

Learning to Fly: Q-Learning vs. Neuroevolution for Autonomous Drone Flight in a 2D Python Simulator

Jack Paine and Albert Haladay
University of Nottingham

This report explores two model-free reinforcement learning approaches, Q-learning and neuroevolution, for the autonomous flight of a simulated 2D drone. The two models are compared in their abilities to reach randomly generated targets, with the neural model showing faster and more precise control.

I. Introduction

We are provided with a 2D drone simulation written in Python [1]. This simplified drone is controlled directly through two propellers on either side and is given sequentially appearing targets to guide its flight path.

A correctly implemented controller should be able to fly the drone towards its target destinations indefinitely, without flying off-screen or staying still for extended periods.

II. Methodology & Implementation

A. Expected Behaviour & Performance Measures

We can assume that the ‘better’ a controller is, the faster it will reach its given targets; from this, a simple benchmark is to monitor how many targets a controller can reach in a given time period.

The target co-ordinates for this test are randomly scattered within the simulation’s bounds, and pre-generated to ensure a fair comparison between controllers. This set is not to be seen by either controller during training – a separate training set of targets is used instead.

B. Formulating the Problem for RL

Q-learning and neuroevolution are both *model-free* algorithms, meaning they do not require an explicit representation of the environment’s dynamics, which is especially beneficial for this task as the dynamics of the drone would be too complex to effectively model. We give each agent:

- A *State Space* (S): A simplified model of the environment at each time step; it should provide enough information for the agent to make effective decisions without being so detailed as to overwhelm the agent.
- An *Action Space* (A): A set of actions the agent can choose from to alter its sta.
- A *Reward Function* (R): Assesses the ‘quality’ of an action based on how the environment changes.

The agent uses this information to learn a policy, π , that maps states to actions $\pi : S \rightarrow A$ in a way that maximises the expected reward. The exact formulation of states, actions and rewards is paramount to the success of the algorithm and may be different for each of our RL models.

C. Q-Learning Model

Q-learning involves iteratively assessing the quality of each state-action pair, which is stored in a *Q-table*. The agent begins with no knowledge and learns which actions to perform through repeated simulation runs. The Q-table is updated at each time step according to the Q-value update rule[2]:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

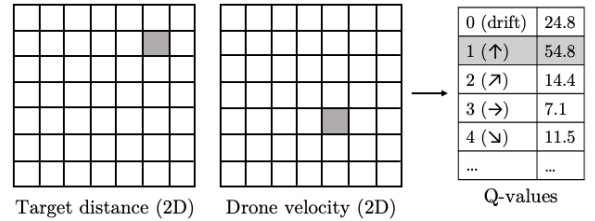
Where r is the immediate reward for the action given by R , $\max Q(s', a')$ is the highest Q-value of the new state, and α, γ are

tunable hyperparameters.

This enables the Q-value of each action to be updated according to its expected return, accounting for the future rewards using γ .

a. *State Space* Q-learning requires that the state and action spaces be discretised. We settled on a state space with 4 dimensions: the x and y distances to the target coordinate, and the x and y velocities of the drone. Each dimension is discretised into one of 7 bins for a total state space of $7^4 = 2401$ possible states, as visualised in FIG. 1.

b. *Action Space* We discretised the action space with 9 heuristics: one for each cardinal and inter-cardinal direction, and one for no movement (‘drift’). They work on the physics of the drone to ensure consistent movement. The actions are rate-limited to once every 0.25s, which ensures each action achieves the desired motion.



0 (drift)	24.8
1 (↑)	54.8
2 (↗)	14.4
3 (→)	7.1
4 (↘)	11.5
...	...

FIG. 1. Illustration example of a Q-table lookup

c. *Reward Function* Our reward function consists of only three components: a large reward for reaching a target; a small punishment for overshooting a target; and an even smaller punishment proportional to the distance from the current target. This reward aims to encourage the drone to move toward the target, whilst punishing overshooting.

d. *Training & Hyperparameters* With no initial knowledge of the environment, the agent is encouraged to experiment by choosing a random action, with probability ϵ . Along with two parameters in the Q-formula, we have three tunable hyperparameters:

1. The learning rate, $0 \leq \alpha \leq 1$, determines the speed at which Q-values are updated. We use a default value of 0.1.
2. The discount factor, $0 \leq \gamma \leq 1$, determines the importance of future rewards. We use a high value of 0.9 since the drone’s actions have a long-term effect on its trajectory.
3. The exploration rate, $0 \leq \epsilon \leq 1$, is the ratio of random actions to be chosen over the best-known action. This decays from 0.9 to 0 throughout the training run.

The Q-learning agent is trained with these parameters for 100,000 training episodes with 6000 time steps each, equivalent to 69.4 days of real-time training. This is long enough to show convergence of the Q-values, implying further training

would have minimal benefit to the agent’s performance.

D. Neuroevolution Model

While neural networks are typically classed under the supervised learning paradigm, they can be tailored for use in reinforcement learning tasks through the use of an evolutionary training algorithm.

Unlike Q-learning, a neural model does not require discretisation of the state or action spaces, potentially allowing for more precise decisions.

a. State Space We augmented the 4 dimensions of the previous model by adding the drone’s current pitch, pitch velocity, and the network’s last outputs (making it a simple form of recurrent neural network). These form the inputs of the network as visualised in FIG. 2.

b. Action Space Instead of re-using the heuristic actions of the previous model, the network directly outputs the two thrust values ($A = [0, 1]^2$) with the vision that its recurrent connection enables it to learn actions for itself.

