JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 37, 1165-1176 (2021)
DOI: 10.6688/JISE.202109_37(5).0012

The Use of Deep Reinforcement Learning for Flying a Drone

SANDRO DOMITRAN AND MARINA BAGIĆ BABAC
Faculty of Electrical Engineering and Computing
University of Zagreb
Zagreb, HR-10000 Croatia
E-mail: {sandro.domitran; marina.bagic}@fer.hr

Nowadays, unmanned aerial vehicles, commonly known as drones, are used for many different purposes. However, it is still a challenging task to fly a drone, which limits its potential for doing more useful things. This paper shows how to design, develop and test an ML-Agent simulation environment by using the Unity engine and the deep reinforcement learning algorithm. First, the drone model needs to be imported in a simulating environment where it should have an ability to fly, and then it should be made to fly using deep reinforcement learning. In addition, the drone can learn to perform a certain task to elaborate the benefits of this approach.

Keywords: unmanned aerial vehicle, flight control, VTOL Tailsitter UAV, machine learning, deep reinforcement learning

1. INTRODUCTION

Over the last couple of years, UAVs (*Unmanned Aerial Vehicle*), or drones are used in many different areas including scientific research, agriculture, search and rescue, filmmaking, and so on [1]. Recent studies have used machine learning techniques to provide solutions for the various problems that have already been identified when UAVs are used for communication purposes. Applying machine learning to UAVs does not only reduce the factor of human error but makes it fly more accurately, efficiently, and safely [2]. However, the process of learning requires a lot of trial and errors because the subject needs to gain experience, so in case of drones this process could be hard and long lasting in the real world. Thus, there is a need to make a simulation in which the problem of learning could be solved easier as well as faster.

In this paper, we have shown a simulation of an environment in which the drone is able to learn how to fly. The simulation of the drone's flight is made in the Unity engine [3] with the process of deep reinforcement learning [4]. Our goal was to make the drone intelligent enough to fly from one place to another, so we could observe how it performs.

2. THEORETICAL FRAMEWORK

Deep reinforcement learning is a combination of artificial neural networks, which have the power to represent and comprehend the environment, and reinforcement learning, with the ability to act on that understanding [5]. The goal of reinforcement learning is to develop a policy that maps agent's states to probabilities of selecting each possible action, and improving it so that is optimal, *i.e.* maximizes the expected reward [4]. Here, the agent

Received March 16, 2020; revised June 25, 2020 & January 6& 25, 2021; accepted January 26, 2021.
Communicated by Chin Ying Liew.

is an entity which takes decisions based on the rewards and punishments directing its activity towards achieving goals and is implemented using the ML-Agent Toolkit [3].

For the task of teaching a drone to fly, we have chosen a *policy-based* reinforcement learning algorithm [6] by which the agent directly learns the policy and acts based on that policy. We have defined a policy as the agent's strategy, and our model of a policy is a neural network which takes observations of the world as an input, an action as an output, and whose weights are the policy parameters. This approach for training the network is called the *policy gradient*, where the policy parameters are updated approximately proportional to the gradient:

$$\Delta\theta \approx \alpha \frac{\partial\rho}{\partial\theta} \tag{1}$$

where θ is the vector of policy parameters, ρ is the performance of the corresponding policy (*e.g.* an average reward per step), and α is a positive-definite step size [7].

More specifically, we have chosen a policy gradient method called *Proximal Policy Optimization* (PPO), which is an improvement of *Trust Region Policy Optimization* (TRPO) in terms of simplicity and computing efficiency [8]. The basic idea of policy gradient methods is applying a computed policy gradient estimator to the stochastic gradient ascent algorithm. The policy gradient estimator $L^{PG}(\theta)$ is defined as follows:

