COMP34212: Cognitive Robotics Report on Deep Learning and Robotics

Harry O'Brien, a64070ho

May 2022

1 Introduction

Deep Reinforcement Learning (DRL) is a model-free, offline, machine learning algorithm that is used by robots to find a general solution to a given task. DRL has been shown to produce human-like skill in computer games, autonomous driving and autonomous flight [9][8]. DRL uses a neural network to consider the current state of a given environment. It then predicts an action that results in the highest long-term gains in achieving a target objective.

In autonomous flight, generating and following a path to fly along is an extremely technical challenge. This is a perfect application of DRL and replaces typical state-estimation algorithms and negative feedback loops. In this work, a drone is tasked with hovering in a given position using onboard sensors as input. This is done in simulation using an implementation of the Twin Delayed DDPG (TD3) algorithm. The produced algorithm serves as a baseline to build upon for a aerial robot to learn more complex manoeuvres. Since the network does not take an image as input, the network can consist entirely of fully connected layers with either ReLu, tanh and linear activation functions. The hyperparameters of the TD3 algorithm are modified during experimentation to reduce convergence time and to optimise the score achieved by the agent.

2 State-of-the-Art Contextualisation

This section gives an overview of the state-of-the-art algorithms used in this work and how this applies to robotics. Critical analysis is given to the role of deep learning in autonomous flight and context given to how this work relates to the latest works in the same field.

2.1 Online and Batch Learning

In Deep Learning (DL), the amount of data that can be categorised into online and batch (sometimes class *offline*) learning. With online learning, the data are presented to the learner one-by-one [2]. This is in contrast to batch learning where all of the data are presented to the learner before the network updates its parameters.

In DRL, learning is performed over a small sample of the data-set. This is known as a mini-batch. The use of mini-batches in DRL aims to reduce the time taken to train an agent while also reducing the number of previously seen samples.

2.2 Reinforcement Learning & Deep Reinforcement Learning

Reinforcement Learning (RL) is an area of Machine Learning (ML) that is concerned with sequential decision-making [3][5][9]. In RL, an *autonomous agent* learns how to achieve a given objective through repetitive interactions with its environment. The interaction between an agent and its environment can be formalised as a Markov Decision Process (MDP) [1], described by a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{S}')$. At each time step t, an agent interacting with an environment observes a state $s_t \in \mathcal{S}$ and performs an action $a_t \in \mathcal{A}$ which determines the reward $r_t \sim \mathcal{R}(s_t, a_t)$ and the next state $s_{t+1} \sim \mathcal{S}'(s_t, a_t)$ [3][5]. An implementation of RL is the Q-learning algorithm. Q-Learning is a model-free off-policy algorithm for estimating the long-term expected return of executing an action a from a given state s. The estimated return values are known as *Q-values*, where a higher Q-value indicates that an action is believed to yield better long-term results. Q-values are stored in a Q-table and are updated during learning to reflect how successful an action in a given state. Although successful in some scenarios, RL algorithms are limited to the number of possible state-action pairs that can feasibly be stored in memory or to domains with fully observed, low-dimensional state spaces [9].

DRL is a modification of the RL algorithm and is described as an unsupervised, offline learning algorithm - unsupervised meaning that the algorithm learns from non-labelled data. In DRL, a Deep Neural Network (DNN) to consider the state of the environment and predict a Q-value for all possible actions. This is done instead of using a Q-table; the DNN acts as a function approximator to replace part of the RL update function. Hence the limitation of the number of possible state-action pairs is removed. Because of this, the use of deep learning models can be implemented to learn a general solution to a complex problem.

