# Using Monte-Carlo Reinforcement Learning to Train a Drone Simulation within a 2D PyGame Environment

Tyler Rolfe, Benjamin Atkinson, and James Marshall

Drones have become increasingly important in a variety of fields, including logistics, military applications and agriculture. This paper explores the use of Monte Carlo reinforcement learning (RL) to train a drone to navigate towards and hit a sequence of targets in a 2D PyGame environment. The drone is trained using an on-policy, first-visit Monte Carlo control method with epsilon-greedy exploration. The results show that the Monte Carlo method successfully converges to a local optimum, enabling the drone to reach one of the targets, but struggles to consistently find the others. The study also explores potential improvements, including expanding the state space and reward functions, which could be facilitated with increased computational resources. The project's GitHub repository can be found by clicking **HERE**.

***Introduction.***– Drones have emerged as a transformative technology across various domains. Large corporations, such as Amazon and DHL, are actively developing delivery drones to enhance speed and efficiency in transportation of consumer goods [1]. In defence, the ongoing war in Ukraine has become the first large-scale conflict where both sides have extensively deployed drones for use as weapons and surveillance [2]. Additionally, drones are being used for innovative solutions to problems in agriculture, such as pollinating crops [3]. Governments around the world are recognising the strategic importance of drones, with the US announcing a $1 billion investment, evenly split across 2024 and 2025, in the Replicator program which aims to field thousands of drones [4], and the UK committing to £4.5 billion of investment over the next decade, from 2024, for UAVs for the military [5]. These substantial investments highlight the financial and strategic incentives for developing drone technologies.

The control of a drone is programmed into its flight controller. Flight controllers that respond to user input are limited by human error and the ability to send signals to the drone. Instead, drones typically require a substantial level of autonomy so that they can accomplish planned missions in unexpected situations without human intervention [6]. One approach to autonomous drone flight is to program the flight controller using reinforcement learning (RL).

The key elements of reinforcement learning (RL) are the agent (the drone), the policy (actions), the environment, the reward/penalty, and the state. The policy maps the drone's state to an action, determining its behaviour based on its position in the environment. After each action, the drone receives a reward or penalty and the next state from the environment. The reward/penalty is determined by a reward function that encourages good actions and discourages bad ones. The value of a state represents the expected cumulative reward, with future rewards discounted to favour actions that lead to immediate gains. The goal of RL is to design a reward function that trains the drone to learn the optimal policy, maximising cumulative reward while completing the task.

The aim of this investigation was to use RL to train a drone in a 2D PyGame environment to fly into targets. Each time the drone reached a target, a new one would spawn in a different location in the environment.

One previous study [7] successfully trained a quadcopter using proximal policy optimisation and deployed it zero-shot on real drones without additional tuning. The training relied on using neural networks to learn the optimal policy. Taking inspiration from their use of an episodic, model-free learning algorithm, this investigation sought to explore whether a Monte Carlo-based approach (also episodic and model-free) could be used to train a drone effectively. Specifically, this study used an on-policy first-visit Monte-Carlo control method with epsilon-greedy for exploration to train a drone in a 2D PyGame environment to fly into targets. The PyGame visualisation of the drone's flight towards targets, paired with the graph showing cumulative reward over training epochs, was used to evaluate training progress. The drone failing to hit all the targets consistently in the visualisation indicates the algorithm has reached a local minimum, where further training may not lead to significant improvement. However, the drone successfully hitting all targets, suggests the algorithm has converged to the global minimum, meaning it had learned the optimal policy.

***Drone Physics.***– For simplicity, the drone is modeled as a rigid body with a weight of $mg = 1$. Vertical motion occurs when the thrust produced by the left and right propellers, $T_{\text{left}}$ and $T_{\text{right}}$, is equal. The drone remains stationary if $T_{\text{left}} = T_{\text{right}} = 0.5$. If $T_{\text{right}} > T_{\text{left}}$, the drone begins to rotate counterclockwise, whereas if $T_{\text{left}} > T_{\text{right}}$, it rotates clockwise. Additionally, the drone ascends if the total thrust ($T_{\text{left}} + T_{\text{right}}$) exceeds its weight and descends if it is less.

***Method.***– The drone may choose from a range of thrust values between 0 and 1 inclusive, in steps of 0.25 from its left and right propellers. This range and step size was chosen to reduce the number of actions the drone has to explore, as for this range the drone only had 25 possible actions to consider for each state. Since the drone has two propellers that control the thrust, the size of the action space scales as the square of the number of allowed thrust values. For example, a step size of 0.1 increases the action space to 121 possible actions per state, significantly increasing the complexity of optimisation.