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t \mid s_t)\hat{A}_t] \tag{2}$$

where π_θ is the policy, that is, a neural network that has the observations from the environment as an input and an action to be taken as an output, $\hat{A}_t$ is the estimator of the advantage function at timestep t and $\hat{E}_t$ is the expectation. $\hat{A}_t$ is defined as the difference between the discounted sum of rewards and the baseline estimate [9]. The discounted sum of rewards can be written as the weighted sum of all the rewards the agent got in the current episode:

$$\sum_{k=0}^{\infty} \gamma^k r_t + k. \tag{3}$$

Factor γ, also known as the discount factor, is used for making future rewards less valuable because we want our agent to care more about sooner rewards. It is important to note that $\hat{A}_t$ is calculated after the whole episode is done, so all the rewards r_t that the agent got are known. The baseline estimate is basically a value function that gives an estimated sum of rewards from this point onward, it guesses what the final reward is going to be in this episode starting from current state. The advantage function estimate is the measure of how much better the action was based on the expectation of what would normally happen in the state that agent was in. By differentiating the objective function, $L^{PG}(\theta)$, we get the estimator. If the obtained value is positive, which means that sequence of actions the agent took resulted in a better than average return, it will increase the probability of selecting it again. The problem occurs if we keep running the gradient descent without setting any limitations on updating our policy because the advantage function is noisy, and it could make too big of an update based on a single batch of collected experience. That would destroy our policy, so we need to limit the update and that is what TRPO does [4]. Both TRPO and PPO are using constraint on policy update, but the major difference is that PPO

includes this extra constraint directly into the objective function while TRPO computes it separately. The central objective function of PPO is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\ \varepsilon, 1+\varepsilon)\hat{A}_t)] \quad (4)$$

where $r_t(\theta)$ is:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}. \quad (5)$$

When $r_t(\theta)$ is greater than 1, then the action is more likely than it was in the old policy. If it is between 0 and 1, it is less likely than before. The first argument in *min()* function is the policy gradient objective which pushes the policy towards actions that return high positive rewards. The second argument is the *clip()* operator which ensures that the probability ratio r is between $1-\varepsilon$ and $1+\varepsilon$ to limit the effect of the gradient update. Then, we take the minimum of both the clipped and the unclipped objective and we get a lower (pessimistic) bound on the unclipped objective [9].

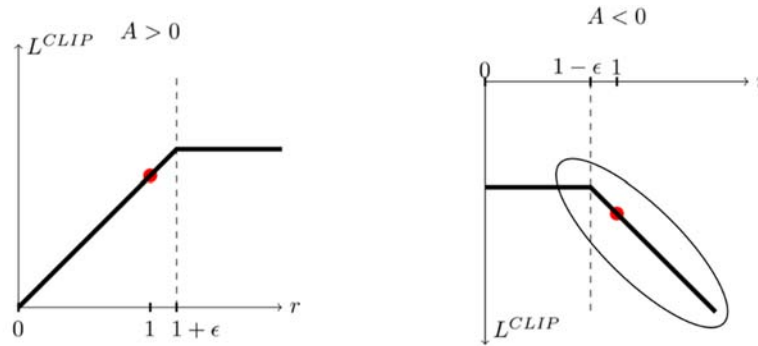


Fig. 1. Single term in $L^{CLIP}(\theta)$ [9].

From Fig. 1, which depicts a single term plotted in $L^{CLIP}(\theta)$, it can be noted how the probability ratio r is clipped at $1-\ \varepsilon$ if the advantage function is negative, and at $1+\varepsilon$ if is positive. The behaviour of the objective function in the circled region can also be noticed. Even though the selected action is more probable, our policy is getting worse because the advantage function is negative. That is why we want to undo this action, and that is what our objective function allows us to do. The final loss function that is used to train our agent is given as follows:

$$L_t^{PPO}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]. \quad (6)$$

The first term in the expression is the central objective function that we described. In the second term, we are updating the baseline network which is actually our value function which tells us how good it is to be in the current state. The third term is called the entropy bonus, it tells us how unpredictable this outcome is, and it ensures that the agent is doing enough exploration [9].

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Fig. 2. PPO algorithm [9].

Fig. 2 shows how the PPO algorithm works; the piece of code labelled with the red rectangle represents the first thread, and the one labelled with the green rectangle represents the second thread. In the first thread, each of *N* (parallel) agents collect *T* time-steps of data based on the current policy, and immediately compute advantage estimates. Once they are done, a second thread collects all that experience and runs the gradient descent on the policy network using the PPO objective. This process repeats until the policy is no longer changing or until we are satisfied with our agent's behaviour [7].