The sigmoid activation function is used, matching the outputs to the $0 \rightarrow 1$ range of accepted thruster values.

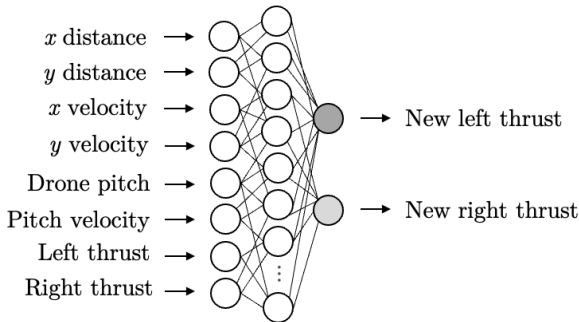


FIG. 2. The basic neural network topology

c. Training & Hyperparameters The network is trained via *Random Search*, a simple neuroevolution algorithm which searches the weight space of a network without the use of gradient calculations. The algorithm loosely mimics the process of natural selection, and is performed as follows [3]:

1. Initialise a neural network with random parameters θ , and fitness score $F_\theta = 0$.
2. Generate n candidate networks, $V_1, V_2, V_3, \dots, V_n$, by randomly mutating θ for each parameter in range $-\alpha \rightarrow \alpha$ (where α is the *mutation rate*).
3. For each candidate V_i , run a simulation to calculate its fitness score, F_{V_i} , as the cumulative reward from a 60s flight simulation.
4. If any of $F_{V_i} > F_\theta$, update the current best parameters and fitness as $\theta \leftarrow V_i$ and $F_\theta \leftarrow F_{V_i}$.
5. Repeat steps 2-4 for g generations, or until the network reaches a satisfactory fitness.

We re-used the reward function from our Q-learning development, with minor modifications to suit the new training algorithm: for example, ending simulations early if the reward dropped below a certain value, to improve the training speed.

For the final training run we use $n = 100$, $g = 1000$, and a decaying α from 1.0 to 0.01. The algorithm is further repeated with varying initial θ to ensure full exploration of the weight space.

d. Network Topology Random Search works with a fixed network topology, which must be found through trial

and error. Only two-layered networks are tested; initial training runs show performance peaks with a hidden layer of 64 neurons (out of 8, 16, 32, 64 & 128 tested).

III. Evaluation

We evaluated each model by monitoring the number of targets hit throughout a two-minute training run with unseen, randomly generated targets. The results can be seen in FIG 3. The neural model outperforms both others, with 47 targets hit, compared to 8 and 17 for the provided heuristic controller and our Q-learning model respectively.

Simulating the drone with the graphics reveals the behaviour of each controller. We see that the heuristic model flies out-of-bounds indefinitely after reaching the 8th target. The Q-learning model behaves sporadically with sometimes unusual movement paths, but always eventually finds its way to the target. The neural model shows much more controlled behaviour.

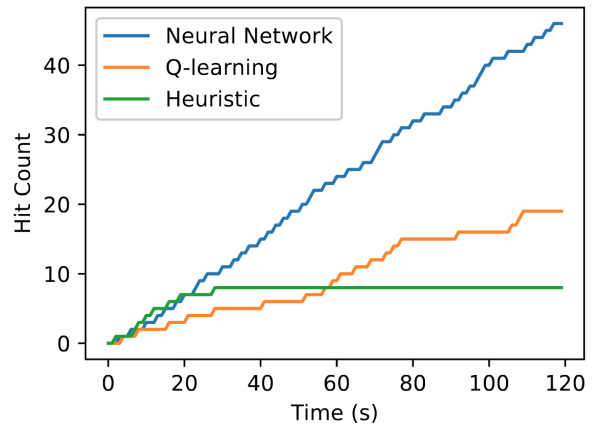


FIG. 3. Performance of each controller on test simulation

IV. Discussion & Conclusion

Along with the benchmark results, it is important to note the caveats of implementing each model. The Q-learning model was challenging to develop, as training only succeeded once the optimal state space was found – the working model with its 4 dimensions and 7 bins is one of over 20 we tried before settling on its performance. It was certainly held back further by the limited choice of heuristic actions.

Contrastingly, the neuroevolution model was relatively easy to implement, and the lack of tuning required for the random search algorithm was particularly beneficial. The model could be improved further still through experimentation of more complex topologies or alternative methods of training like NEAT[4], which adapts the network topology to best suit the task.

To conclude, we found the neuroevolution model to be very suitable for this task, with noteworthy results from the Q-learning model.

References

- [1] J. Mair, Drone flight controller repository, <https://github.com/JamieMair/DroneControllerMLiS> (2020).
- [2] A. G. B. Richard S. Sutton, *Reinforcement learning: an introduction*, Adaptive Computation and Machine Learning (The MIT Press, 1998).
- [3] Z. B. Zabinsky, *Random Search Algorithms*, Tech. Rep. (University of Washington, 2009).
- [4] K. O. Stanley and R. Miikkulainen, Evolving neural networks through augmenting topologies, *Evolutionary Computation* **10**, 99–127 (2002).