A simple state space, $(\Delta x, \Delta y)$, where $\Delta x$ and $\Delta y$ represent the horizontal and vertical distances between the drone and the target was defined, allowing the drone to directly learn from its position relative to the target. To ensure a finite state space, distances were multiplied by 10 and then truncated to transform them from decimal to integer values.

The values of performing specific actions in a given state were stored in a Q-table. Initially, all Q-values were set to a small positive value (0.1) to encourage exploration when encountering new states.

The reward function

$$R(s,a) = \begin{cases} 100 & \text{if the drone has reached the target} \\ -50 & \text{if } d > 9 \\ 10 - d & \text{otherwise} \end{cases} \tag{1}$$

was largely dependant on the Euclidean distance

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{2}$$

between the drone and target. Equation (1) highlights that a large positive reward for reaching the target, a large penalty for failure (defined as exceeding $d > 9$), and a small positive reward that increases as the drone gets closer to the target were given to the drone.

Having determined a reward the Monte Carlo update formula was used according the recursive relation:

$$G \leftarrow R + \gamma G \tag{3}$$

Where $R$ represents the immediate reward for taking an action, $\gamma$ is the discount factor and $G$ is the expected return. Since the action selection process is traced backward from the terminal state to the current state, the return $G$ is used because it accounts for the total reward accumulated from the current time-step onward.

The Q-values were updated using the Monte-Carlo update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G - Q(s,a)) \tag{4}$$

where $\alpha$ is the learning rate, a weighting factor which determines how much $Q$ is updated based on the new value.

This method uses first visit Monte Carlo updates rather than every visit updates as it helps to reduce bias by treating visits independently and preventing redundant updates [8].

To balance the exploration-exploitation trade-off, a linear epsilon-greedy strategy was used. A random number between 0 and 1 was sampled from a uniform distribution. If the number was less than $\epsilon$ a random action was performed (exploration), otherwise, the action with the highest Q-value was selected (exploitation). The exploration rate was updated as:

$$\epsilon \leftarrow \max(\epsilon_{\min}\ , \epsilon \cdot (1 - \epsilon_{\text{decay}})) \tag{5}$$

where $\epsilon_{min}$ is the minimum exploration rate and $\epsilon_{decay}$ is the decay factor to gradually reduce exploration rate and increase exploitation rate $(1 - \epsilon)$. This strategy encourages more exploration at the start when the drone has limited knowledge of the environment, allowing the drone to explore and learn the best possible actions. Over time, as $\epsilon$ linearly decays from its initial value of 1 to a minimum of 0.1, the policy gradually shifts towards one of exploitation, coinciding with the model gaining more knowledge. $\epsilon_{min}$ was set to be 0.1 to ensure that the drone retains some level of exploration throughout the whole training process, with the aim of preventing it getting stuck in local minima.

- justify why linear epsilon decay was used instead of exponential

***Results.***–

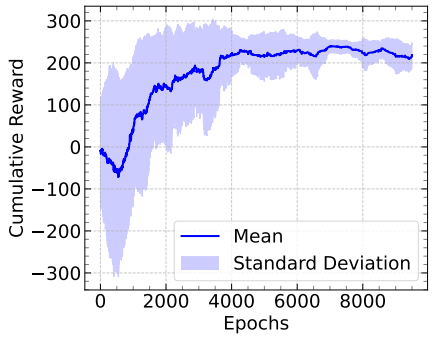


FIG. 1. **Graph of cumulative reward against epochs. The mean plotted is the average of 10 runs, each with 10,000 epochs. Hyperparameters:** $\alpha = 0.05$**,** $\gamma = 0.9$**,** $\epsilon_{decay} = 0.001$ **and** $\epsilon_{min} = 0.1$**.**

Figure 1 shows a graph of the mean cumulative reward of 10 runs, each of 10,000 epochs. A rolling average (with window size 500) was applied to reduce noise and improve interpretability. This reduced the sample size per run from 10,000 to 9501. The standard deviation was plotted alongside the mean to illustrate the error. It was highest during the early training phase (0–4000 epochs) due to a larger exploration rate resulting in actions being chosen randomly. As training progressed and the exploration rate decayed, the standard deviation decreased. The drone's cumulative reward converged to