3. IMPLEMENTATION AND RESULTS

3.1 Experimental Setup

For creating the environment in which the agent is training, the Unity [3] is used, a cross-platform game engine primarily used to develop video games and simulations. For the purposes of the training process, the ML-Agents Toolkit is used, an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents, as well as the TensorFlow, an open-source library that serves for performing computations using deep learning models.

When using ML-Agents, the focus is on modelling our problem in terms of rewards and behaviour of the agent. The ML-Agents workflow has three high-level steps: creating our environment, training of the brains in that environment and embedding those trained brains back to Unity. In ML-Agents, there are three basic concepts introduced: the agent, the brain and the academy. The academy enables everything to work together and it allows Unity to communicate with TensorFlow. The brain is a neural network that makes decisions. The agents take decisions from the brain and act based on them in the environment. At any given step of the simulation, each agent observes the world and inputs those observations into the brain which tells him what to do next. Each brain can control multiple agents and all of them can train the same brain [5].

The process starts by defining the agent for the problem we are trying to solve, that is, making an autonomous tailsitter that should be able to fly to a certain target. Thus, the tailsitter is going to be the agent because we want to teach it how to fly and see how it performs. After reaching the target, it would get a new target, *etc.* until the simulation is complete. Each new target is actually a new task, so the number of targets does not actually affect the performance of the task. The first step would be modelling the environment in

which the agent learns and enabling it to do the actions, which will give it the ability of flying. Before training the agent, we want to be able to control it manually to make sure everything works fine before we start with the training.

After a 3D model of a tailsitter is made, which is basically an airplane model with two rotors and two elevators, the ground limits the agent's movement and the sphere that we use as a target. Then, the colliders are added, and the force is applied based on how long the button is pressed. There could be two buttons, one for each rotor, for increasing force while the button is pressed and decreasing while the button is released. The rotors will have some maximum force and a minimum force of zero. Also, four more buttons are needed, two for each elevator, which determine the direction of elevator rotation that directly affects the amount of force applied. While force from the rotors is always applied upwards (with reference to the tailsitter), the force from the elevators could be applied forwards or backwards. After applying this to our model in Unity, it is now capable of flying and we need to get it ready for learning.

Based on our problem, we need to define the observation vector, the action vector and the reward system. Since we want to make our agent fly, our objective is flying from target to target. The action vector is the same, but instead of taking actions based on pressed buttons, our agent's brain decides which actions to take, which makes our agent autonomous. In our observation vector, we add all the important information that describes the current state of our agent in the best way, that is, the distance from the target, the velocity and the angular velocity of the agent. Although sufficient for our goal, this is not a complete list of parameters that a control policy needs to consider and it depends upon specific goals. For example, if we wanted to reduce the battery consumption of the tailsitter, we would add a battery that would decrease with respect to the amount of force generated by the propellers, thus the battery consumption would be one of the parameters.

The observation vector is the input to our neural network and the action vector is the output. In addition, we are using the reward shaping system that is based on several functions. Each of them has a different purpose and meaning. First one is the reward function for the distance from the target. In order to make our agent fly to the target, we want to punish it more the farther it is from the target. This function is described by the following expression:

$$f(x) = k \cdot (e^{-ax} - c). \tag{7}$$

Here, x is the square distance to the target, a is some positive constant that changes the gradient of the reward function, and c is a constant which moves the function up or down. The k parameter is the scaling of the reward that depends on other rewards in the environment. The smaller the angular velocity is, the agent is more stable, so we use the same function as for the distance from the target. The only differences are that here, the x parameter is the angular velocity and we choose different values for the a, c and k, but they still have the same meaning. It is important to note that there are no exact values for a, c and k, as they are problem dependent, but it is possible to pick the wrong values for these parameters, which could lead to an unstable training process, and wrong solutions. These two functions are executed every step of the training process and their output agent gets as a reward.

