Deep Reinforcement Learning on Robot control (Lunar Lander, OpenGym)

Damian Abasto, damian.f.abasto@gmail.com

***Abstract*—Deep Reinforcement Learning is applied to solve the `gym` Lunar Lander environment, including variants such as experience replay, double deep Q-learning and dueling networks. The problem consist of landing a space craft by correctly firing different engines, or do nothing. The state space consist of 8 continuous variables that provide information on the craft position and velocity, plus two boolean variables specifying whether or not the craft legs have touched the ground. The action space is discrete and four dimensional. The problem of credit assignment is particularly acute in this problem, as the agent receives a positive reward only after landing successfully at the origin, potentially after several prior steps.**

The neural network was implemented using `PyTorch` and run in the Google Cloud levering `Google Colab` GPUs. Several configurations of the hyperparameters were explored by performing a grid search and analyzing the performance of the agent over multiple episodes for training and test runs. We report the best set of hyper-parameters found, analyzed how well the learned Q-values generalized to unseen states, as well as study the effect of various hyperparameter choices and their intuitive impact of the agent's training performance.

I. Introduction

A. Problem statement

The objective of this work is to utilize techniques from Reinforcement Learning to solve the LunarLander-v2 environment [3]. This environment consists of a space craft (agent) that must successfully navigate through a 2D space and land at the origin $(x, y) = (0, 0)$, starting from a random vertical position above the ground. The agent observations are 8 dimensional, consisting of the x and y positions and velocities, angular position and velocity, and two additional boolean variables indicating whether the left or right leg of the space craft touched the ground. The agent can perform four actions, which are do nothing, fire the left engine, fire the main engine, or the right engine. The agent receives a reward between 100 to 140 if it moves from the top and lands on the origin with zero speed. Rewards are negative if it moves away from the origin, it is -100 if it crashes, receives-0.3 points for firing the main engine and -0.03 for using the side engines for each frame. The environment is considered solved if the agent receives an average reward of at least 200 for the last consecutive 100 episodes.

B. Techniques

We employ off-policy Q-learning control to solve the Lunar Lander-v2 environment, where the objective is to achieve the optimal action-value function $q_*(s, a)$ by performing recurrent updates to estimates of the value function, following the off-policy[1] TD control algorithm known as Q-leaning [6]: $Q(S_{t+1}, A_{t+1}) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))]$. Importantly, the fixed point of this equation approximates the optimal action-value function q_*, regardless of the policy being followed.

1) Function approximation with neural networks: The first six coordinates from the observation space span a continuum. One possible approach to address a continuous state space consists of discretizing the state space S_t. However, this could potentially provide a coarse approximation of the action value function $Q(S_t, A_t)$ with poor generalization, as an update to the function for one given bin will be independent and not affect the function in another given bin, even if those are nearby in the state space S_t. A better approach that tends to generalize much better is to use function approximation to represent the action value function $Q(S_t, A_t)$ using neural networks, with parameters w, so that $Q(S_t, A_t) \equiv Q(S_t, A_t; w)$. Not only are neural networks universal function approximators[2], but they also provide generalization out of sample, as an update to the parameters w will affect the action value functions across all states S_t and actions A_t, in contrast to the discretization approach mentioned before. This also translates into fewer observations and episodes that will need to be run for the agent to generalized and solve the environment.

Neural networks consist of a successive concatenation of linear transformations followed by component-wise nonlinear functions, $a^{(k)} = b^{(k)} + W^{(k)} h^{(k)}$, $h^{(k)} = f(a^{(k)})$, with the activations $a^{(k)}$ such that $x^{(0)} \equiv x$, with x the 8 dimensional vector of observations received by the agent, and f a non-linear function applied component-wise to the activations. When applied to Q-learning, the neural network parameters w_t for a given step t are updated in the gradient direction that minimizes the difference between the TD target $\delta_t^Q = [R_{t+1} + \gamma \max_a Q(S_{t+1}, a; w_t)$ and $Q(S_t, A_t; w_t)$, that is:

$$w_{t+1} = w_t + \alpha(\delta_t^Q - Q(S_t, A_t; w))\nabla_w Q(S_t, A_t; w) \quad (1)$$

This update can be achieved by doing stochastic gradient descent with min-batches, choosing the mean square loss as the objective function [3]. We employ `PyTorch` as the