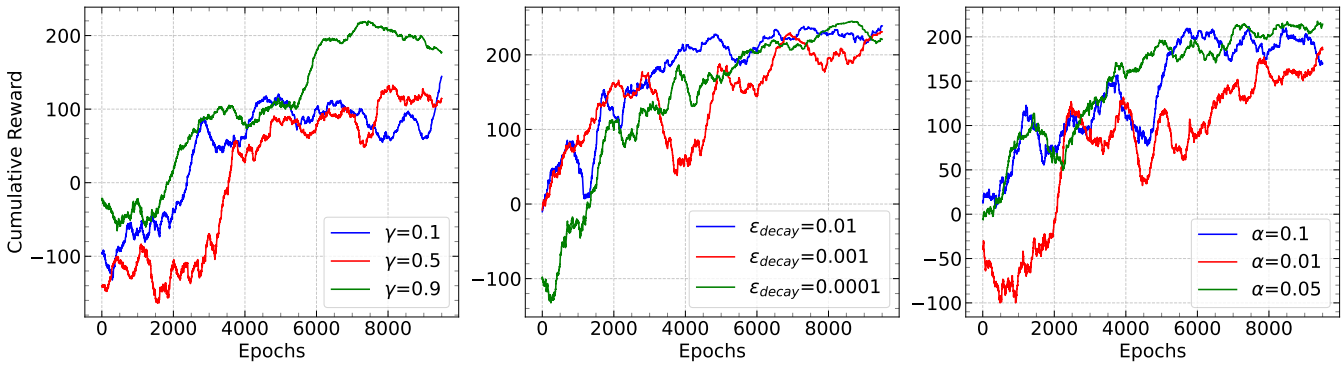


FIG. 2. **Comparison of cumulative reward against training epochs for different values of discount factor ($\gamma$), epsilon decay rate ($\epsilon_{decay}$), and learning rate ($\alpha$).**

approximately 200-230 after around 4000 epochs. The PyGame simulation revealed that the drone consistently reached only one of the targets, failing to find the others. This indicates convergence to a local optimum.

The learning rate was set to $\alpha = 0.05$ to allow gradual updates of the Q-values while avoiding instability. A higher learning rate could result in excessive variance in updates, whereas a lower rate might slow the learning process significantly. The discount factor was set to $\gamma = 0.9$ to encourage long-term reward optimisation while maintaining a reasonable emphasis on immediate rewards. A lower value would lead to short-sighted decision-making, while a value closer to one could delay convergence due to excessive prioritisation of future rewards. These are commonly used values for these hyperparameters (see [9]). $\epsilon_{decay}$ was set to 0.001 to gradually reduce randomness in action selection.

Figure 2 illustrates the effect of different hyperparameter values on cumulative reward progression over training epochs.

In the leftmost plot, the performance of different values of $\gamma$ is compared. A higher discount factor ($\gamma = 0.9$, green) leads to the best performance, while both $\gamma = 0.5$ (red) and $\gamma = 0.1$ (blue) result in similar lower cumulative rewards. This is expected, as a larger $\gamma$ allows the agent to place greater emphasis on future rewards rather than immediate gains, leading to better long-term decision-making. In contrast, lower $\gamma$ values discount future rewards more heavily, encouraging short-sighted policies that do not yield optimal results in the long run. The similarity in performance between $\gamma = 0.1$ and $\gamma = 0.5$ suggests that beyond a certain threshold, overly aggressive discounting significantly hampers the drones learning efficiency.

The middle plot examines the impact of different $\epsilon_{decay}$ values on training progression. Despite initial differences in learning speed, all three decay rates ultimately converge to a similar cumulative reward. The smallest decay rate ($\epsilon_{decay} = 0.0001$, green) takes the longest to reach convergence, as it maintains exploration for an extended period before stabilizing. In contrast, the largest decay rate ($\epsilon_{decay} = 0.01$, blue) reduces exploration too quickly, leading to faster convergence but potentially suboptimal policies due to premature exploitation. The intermediate decay rate ($\epsilon_{decay} = 0.001$, red) offers a balance between exploration and exploitation, converging at a reasonable pace while maintaining stable learning.

The rightmost plot illustrates the effect of varying the learning rate $\alpha$. While all three values exhibit an increasing trend in cumulative reward, the rate and stability of learning differ. The intermediate learning rate ($\alpha = 0.1$, blue) results in rapid but unstable learning, with large fluctuations in performance. The lowest learning rate ($\alpha = 0.01$, red) produces a smoother but slower learning process, ultimately reaching a lower final reward. The intermediate learning rate ($\alpha = 0.05$, green) achieves a balance between stability and learning speed, reaching a high cumulative reward with moderate variability.