We also want our agent to spend as much time in the air as possible and every time it falls on the ground it needs to get punished based on the time spent in air. The function used for that purpose is given with the following expression:

$$g(x) = \left(\frac{-c}{\max Step}\right) \cdot (\max Step - x). \tag{8}$$

Here, the x is the number of steps that defines how long the agent has been flying, the *maxStep* is the maximum number of steps before the automatic reset of the environment and c is the positive constant used for scaling the reward. This function is executed once in the episode and only if the agent hits the ground or goes out of boundaries. That also means that it should have a larger absolute value because we want it to be measurable with all the rewards we get during the episode. The only thing left is the reward for picking the target which is a positive constant. It should be large enough for motivating the agent to pick it, because picking it means that the reward is spawned at some random position which results in increasing the distance between them. Without this, the agent could, for example, stay around the target as close as possible without touching it, which makes it able to avoid spawning the target again.

3.2 Parameter Estimation

For a simulation, a set of hyperparameters for the training process needs to be adjusted. The lower the γ parameter is, which is the discount factor for future rewards, the agent cares less about the rewards it gets in the future. Typical range of this parameter is between 0.8 and 0.995. By lowering the ε parameter, which is used as a clipping boundary, the training process is slower but more stable. The ε is usually between 0.9 and 0.95. We can think of the λ parameter as how much the agent relies on its current value estimate when calculating an updated value estimate. When the value of λ is low, it means that it relies more on current value estimate, and when it is high, it relies more on actual received rewards. It should be between 0.1 and 0.3. The buffer size parameter corresponds to the number of experiences that should be collected before doing any update of the model. A large buffer size parameter can lead to more stable, but also slower, training updates. Similarly, the batch size parameter represents the number of experiences used for one iteration of the gradient descent update. The buffer size should be multiple of the batch size, and its typical range is between 2,048 and 409,600, while the batch size parameter in continuous action space should be between 512 and 5,120. The learning rate parameter is the size of each gradient descent step. The larger this value is, the training process is faster, but if this value is too large it could lead to unstable training. The typical range of the learning rate is between 0.00001 and 0.001. The β parameter makes the agent properly explore the action space by making the policy more random, with typical range between 0.0001 and 0.01.

3.3 Simulation Results

The training process has been run a few hundred times with various combinations of different hyperparameters and reward function parameters, so only the most successful results are shown here. The training is run on multiple agents at once and all of them are training the same brain, so the graphs show the average values.

The graph in Fig. 3 shows the environment cumulative reward, which is the main indicator of our agent's success, provided that the reward system has been set properly. This graph plots the mean episode reward over all the agents through time. It can be noted that in the following graphs, time is always shown in steps, where a step is defined as one cycle of the episode.

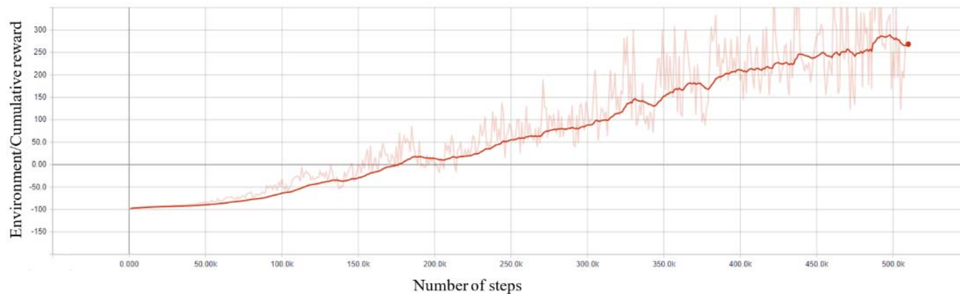


Fig. 3. Environment/Cumulative reward with respect to number of control steps (up to 500,000).

Our agent's goal is to maximize the reward it gets, so accordingly, the reward increases while learning and it tends to stabilize. It can be observed from Fig. 3 that the reward value after 500,000 steps increased from −100 to 300. This value would keep growing if the training was continued, and we could not know exactly how long it would last until it completely stabilizes. The graph in Fig. 4 shows the same training process after 5 million steps. It can be noted that after 5 million steps, the reward is still increasing, which means that our agent is still able to learn.

