate the current distribution of the policy.

IV. Experiments

A. AirSim

We conducted our experiments in AirSim [7], an open-source, cross platform simulator for drones and cars. It is built on Epic Games' Unreal Engine 4 [8]. We chose this platform because it can create physically and visually realistic simulations, and it is a popular choice for researching AI algorithms for autonomous vehicles.

Figure (3) shows our simulation environment setup. We use a flat world model containing obstacles composed of stacked blocks. These obstacles of different heights and sizes can be representative of buildings, trees, or vehicles.

The landing pad is represented by a small orange square, of similar size and shape to that of small hobby drone landing pads. The drone's starting position is 10 meters to the left and 10 meters down from the landing pad with an elevation of 50 meters. The simulation starts each episode with the drone hovering with zero initial velocity.

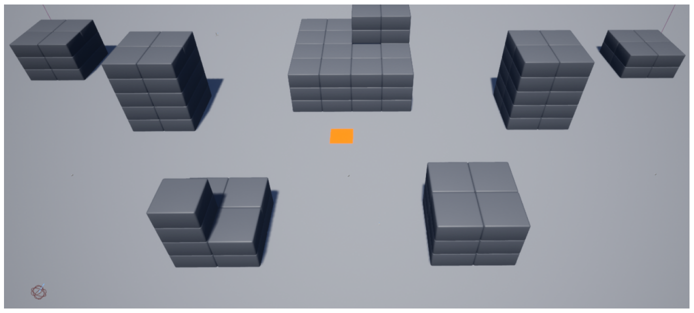

Fig. 3. Simulation Environment

We utilize the Python API to control the multirotor in the simulation programmatically. Through the API, we are able to retrieve images, get vehicle state, and control the vehicle through either position or velocity control commands. The multirotor has a downward facing camera that supports depth and color images. The color images used as the input to the DQN after down sampling and cropping.

Figure (4) shows an image taken by the drone's camera after being converted to grayscale. The final steps of pre-processing downsample and crop to a square image.

Fig. 4. DQN Input Image

B. Training

The DQN is trained by setting up AirSim as an OpenAI gym wrapper [9]. The gym wrapper allows us to run a training loop where we sequence through obtaining an image, computing the action to take based on the current policy, getting a reward, etc. The training hyperparameters are configured as in Table I.

V. Results

The result of our training process is shown in Figure 5 and Figure 6. We track the reward and distance to the landing pad for each transition. As shown in Figure 5, the distance between the drone and landing pad decreases over time. Additionally, the reward converges towards zero over time. While our results appeared promising, we did not quite achieve the drone landing consistently on the landing pad. In our testing, the multirotor seemed to approach only half way to the landing pad.

TABLE I
DQN Hyperparameter Training Parameters

Parameter	Value
learning rate	0.00025
batch size	32
training freq	4
target update interval	10000
learning starts	200000
buffer size	500000
max grad norm	10
exploration fraction	0.1
exploration final eps	0.01