[1]The Algorithm is called off-policy because the policy followed by the agent (behavioral policy) is different form the optimal policy the agent is trying to learn

[2]This means neural networks approximate any function within any given accuracy, provided the network has sufficient number of parameters or layers and non-linear activation functions, [2]

[3]The choice of loss function does matter. We obtained worse results using L1 loss, for example

package to perform deep learning, utilize the Relu activation function $Relu(x) = max(x, 0)$ for the non-linear function, and experimented with various different configurations of neurons per layer. The first layer matrix $W^{(k)}$ is $n^{[1]} \times 8$, with $n^{[l]}$ the number of neurons in the layer l, and the last layer does not have a non-linear activation function, so that the output network can take both positive and negative values, with output dimension of 4, equal to the action space. With this architecture, a one pass of the neural network with input S_t approximates the value function across all four possible actions A_t.

It is known in the literature (see [5]) that using neural networks can make the solutions unstable . Many different heuristics can be developed to stabilize and improve the performance of Q-learning with neural networks (or deep Q-learning), which we have implemented in the current work and describe briefly now.

2) Experience replay: Instead of an agent learning incrementally from a stream of observations, in experience replay those observations are stored in a memory buffer of fixed size and then sampled uniformly at random. This sampling breaks the strong correlation of consecutive experiences, making it more suitable to the i.i.d. assumption made by a stochastic gradient algorithm. In addition, the buffer technique could potentially store and sample more than once rare experiences that could prove useful for learning (such as the agent landing successfully or prior steps that led to this). We implemented experience replay via a simple Python's `deque` container and analyzed the effect of various replay sizes in the later section[4].

3) Double Deep Q-learning: In the setting described in eq.(1), the same network is used to select the best action a and to evaluate this action. This can overestimate the Q-values. Even in a setting where all actions were equally good, since the neural network are really approximations of the true Q-value functions, some actions will be slightly greater than others due to randomness. However, since the max in eq.(1) will always select the largest Q-value, this will produce an over-optimistic estimation of Q-values. To solve for this, the authors in [1] proposed two networks, represented by two sets of parameters w and w^-, one for selecting the best action a, and another one to evaluate it. The TD target is as follows:

$$\delta_t^{doubleQ} = \left[R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}_a Q(S_t, a; w); w^-)\right] \tag{2}$$

where in practice the same network configuration (number of layers and dimensions) are used for both the *training* model (used to select the best action and whose parameters w are updated via gradient descent), and the *target* model (used to evaluate the best action and whose parameters w^- are copied over from the training model after certain frame steps within an episode). By copying the parameters from the training network to the target network only infrequently, we also stabilize the target δ and produce a sampling that is more identically distributed over the experiences, as the target model behaves more stationary and their Q-values are therefore not changing over time, for a given set of frame steps. In addition, [4] introduced the idea of *soft* updates, whereby the target model parameters are updated with a weighted average between its current parameters and those from the training model, as follows: $w^- = \tau w + (1 - \tau)w^-$, with a small weight τ. This setup has been found to stabilize and improve the performance of deep Q-learning algorithms. We have implemented both double Q-learning and soft updates in this project.

4) Dueling Deep Q-learning: Originally introduced in the context of convolutional neural networks, the idea of dueling deep Q-learning [7] is to break up the estimation of the state action value as a separate estimation of the state value function and a state-dependent action advantage function, to better generalize actions. Depending on the state, the choice of action can have less repercussion than for other states, and so it is unnecessary to estimate the value of each action choice for every state. To materialize this insight, the authors started from the definition of the advantage function $A(s, a)$ given by $Q_\pi(s, a) = V_\pi(s) + A_\pi(s, a)$. Since $V_\pi(s) = E_\pi[Q_\pi(s, a)]$, it follows that $E_\pi[A_\pi(s, a)] = 0$. To increase stability and address identifiability issues[5], the authors proposed the following scheme to compute $Q(S, A; w)$:

$$Q(S, A; w) = V(S; w) + \Big(A(s, a; w) - 1/|\mathcal{A}| \sum_{a'} A(s, a'; w)\Big), \tag{3}$$

where $|\mathcal{A}|$ denotes the size of the discrete set of actions a it can choose from, so that $a = \{1, 2, 3, ..., |\mathcal{A}|\}$. We have implemented this technique in PyTorch to solve Lunar Lander.