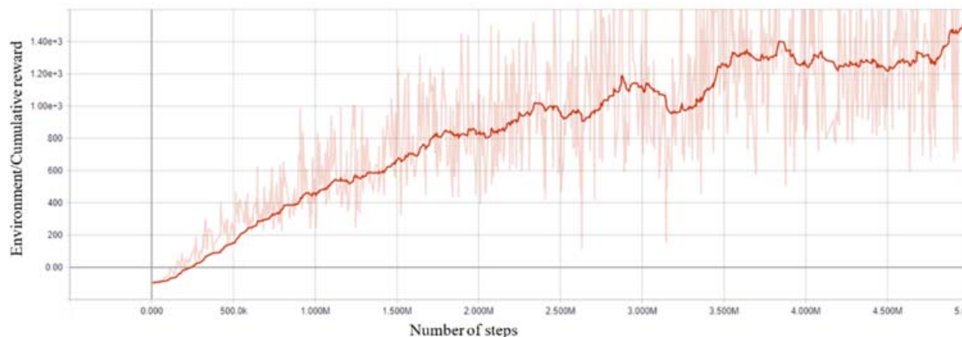


Fig. 4. Environment/Cumulative reward with respect to number of control steps (up to 5 million).

The graph in Fig. 5 shows the episode length through time (until 500,000 steps), which tells us how long on average the agent was 'alive' before it went to the terminal state. This graph should increase or decrease while the agent is learning, which depends on how our problem is defined. In our case, we want our agent to fly as long as possible, so the episode length should increase.

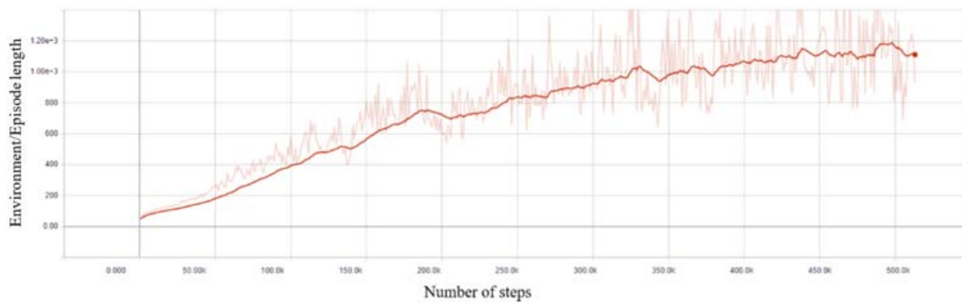


Fig. 5. Environment/Episode length with respect to number of control steps.

The next two graphs give us the information on how our policy is doing through the training process. The first of them (Fig. 6) is the policy entropy which shows us how the entropy of the policy changes through time (until 500,000 steps). At the beginning, the agent takes random actions. While the agent is learning, this randomness should slowly decrease, which means that the agent takes more actions based on its policy.

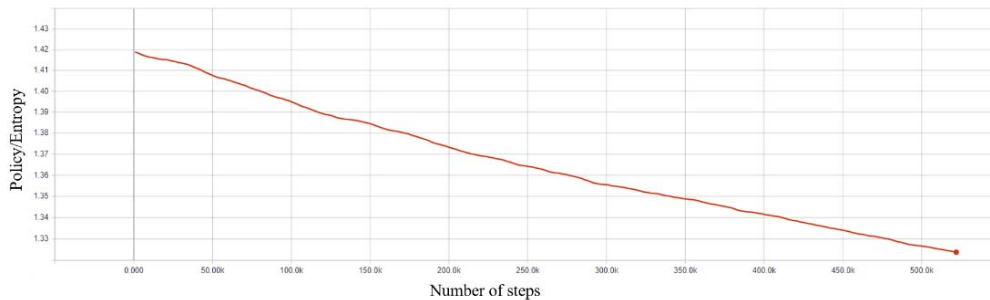


Fig. 6. Policy/Entropy with respect to number of control steps.