In DRL, loss, $\mathcal{L}(\theta)$, is calculated using a Mean Squared Error (MSE) function and evaluates how well the algorithm is performing. MSE measures the average of the squared distance between predictions and observations. It is only concerned with the magnitude of the prediction and, due to the exponential factor, predicted values that are further away from the actual value are penalised much more significantly.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value pair. This is caused by multiple factors [9]:

- Correlations in sequences of observations used since they happen chronologically
- The fact that small updates to a Q-value may significantly change the policy and therefore change the data distribution
- Correlations between the action-values (Q-values) and the target values $r + \gamma \cdot \max_{a'} Q(s', a')$

To remove these correlations, an experience replay buffer is used to store the transition between states. When training a DRL algorithm, random unique experiences are sampled from this buffer to form a mini-batch. This mini-batch is then fed to the network as training data. Furthermore, a second, target network is introduced that is updated less frequently. This further reduces correlations between recently seen states and the expected Q-value of an action, thus making the algorithm more stable. This is called Double Deep Q-Learning (DDQN).

To combat the DRL agent from settling into local minima, an algorithm called *epsilon decay* is used. This algorithm has a reducing threshold value, ϵ, that dictates whether an agent is more likely to exploit the current state of the environment for a high Q-value or to select a random action to further explore the environment. Thus, epsilon decay addresses the *exploration versus exploitation* problem which is key to finding previously unknown, high-reward states during training. Initially, the agent selects more random

actions to explore the environment. In contrast, towards the end of training, the agent is more likely to exploit the current environment to repeatably enter states with high rewards.

2.2.1 Actor-Critic Networks

Algorithms such as deep-Q learning are actor-only algorithms and suffer from high variance and noisy gradients, and can only work in discrete action spaces. Actor-critic networks are designed to address these issues. The critic uses an approximation architecture to estimate a value function, typically a Q-value, which is then used to update the actor's policy parameters in the direction of best performance. Hence, the value estimation and action selection functions are separated. This means that algorithms using actor-critic networks can construct a good approximation of the value function while maintaining a low variance during action selection.

2.3 Deep Deterministic Policy Gradient

Lillicrap et al. have [7] designed and implemented a general purpose, actor-critic, model-free algorithm that can operate over continuous action spaces. Based on Deterministic Policy Gradient (DPG), the algorithm's use of a DNN as a function estimator led to the name *Deep Deterministic Policy Gradient (DDPG)*. The ability for a robot to operate in continuous action spaces is critical since it may not always be possible for the observation and action spaces in real-world environments to be divided up into a discrete space (for example Cartesian coordinates or joint angles). This allows for the output of the robot's network to be used directly as commands to motors and actuators.
In DDPG, the actor and critic networks are trained separately. As with typical DRL, a replay buffer is used to store experiences and samples are taken that are assumed to be independent and identically distributed. Mini-batches are also used during implementation to make efficient use of hardware optimisations. The critic $Q(s, a)$ is learned using the Bellman equation as in Q-learning, whilst the actor network is trained by freezing the critic, meaning that the critic cannot adjust its weights, and then trained to minimise MSE loss.
The implementation discussed above made the network inherently unstable. By taking inspiration from DDQL target actor and critic networks were created for the DDPG algorithm. This inevitable helped the algorithm become more stable. Furthermore, to ensure that a local maxima isn't reached, a noise process is implemented inside the actor policy.
Lilicrap et al. found that their algorithm was able to have results competitive to planning algorithms that had full access to the dynamics of the given domain and its derivatives. Since DDPG treats the problem of exploration independently from learning, the algorithm was able to explore continuous action spaces. In some tasks, learning from pixel values rather than simple inputs, such as a robot's arm-angle, was "just as fast" [7].
However, because nonlinear function approximators are introduced in DDPG, there is a chance that convergence never happens. Therefore many experiments are required to be run to find the optimum hyperparameters. Many of these experiments would result in the network never converging. This requires many hours of additional experimentation and GPU time for training each different model. For this reason, Fujimoto et al. developed the TD3 algorithm [4].

2.4 Twin Delayed DDPG

TD3 addresses the issues in DDPG by introducing three additional features [4]:

1. **Clipped Double-Q Learning**. TD3 learns two Q-functions instead of one and uses the smaller of the two Q-values to form the targets in the loss functions.
2. **Delayed Policy Updates**. TD3 updates the policy and target networks less frequently than the Q-function.
3. **Action noise regularisation**. TD3 adds noise to the target action to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

In summary, the extension of the DDPG algorithm with these changes results in an algorithm that is more stable, more efficient, and less prone to overfitting during training, while still maintaining the ability to operate in continuous action spaces.