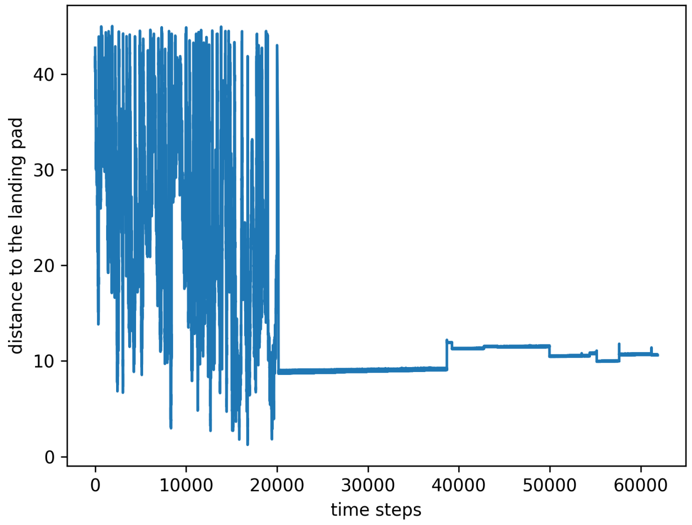

Fig. 5. Distance decreasing over time

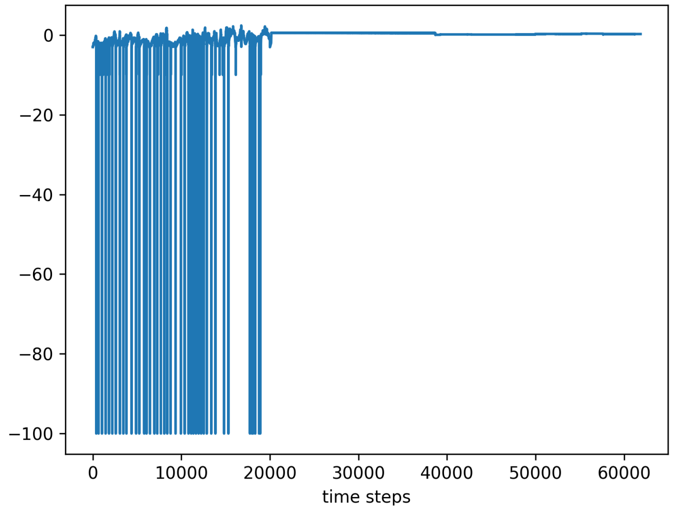

Fig. 6. Reward increasing over time

In Figure 7, we show screenshots of the drone executing the optimal policy computed while training. The drone descends towards the ground and moves horizontally towards the landing pad. However, it never actually lands on the landing pad.

We were not able to diagnose this issue and find an explanation as to why the performance stalled before landing.

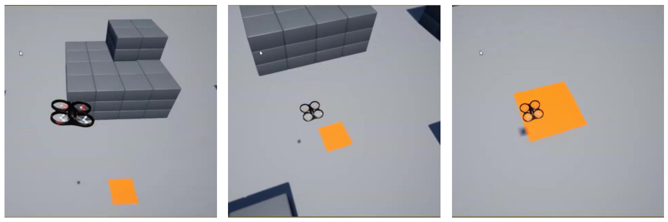

Fig. 7. Screenshots of multirotor in the simulator

VI. Conclusion and Future Work

In this work, we have demonstrated our approach to land a multirotor on a designated landing pad using deep reinforcement learning. We recognize that our approach is only the start to a planning/control module, and there is much to be improved.

Our approach discretized the action space into 7 actions corresponding to directions in a grid world. This would not be precise enough in a real-world landing scenario, and a continuous action space is more desirable to fine-tune the landing.

In the future, we can make the simulation scenario more realistic. For example, the initial location of the drone and landing pad can be randomized. Additionally, new scenes can be introduced to include moving obstacles, such as pedestrians, cars, and buildings.

References

[1] Y. Xu, Z. Liu, and X. Wang, “Monocular vision based autonomous landing of quadrotor through deep reinforcement learning,” in *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 10 014–10 019.

[2] M. Saavedra-Ruiz, A. M. Pinto-Vargas, and V. Romero-Cano, “Monocular visual autonomous landing system for quadcopter drones using software in the loop,” *IEEE Aerospace and Electronic Systems Magazine*, vol. 37, no. 5, pp. 2–16, 2022.

[3] Z. Zhao, J. Lee, Z. Li, C. H. Park, and P. Wei, “Vision-based perception with safety awareness for uas autonomous landing,” in *SciTech*, 2023. [Online]. Available: https://lnkd.in/eDZUJHkA

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2015.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” 2017. [Online]. Available: https://arxiv.org/abs/1712.01815

[7] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[8] “Unreal engine 4,” https://github.com/EpicGames/UnrealEngine, 2021.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016. [Online]. Available: https://arxiv.org/abs/1606.01540