Fig. 7. Losses/Policy loss with respect to number of control steps.

Then, the graph in Fig. 7 depicts the mean magnitude of the policy loss function (after 500,000 steps). This corresponds to how much the policy is changing, so during the successful training process the magnitude of this graph should decrease. It should decrease because that would indicate our policy is doing more stable updates as the training passes.

These results tell us that the training process is being successful, and by looking at the simulation, we could see that the tailsitter is now able to fly very quickly and accurately from target to target. It is important to notice that some changes in the training process would change the flight behaviour in some way, but it would still probably be a good solution as well. For example, we could make the tailsitter flight more realistic by constraining how its rotor power changes. Before this modification, our rotor power could change from zero to maximum in just one step what is impossible in the real world. Instead, we can limit the change of the rotor in one step with some constant what will make the whole flight more realistic. Thus, Figs. 8-11 show the graphs generated by training a more realistic model (that better represents real-world conditions, that is achieved by changing the environmental model in addition to the change of the rotor) in comparison to the above graphs. The red line in the figures indicates instantaneous change in rotor angular velocity (*instantaneous*) and the grey line indicates delayed change in rotor angular velocity (*delayed*). It can be noted that the new (more realistic) model is learning much slower, but also more stable because, in every step, the agent has less possible moves than before, so it is easier to pick the right action. As expected, the new model is slower and less accurate, so it gets less rewarded comparing to the old, less realistic model.

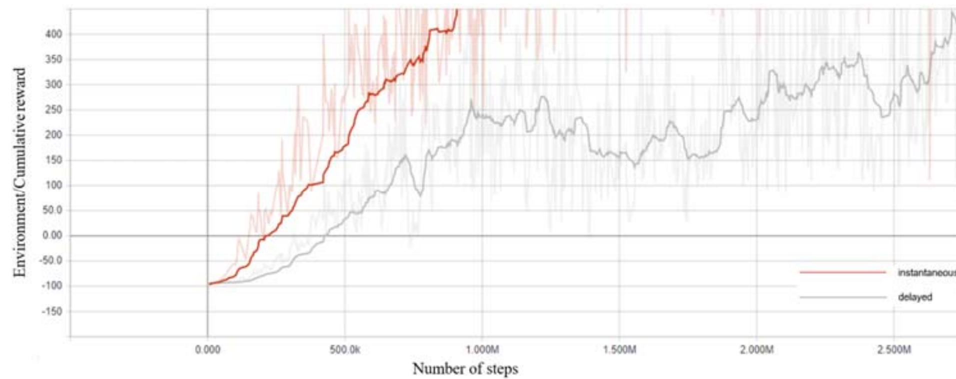


Fig. 8. Environment/Cumulative reward with respect to number of control steps.

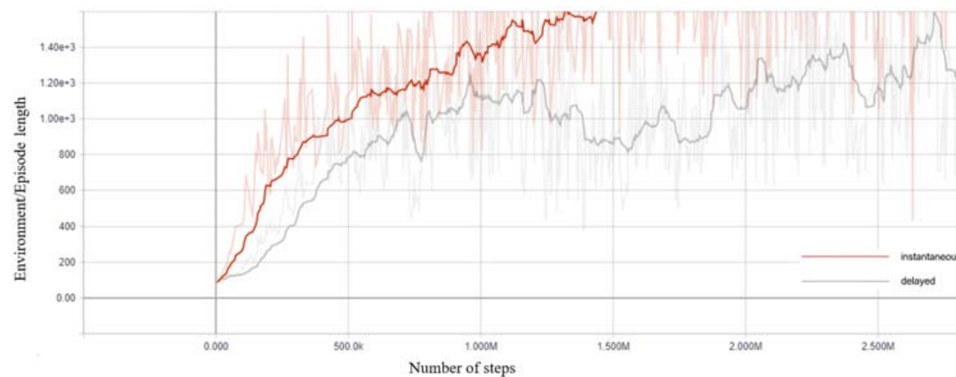


Fig. 9. Environment/Episode length with respect to number of control steps.