5) ϵ-greedy exploration: We have combined all the previous techniques with an ϵ-greedy type policy to encourage exploration. The best performing methodology was to perform a power-like decaying scheme with a minimum, where for a given episode t, $\epsilon_t = min(\epsilon_{\text{min}}, \epsilon_0 \times (\delta_\epsilon)^t)$, for some initial ϵ_0 decay constant δ_ϵ and minimum ϵ_{min}, which guarantees a minimum level of exploration regardless of the episode.

II. Results

In this section we analyze the results we have obtained by the various techniques mentioned in the previous section. There are a plethora of hyperparameters to tune, and we have implemented a simple, deterministic grid search to test the agent through them and be able to choose the best performing combination. We utilized PyTorch to train the neural networks, and used `Google Colab` GPUs to seep up the computations. Various different sets of hyperparameters were found to solve

[4]A `deque` container with fixed length `lmax` implements a first in, last out. If A=[] is an empty deque, then A.insert(2) gives A = [2], A.insert(3) gives A = [2, 3], A.insert(4) gives A = [3, 4]. In this way, after `lmax` insertions, the oldest experience start to get discarded, only retaining more recent experiences. The buffer then has fixed memory, only saving the most recent `lmax` experiences

[5]A naive approach to implement a dueling network setting would be to use one network is to estimate $V(s)$ and another one for $A(s, a)$. However, since $Q_\pi(s, a) = V_\pi(s) + A_\pi(s, a)$, the networks would not be unique, as one can add and subtract a constant to V and A, respectively, resulting in the same Q-value. This naive approach is unidentifiable.

the environment, but a given set will be deem best if it is the one that solves the environment with the least number of episodes. In addition to solve the environment, we will analyze the effect of changing various hyper-parameters.

In order to ensure repeatable results for the best performing hyperparameter choice, we fixed the seed for PyTorch, the LunarLander-v2 environment, as well as the random number generators for ϵ-greedy when running the best set of hyper-parameters, while we performed various sets of runs over the same hyperparameter and not seeding the environment to test the robustness and influence of various hyperparameter values.

1) Best results: As mentioned before, the environment is considered solved when the agent mean reward over the last consecutive 100 episodes is at least 200. The Python implementation runs for a maximum of 1000 episodes and stop training as soon as this condition is met[6]. In addition, a *test* run is performed with 120 episodes, with a fresh new environment and with zero exploration. A simple grid search was implemented to search over the space of hyperparameters, all of them starting from a given base case. At the beginning of hyperparameter exploration, we utilized shorter episodes, 300, and iterated around parameters that have been published in the literature, until the best set of parameters was identified[7].

Figure 1 presents the best agent *training* and *test* performance which we have found, which solved the environment in 383 episodes, with a rolling *training* average reward of 200.8 at episode 383 over the last consecutive 100 episodes[8], and achieves a *test* mean reward of 234.2 over 120 test episodes[9]. For reference later, the rolling and mean standard deviations are also displayed.

The best parameters found are as follows: $\epsilon_0 = 1.0$, $\epsilon_{\text{min}} = 0.01$, $\delta_\epsilon = 0.995$, replay buffer size of 100,000, $\gamma = 0.99$, $\tau = 0.01$, $\alpha = 0.001$, with a neural network consisting of three hidden layers, with dimensions 128, 64, and 32, and using mini-batch gradient descent with a batch size of 64.

To visualize the Q-values learned by this best model and to study how well the neural network generalizes to unseen observations, we ran the trained neural network through an array of values across the coordinate/observation space, and computed the value function $V(s)$ from the neural network model by doing $V(s) = \max_a Q(s, a)$. For easier interpretation, we change one coordinate at a time, setting all others to zero[10]. The result is shown in Figure 2. The observations generated this way are artificial, as it is unlikely the agent experienced any of them. The plot shows this clearly, as the neural network does not generalize well for a range of coordinate values. For example, while the plot of value function across y-coordinates shows a decreasing value function from 0.0 up to around 0.3 (this makes sense, since the higher the lunar lander is, with zero velocity, the more difficult is to reach to the landing position, so the worse the state should be), it then starts to increase again, which should not be expected. And while the x-coordinate plot shape makes sense (the maximum value should be at 0.0), the x-velocity does not look correct either, as the agent should be worse off, with negative state values, if it is almost at landing position, but has a large negative or positive velocities. The plot shows the opposite. Similarly for the angular position.