2.5 DRL-based flight

DRL based flight is a research area that is currently under active research. One such work by Shadeed et al. [10] develops a DRL method for aggressive flight manoeuvres. A path is first generated that is known to not exceed the given dynamical limitations of the vehicle. Pitch and roll references are then generated by an AI algorithm and given to a low-level proportional-integral controller for attitude control. To do this, the authors implemented Proximal Policy Optimisation (PPO), another type of DRL algorithm that can operate on continuous action spaces. The algorithm was trained by being given a target trajectory and comparing how closely it was able to follow the path using root mean-squared error. The results of this work show that the trained robot was able to accurately follow the desired path with maximum acceleration values of 6.2 m/s and a target velocity error of only 0.16m/s.
In the algorithm developed in this report, TD3 is used instead of PPO and the target trajectory and attitude is fixed. Furthermore, in contrast to Shadeed et al., the algorithm produced in this work directly generates motor commands, therefore combining both the trajectory tracking and low-level controllers.

3 Methodology

This section will first introduce a formal definition for the problem of autonomous flight and will then propose a solution to this problem using TD3.

3.1 Problem Definition

The problem of autonomous flight can be described as a single robot, R, maintaining a given attitude $\mathcal{G}$ while not being in contact with another object or surface. The robot is equipped with sensors capable of determining its current state. This includes the attitude, A and velocity, V, of the robot. The objective is to maintain the given attitude

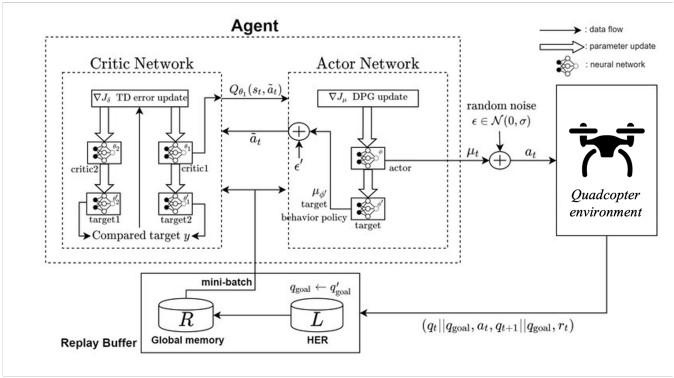


Figure 1: Architecture of the TD3 network integrated into the quadcopter's simulated environment [6].

without translation ($V = 0$) by varying rotor speeds to adjust the robot's current physical state, thus minimising the function:

$$\min \left[\sum_{j=1}^{h} ||\mathcal{G} - A_j|| + V_j \right]$$

where h denotes the total number of time-steps in an episode.

3.2 Architecture

The architecture of this work is an implementation of TD3 and is shown in figure 1.
The network structure varies during experimentation in the number of layers and the number of neurons in each layer, but generally, there are between 5 and 6 layers with the hidden layers having between 128 and 512 neurons each. Each model consists of six different networks; the target and core networks both contain two critic networks and one actor network each.
The network receives continuous (non-discrete) values as input from the sensors in the simulated drone. These values are the current attitude and the linear and angular velocities of the quadcopter. They are encoded and interpreted by the model to produce four output values that represents the RPM of each of the quadcopter's motors. This information is then passed to the simulator for execution.

Reward Function A reward function is required to inform the robot how optimal the current state is. Intuitively, the higher the reward, the better the state. For this work the following reward function is used:

$$r_i = \begin{cases} -(||\mathcal{G} - A|| + V), & \text{each t} \\ -1000, & \text{on collision} \end{cases}$$

where $\mathcal{G}$ is the target attitude $[0, 0, 1]$ (a vector pointing vertically upwards).
This function incentivises the robot to remain as close as possible to the target vector for the duration of the episode. States where the robot's attitude is further from the target vector are punished much more severely. The environment is considered *solved* when the robot scores 0 over the course of an entire episode, i.e. it does not defer from the target attitude.l

Experience Replay During execution in the training environment, the robot receives an observation about its environment. This observation, the selected action and reward received at each time step is stored in an experience replay buffer. Thus a robot experience is defined by the tuple $< s_i, a_i, s'_i, r_i >$. A mini-batch of experiences is uniformly sampled from the experience replay to remove the temporal correlation between training samples during policy updates as proposed in [3].