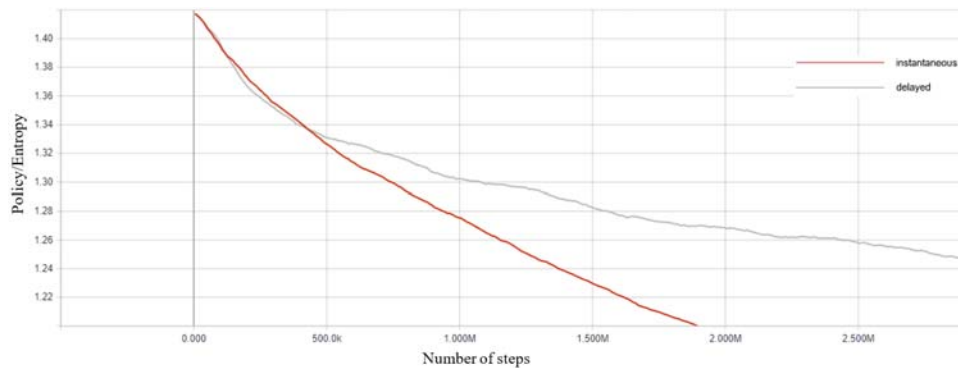


Fig. 10. Policy/Entropy with respect to number of control steps.

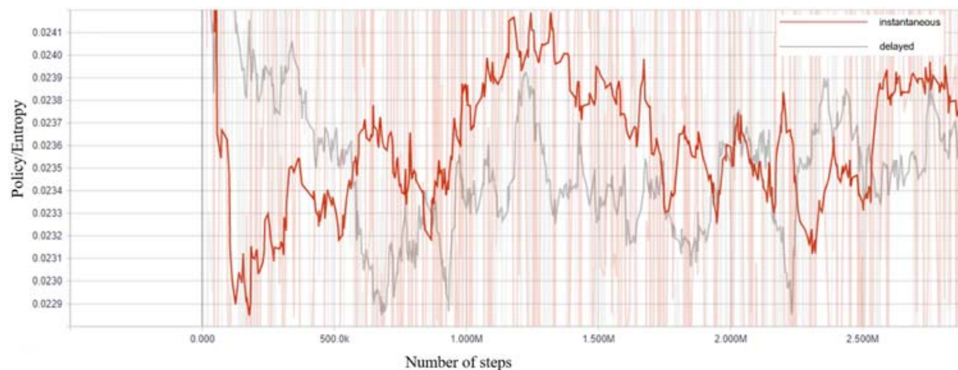


Fig. 11. Losses/Policy loss with respect to number of control steps.

3.4 Discussion

Enabling a drone to autonomously fly through highly unstructured environments represents a challenging task [10]. Current methods based on deep learning have made a progress in this direction, but still many technical and theoretical questions remain open [11]. The difficulty and feasibility of the learning process depends mostly on the goals we want to achieve, and once we define a model, the goal functions will vary depending on the vehicle. More specifically, the target functions determine the rewards and penalties for the agent, and we used the penalty for distance to the target, penalty for the angular velocity, penalty/reward for the time spent in the air, and reward for collecting target. For this specific task, these functions would be almost the same for any type of aircraft (airplane, helicopter, quadcopter, *etc.*), and what might be different is the observation vector and the action vector because the vehicles differ in the way they move and steer. If we wanted to do the same thing for another type of vehicle, for example, a car, in addition to the observation vector and the action vector, we would have to change the functions of the target because those are specific to aircraft.

Most tailsitter control approaches rely on a mathematical model of the tailsitter and its dynamics, which are nonlinear and may carry inaccuracies due to the impossibility of modelling all aspects of the vehicle's dynamics [6]. An alternative to these control tech-

niques can be obtained with intelligent controllers, developed through machine learning and optimization methods, such as state-of-the-art reinforcement learning, applied in continuous tasks [12].

Although deterministic policy gradients have certain advantages over stochastic policy gradients, such as value/advantage estimations with lower variance, they require a good exploration strategy to explore its state space efficiently [7]. Hence, stochastic policy gradients can present a better sample efficiency, which has a direct impact on the number of timesteps or episodes needed for control policy convergence [13].