The agent only sees a very small fraction of the complete state space, and its generalization to states it has not seen can be poor, specially without much exploration. We stress that we set the seeds to ensure reproducible results for the best performing agent that we found. However, we have not set the seed in the following section, when studying the effect of hyperparameters[11].

2) Impact of hyperparameters: In this section we analyze the effect of various hyper-parameter value choices. We do this by running the agent for 500 episodes, 20 times each, with different seeds for the environment, the agent, and PyTorch. The rewards are then averaged over the 20 runs across each episode. This not only creates a more robust measure of the hyper-parameter impact on the training performance over several different environments, but also, accounts for the fact that different `Google Colab` GPU sessions did not produce reproducible results. Figure 3 displays the training performance of the agent over 20 sets, each set consisting of 500 episodes, each set with a different seed. Displayed is the average reward over 20 runs (in gray color), and the moving average on a window of 10 episodes (in color).

Neural Network dimension represents the hidden layers of the network, and we see that on average the agent shows lightly better performance with increasing network sizes and number of parameters, as the neural network can represent more complex functions. However, the increase is not that significant, at least in the 500 episode run, and even with the smallest [32, 16, 16] the agent can achieve reasonably good average rewards[12].

Decay rate γ determines how much to weigh recent rewards R_t compared to rewards that happened later on. Interestingly, only the higher gamma $\gamma = 0.99$ performed best. At $\gamma = 0.99$, it takes around $\log 0.5 / \log \gamma \sim 68$ steps to weight the rewards at 0.5, while it only takes 3 steps for $\gamma = 0.8$. The agent only gets a positive, relative large reward, when it is able to land the craft successfully, with potentially multiple prior steps with negative rewards. Therefore, the agent faces a challenging *credit assignment* problem, where it is necessary

[6]In the code we checked `env._max_episode_steps`, to distinguish between "good" episodes where the agent took perfect actions, or just episodes that ended in a crash

[7]However, cases were found where the agent performed well at the 300 episode mark, only to stall their performance should then run longer.

[8]The Rolling average is computed over a window of the last 100 episodes. $R_{\text{rolling mean}}$ displayed is the last rolling average computed

[9]Note that the test reward is higher than training, since we disabled random exploration, and only act greedily with respect to the optimized Q-function obtained by the training neural network

[10]Except for the y-coordinate, which has an offset of 0.01. This was make it consistent with either of the lunar lander legs not touching the ground, so their values are zero

[11]We hypothesize that training the same agent across multiple random instances of the gym environment should help the agent generalize better.

[12]Given the train performance we see, we hypothesize that even with [32, 16, 16] the agent may solve the environment for a *test* run, provided sufficiently large number of episodes

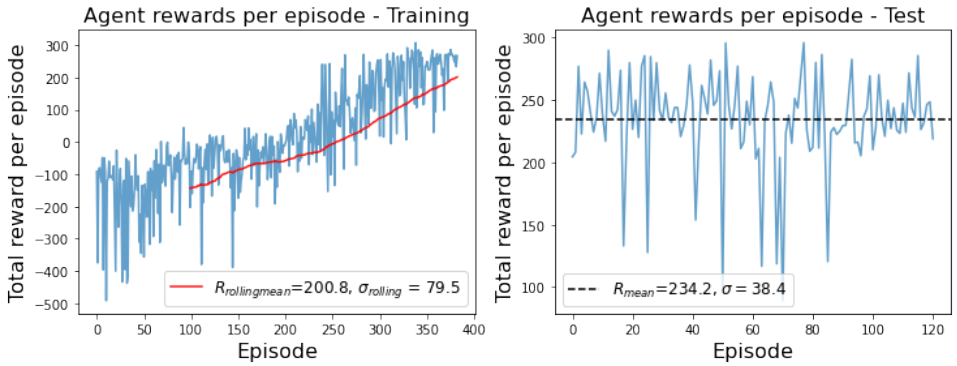

Fig. 1. Left: Best performing agent, optimized over a set of hyper-parameters. The plot shows the evolution of the rewards, together with their 100 episode rolling mean, as recorded during training. This agent solved the environment after 383 episodes, as can be observed by the $R_{\text{rolling mean}} = 200.8$, computed over a rolling window of 100. Right: performance of the best agent for a test run, where there is no exploration, and the agent follows the greedy policy dictated by $Q(s, a)$. As can be observed, the average achieves a *test* mean reward of 234.2 over 120 episodes. For reference later, the rolling and mean standard deviations are also displayed.