4 Implementation

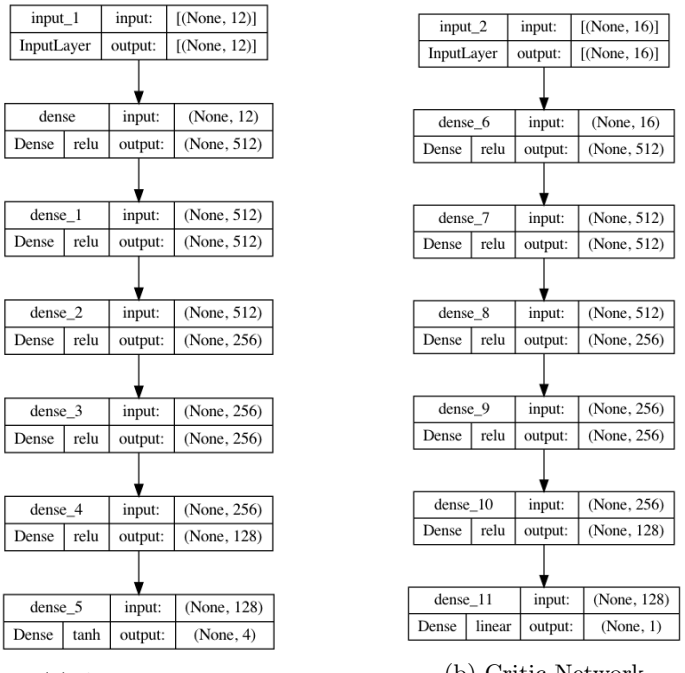
This section describes how the described architecture is implemented. The pseudocode of the TD3 algorithm is given, along with the network architecture used. For the implementation, a training and testing environment was first created. Then, an autonomous agent that implemented TD3 was created using TensorFlow and Keras.

4.1 Environment

To train the agent, a environment must first be created that passes the relevant information to the robot and can handle the commands sent by the agent in response. This is the implementation of the MDP model described in section 2 of this work. The environment was created using PyBullet, a python-based robotics simulator. In the environment, the robot has full range of movement and collisions are accurately modelled. Furthermore, the data about the state of the robot can be easily passed to the agent as an observation. Noise was also be applied to the sensor readings so that the simulation more accurately modelled real-world behaviours.

4.2 Twin Delayed DDPG Implementation

TD3 was implemented directly from the algorithm described by Fujimoto et al [4]. Six networks were created to reflect the policy and target actor and critics. The actor network is shown in figure 2a, and critic network is shown in figure 2b. Note that both critics use the same architecture.



(a) Actor Network

(b) Critic Network

Figure 2: The implemented architecture of the actor and critic neural networks.

The use of convolutional layers was not applicable here since all the sensor data can be encoded by the fully connected layers of the network. Furthermore, dropout layers were not used since they increase variance in the training data. The large amounts of variance seen in the generated experiences already tends to be a major issue for learning stability, so the use of dropout layers would have a negative effect on the algorithm.

To implement the TD3 algorithm, the psuedocode in algorithm 1 was used and a solution was written using Python. At each step, the four outputs of the actor network were passed to the environment for action execution. An updated state and respective reward value was then returned back to the agent. The combination of the action, previous state, new state and reward value was stored as an experience in the agent's replay memory. Training was then conducted on a random sample of this memory after every episode.