The choice of observation vector that describes the current state of our agent proved to be significant for its rate of improvement during training and its final performance [7]. Therefore, other possible combinations of these observations can also be investigated to describe the current state of the agent in the best way. Another way the designer can influence and direct the behaviour of the agent is the choice of the reward function. However, analysing different choices of the reward function was not given much focus here as the original choice gave satisfying results.

It should be noted that the whole training, experiments and reward structures can be designed to facilitate learning of more advanced behavior, tighter control or better robustness.

4. CONCLUSIONS

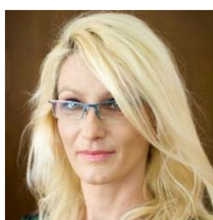
This paper has shown how the tailsitter flies by using deep reinforcement learning, based on the proximal policy optimization method. A simulation of the tailsitter in the Unity Engine is explained and, the model in trained to learn to fly from target to target, by using the ML-Agents Toolkit. The obtained results demonstrated that the model was successfully trained, and it was able to solve the problem well. Although this is a simplification of the actual physics behind the tailsitter flight, it can be assumed that it is applicable to the real tailsitter by making a simulation more realistic, until the gap between the simulation and the real world is so small that we could transfer it directly to the real world. After the transition to the real word, the tailsitter would need to be trained in these new conditions, and after some adjustments, we would have the intelligent drone. The machine learning could not only be used in controlling the drone's flight, but also in making it capable of solving some more difficult problems.

REFERENCES

1. A. Loquercio and D. Scaramuzza, "Learning to control drones in natural environments: A survey," in *Workshop on Perception*, *Inference*, *and Learning for Joint Semantic*, *Geometric*, *and Physical Understanding*, 2018.
2. R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*, Princeton University Press, Woodstock, UK, 2012.
3. A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv Preprint*, 2019, arXiv:1809.02627.
4. M. A. Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeep learning.com/, 2015.

5. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Vol. 1, MIT Press Cambridge, MA, 1998,
6. E. Bøhn, E. M. Coates, S. Moe, and T. A. Johansen, "Deep reinforcement learning attitude control of fixed-wing UAVs using proximal policy optimization," in *Proceedings of International Conference on Unmanned Aircraft Systems*, 2019, pp. 523-533.
7. G. C. Lopes, M. Ferreira, A. da S. Simões, and E. L. Colombini, "Intelligent control of a quadrotor with proximal policy optimization reinforcement learning," in *Proceedings of Latin American Robotic Symposium; Brazilian Symposium on Robotics; Workshop on Robotics in Education*, 2018, pp. 503-508.
8. S.-I. Amari, "Natural gradient works efficiently in learning," *Neural Computation*, Vol. 10, 1998, pp. 251-276.
9. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "OpenAI: Proximal policy optimization algorithms," *arXiv Preprint*, 2017, arXiv:1707.06347.
10. J. Hill, and A. Rogers, *The Rise of the Drones: From The Great War to Gaza*, Vancouver Island University Arts & Humanities Colloquium Series, 2014.
11. V. François-Lavet, P. Henderson, R. Islam, Marc G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv Preprint*, 2018, arXiv:1811.12560.
12. A. Loquercio, A. I. Maqueda, C. R. Del Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," *IEEE Robotics and Automation Letters*, 2018, pp. 1088-1095.
13. R. S. Sutton, D. Mcallester, S. Singh, Satinder, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems*, 2019, pp. 1057-1063.

Sandro Domitran received the B.S. degree in Computer Science from the University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia, where he is currently pursuing the M.S. degree. His research interests include unmanned autonomous vehicles and artificial intelligence applications.



Marina Bagić Babac is an Assistant Professor at Department of Applied Computing at the University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia. She received the B.Sc., M.Sc. and Ph.D. degrees in Electrical Engineering from the University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia, and M.Sc. in Journalism from the University of Zagreb, Faculty of Political Science. Her research interests include machine learning, statistical network science and natural language processing.