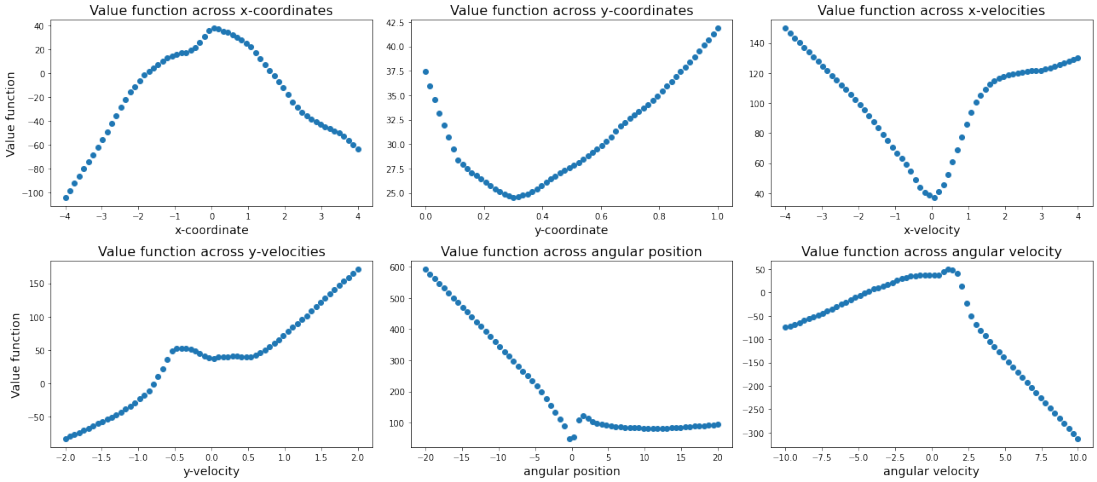

Fig. 2. Representation of the value function computed from the neural network model, $V(s) = \max_a Q(s, a)$, across different values of the coordinate/observation space. For easier interpretation, we change one coordinate at a time, setting all others to zero. The network does not generalize well in certain cases, such as for example the y-coordinate. See the body of the text for more discussion.

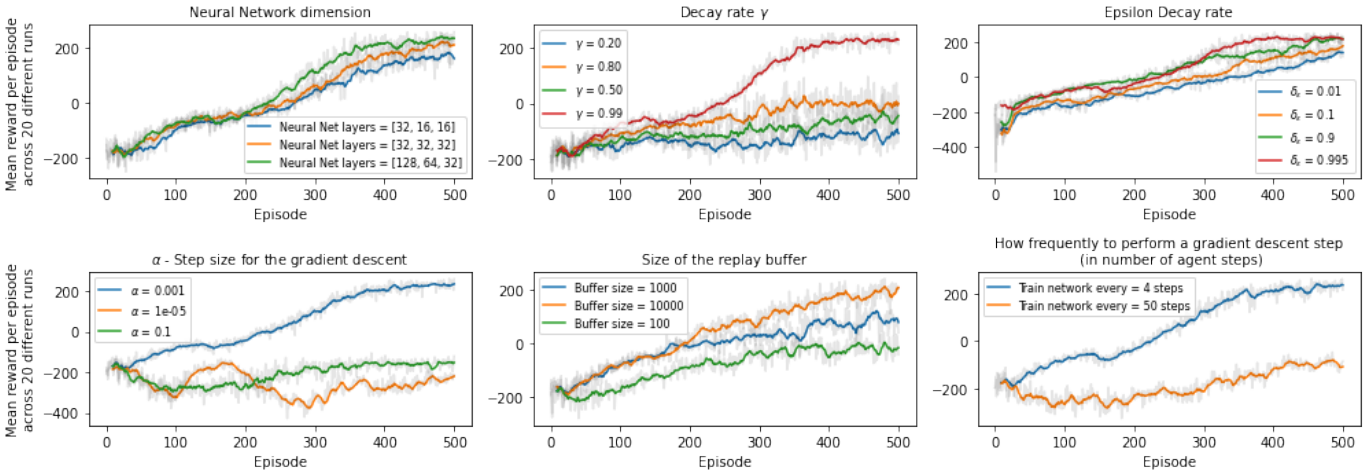

Fig. 3. Impact of the hyperparameters on the training performance of the agent. The agent was run over 20 sets, each set consisting of 500 episodes, each set with a different seed. Displayed is the average reward over 20 runs (in gray color), and the moving average on a window of 10 episodes (in color). α seems to be one of the most important, and also the (only?) parameter where the train performance is non-monotonic with respect to it, making it particularly difficult to fine tune.

to determine, from the multitude of prior steps, which ones were the most important in landing successfully. If the prior rewards are heavily discounted, prior good steps that lead to the successful reward may be given much less weigh and make it harder to the agent to determine the right succession of steps to take, prior to the large, positive reward at the end of a successful landing.