Algorithm 1 Twin Delayed DDPG (TD3) [4]

Initialise critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialise target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialise replay buffer $\mathcal{B}$
for each $t \in T$ **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon, \epsilon \sim \mathcal{N}(0, \delta)$ and observe reward r and new state s'
  store transition tuple (s, a, r, s') in $\mathcal{B}$
  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \epsilon \sim clip(\mathcal{N}(0, \tilde{\delta}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics: $\theta_i \leftarrow argmin_{\theta_i} \frac{1}{N} \sum (y - Q_{\theta_i}(s, a))^2$
  if $t \mod d == 0$ **then**
    Update ϕ by the deterministic policy gradient:
    $\Delta_\phi J(\phi) = \frac{1}{N} \sum \delta_a Q_{\theta_1}(s, a)|_{\pi_\phi(s)} \Delta_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi'_i \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

5 Experimentation

This section describes, interprets and assesses the results of hyperparameter testing in simulations. The experimental method is first described and then the results of the experiments are analysed.

5.1 Hyperparameter Exploration Approach

There are several key hyperparameters that can be modified in the TD3 algorithm: the learning rate γ, the target network update bias τ and the layer count & neuron size. During hyperparameter exploration, one of these values will be changed, and the rest kept constant. The network was then trained over 120,000 episodes. After training, validation was performed in a new copy of the training environment. This is shown in figure 3.

.

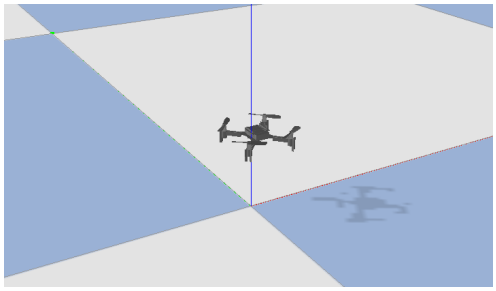
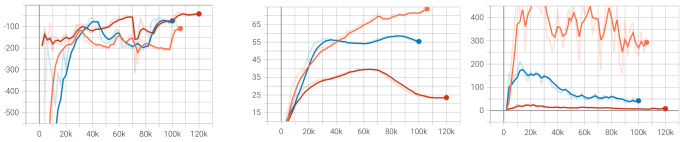


Figure 3: Screenshot of the running pybullet simulator

5.2 Results and Analysis

5.2.1 Modifying Network Topology

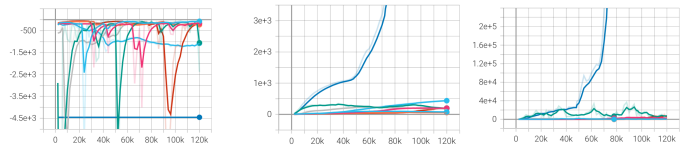


(a) Reward value (b) Actor loss (c) Critic loss

The three network topologies used are (a) [512, 512, 256, 128], (b) [512, 256, 128], and (c) [256, 256, 128], where the values represent the number of neurons in each layer. These are depicted by the orange, blue and red lines respectively in the above graphs. Immediately, it is visible that algorithm under topology a has significant variance in the critic loss. It is likely that this is a result of the learning rate not being appropriate for the larger network structure. The lack of stability in the critic loss is reflected in the mean reward value; topology a is able to reach a mean reward that is more than 100 points lower than topology c.
When comparing the smaller networks, b and c, it appears as though topology c, which has half of the number of neurons in the first layer, reaches a better, stable performance faster. Although topology b initially receives a high reward value, it immediately drops and approaches the ability of topology c after 20,000 episodes. Although c performed better after training, there were still significant drops in performance during training in both topologies. This is likely due to the lack of learning rate decay, and could be an aspect to explore in future works.
The trend appears to be that the smaller networks perform better; the final mean reward value for topology b and c are -73 and -39 respectively. Therefore, it appears as though a smaller network is more appropriate in this use case. However, due to the time constraints of this work, it is unclear whether a larger network would perform better if trained for longer periods of time.

5.2.2 Modifying Learning Rate

Experimentation on the learning rate was performed on all three of the network topologies from the previous section to find if a given topology would perform better under a different learning rate.