Overall, while different hyperparameter values influence the rate of convergence, stability, and overall performance, the results suggest that well-tuned parameters can significantly enhance learning efficiency. Notably, all $\epsilon_{decay}$ values lead to convergence at a similar cumulative reward, highlighting the robustness of the training process. However, the choice of $\gamma$ and $\alpha$ strongly affects both the final reward and the stability of training, reinforcing the importance of careful hyperparameter selection.

***Theoretical Improvements.***– A major barrier to reinforcement learning is the amount of computational power needed to train more complex models [reference], this has been a key consideration in our model. In this section we explore advancements to the model, the majority of which require more computational power than our method.

The most important consideration is that our state space fails to give a complete picture of the state of the drone, reducing the drone's generalisability. Two key factors not included in our model are the velocity (in both

the x and y directions) and pitch of the drone, including this extra information would allow the drone to recognise for example, that it is approaching a target way too fast and will shoot off after rather than being able to slow and change direction towards another target, likewise it will be unable to recognise if it is pointed in completely the wrong direction. Our state space ranged from 0 to 9 for both x and y distance allowing 100 total states for the drone to explore, having 10 states each for both velocity components as well as the pitch would increase this to 100,000 states, clearly a much larger computational task.

A benefit to this method is it makes it much easier to add velocity and pitch based rewards such as a reward for slowing on approach to a target or a reward for having the pitch aligned to the target. A specific reward that could be beneficial within this expanded state space is a velocity alignment reward the method for calculating this reward would be as follows:

$$A = \hat{\mathbf{v}} \cdot \hat{\mathbf{d}} \tag{6}$$

Where $\hat{\mathbf{v}}$ and $\hat{\mathbf{d}}$ are the velocity and distance unit vectors of the drone and $A$ is the alignment value of the drone. This reward would be scaled to compliment a distance based reward without overpowering it and could contain a term encouraging it to move and a specific velocity by rewarding velocities closer to this term more than those further from it.

A less computationally expensive (but non generalisable) option would be to treat each target as its own individual state for a sequence of four targets this would mean the state space only increased to 400, unfortunately this is a difficult approach as the drone keeps no information regarding the velocity or pitch so on different iterations if the drone has enough variation when hitting targets (and thus changing to a completely new state space) it may struggle to learn this.

One option which could be considered to increase precision when close to the drone is applying a transformation by $\log(1+x)$. This would make the gaps between successive values in the state space smaller (in real terms) when the distance $x$ is smaller and larger when $x$ is larger.

***Conclusion.***– In this work, we employed an on-policy, first-visit Monte Carlo method with $\epsilon$-greedy exploration to train a drone in a 2D PyGame environment. The objective was for the drone to navigate towards specified targets. This approach extends previous research that utilised Proximal Policy Optimisation (PPO) to successfully train a drone. Our goal was to explore the feasibility of implementing a different episodic, model-free reinforcement learning algorithm for drone training.

However, the results demonstrate that the Monte Carlo-based method was not successful in this case. While the drone showed evidence of learning and converging to a maximum cumulative reward, it ultimately converged to a local optimum, as it was only able to reach one of the targets. This suggests that the Monte Carlo method, in this context, may have limitations when compared to other approaches like PPO.


[1] C. A. Lin, K. Shah, L. C. C. Mauntel, and S. A. Shah, The Bulletin of the American Society of Hospital Pharmacists **75**, 153 (2018).

[2] D. Kunertova, Bulletin of the atomic scientists **79**, 95 (2023).

[3] T. Hiraguri, H. Shimizu, T. Kimura, T. Matsuda, K. Maruta, Y. Takemura, T. Ohya, and T. Takanashi, IEEE Access (2023).

[4] C. Albon and N. Robertson, Defense News (2024).

[5] M. of Defence, GOV.UK (2024).

[6] A. T. Azar, A. Koubaa, N. Ali Mohamed, H. A. Ibrahim, Z. F. Ibrahim, M. Kazim, A. Ammar, B. Benjdira, A. M. Khamis, I. A. Hameed, *et al.*, Electronics **10**, 999 (2021).

[7] D. Zhang, A. Loquercio, X. Wu, A. Kumar, J. Malik, and M. W. Mueller, ar Xiv preprint (2022).

[8] S. P. Singh and R. S. Sutton, Machine Learning **22**, 123 (1996).

[9] Z. Wang and M. E. Taylor, in *IJCAI* (2017) pp. 3027–3033.