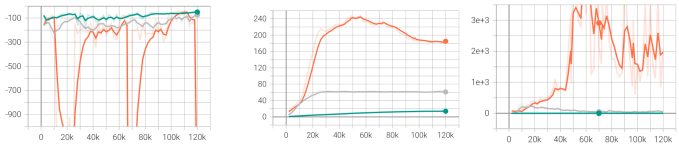


(a) Reward value (b) Actor loss (c) Critic loss

Generally, the larger network topologies performed better with smaller learning rates, whilst the inverse is true for smaller networks. When the learning rate is too large, there are significant drops in the mean reward value during training. This is seen most prominently with the light blue and red lines which refer to topology a with $\gamma = 0.1 \times 10^{-3}$ topology c with $\gamma = 0.3 \times 10^{-3}$. Regardless, when the learning rate is too high, like in the case of the dark blue line, the algorithm is unable to learn a solution and the reward value was pinned to around -4400. This occurred on topology b with $\gamma = 0.5 \times 10^{-3}$.
The blue line also suffered from an exponentially growing loss value for both the actor and critics. It is evident that, under this configuration, the algorithm would never recover. If the other values of γ are considered, there is a general trend that the algorithm will converge, with the larger networks still performing worse than the smaller ones. This further enforces the belief that a smaller network is more appropriate here.

5.2.3 Modifying Target Network Update Bias

By modifying the target network update bias τ, the latest experiences seen during training will have a stronger or weaker effect on the algorithm. During this experiment, the network used three layers with 256, 256 and 128 neurons respectively, with a learning rate of 0.3×10^{-3}.



(a) Reward value (b) Actor loss (c) Critic loss

In the graphs above, the teal, grey and orange lines use values of 0.01, 0.025 and 0.075 respectively. In graph a, three large downwards spikes can be seen with $\tau = 0.075$. This correlates to the period in which target updating is performed. Due to the high value of τ, there is a significant change in the network weights during update which leads to a drastically worse performance of over 900 points while the network recovers.
Whats more, the loss value for the actor and critics appear to be very sensitive to τ. When using $\tau = 0.075$, both losses dramatically increase. Even changing the value of τ between 0.01 and 0.025 has severe negative consequences. With $\tau = 0.01$, actor loss never goes above 15, and critic loss never goes above 2. In contrast, with a value of 0.025, the actor loss sharply rises to 62 and maintains this value throughout training, while the critic loss rises to 320 and slowly descends to a minima of 44. It is interesting to note that with $\tau = 0.025$ the actor loss maintains a value of 62 whilst the critic loss drops. This may be because of the use of noise applied to the actor in TD3 which, when also considering the learning rate, results in the actor not being able to exploit the environment, and thus not being able to find patterns that result in higher cumulative rewards.

6 Conclusion and Future Works

In conclusion, the results of this work show that a robot is capable of learning control at a low level in order to track a given vector with little error. Future simulations of this work may involve the use of a convolutional neural network topology being placed below the fully connected layers so that the robot can have a broader understanding of its environment. In addition to this, the topology of the network may not be appropriate for following a path, as opposed to a stationary vector. In this case, additional work would have to be performed to analyse the optimal topology for the adapted use-case.

References

[1] R. BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[2] S. Ben-David, E. Kushilevitz, and Y. Mansour. Online learning versus offline learning. *Machine Learning*, 29(1):45–63, Oct 1997.

[3] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.

[4] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[5] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[6] M. Kim, D.-K. Han, J.-H. Park, and J.-S. Kim. Motion planning of robot manipulators for a smoother path using a twin delayed deep deterministic policy gradient with hindsight experience replay. *Applied Sciences*, 10:575, 01 2020.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2015.

[8] Z. Ma, J. Hu, Y. Niu, and H. Yu. *Reactive Obstacle Avoidance Method for a UAV*, pages 83–108. Springer International Publishing, Cham, 2021.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. 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*, doi:10.1038/nature14236, 2015.

[10] O. Shadeed, M. Hasanzade, and E. Koyuncu. Deep reinforcement learning based aggressive flight trajectory tracker. 01 2021.