The decay rate δ_ϵ determines how quickly the initial exploration rate $\epsilon_0 = 1.0$ decays with the episode t, $\epsilon_t = \delta_\epsilon \times \epsilon_0$. The higher the decay rate, the less the agent will be incentivized to explore. The optimal value was found at $\delta_\epsilon = 0.995$, supporting longer exploration. The agent performs worse during the initial episodes for smaller values of the decay rate, probably because of it being stuck in local minima [13]. However, regardless of the decay rate, the original exploration seems to be enough for the agent to eventually perform equally well. Q-learning, as opposed to Sarsa, does not need non-zero continuous exploration throughout the episode, in part from the $\max_a$, making it greedy with respect to the best possible actions it has seen so far.

α **- Step size for the gradient descend** determines the size of the step towards the direction of the gradient descent in eq.(1). Large values of α can produce the gradient descent solution to "overshoot" the (local) minimum of the loss function and produce an oscillatory behavior around the fixed point, never converging to the solution. On the contrary, small values can make it harder and longer for the agent to converge. The optimal value was found at $\alpha = 0.001$, and we observe this parameter can have a large impact on the training performance of the agent. It also seems this is the (only?) parameter where the train performance is non-monotonic with respect to it, making it particularly difficult to fine tune.

Size of the replay buffer represents the maximum number of recent experiences that the agent is able to re-sample, before being discarded. Small replay buffer sizes will only contain relatively recent experiences, thus potentially discarding useful or rare experiences that may have happened in the past, making it harder for the agent to learn from them. On the contrary, larger buffer sizes allows the agent to retain and sample a larger set of potentially useful experiences (in this regard, the size of the memory has a somewhat analogous function compared to the decay rate γ: while larger γ "increases the influence" of past rewards, larger buffer sizes increase the memory and potential for re-encountering useful experiences with large rewards). Figure 3 confirms this intuition, as larger replay sizes improve the agent's performance monotonically.

Frequency of gradient descent, in number of agent steps represents how frequently the gradient descent and the update of the target neural network model is performed (the gradient descent step and the update of the target model were done one after the other). If the gradient steps and updates are only done infrequently, the agent will take longer to learn, as the updates to the parameters w would happen less frequently. Also, since the target network parameters were updated right after the gradient descent, the more infrequent this update is performed, the less accurate would be the evaluation of Q-values at the best possible action a, making the estimates more "obsolete".

[13] We have noticed sometimes the agent will just hover the spacecraft in the air and keep getting small, negative rewards, as opposed to "discover"' the larger reward for landing at the origin

III. Conclusion

The Lunar Lander-v2 environment was solved via deep Q-learning methods that perform Q-learning with a functional approximation using neural networks. In addition, various variants (experience replay, double deep Q-learning and dueling networks) were implemented and showed to improve the stability and performance of the agent. A simple grid search was utilized to search over the space of hyperparameters, reaching a best performing agent that can achieve a reward of +200 over the last 100 consecutive steps over only 383 episodes. The influence of each hyperparameter on the agent's performance and intuition behind their impact were also analyzed, by performing multiple different runs, across various ranges of hyperparameter selections. Finally, the Q-values learned by the neural network were tested on artificially generated ranges of parameters, and somewhat poor generation was observed for ranges outside of what the agent has experienced. This environment is challenging, as the agent has 4 actions to choose from, and receives observations over a continuum space of 6 real numbers plus two boolean variables. Still, via these techniques, the agent can learn to successfully land the craft, for both train and test runs.

References

[1] A. G. Hado van Hasselt and D. Silver. Deep reinforcement learning with double q-learning. 2015.
[2] Y. B. Ian Goodfellow and A. Courville. *Deep Learning*. 2016.
[3] OpenAI. Gym.
[4] e. a. Peter Henderson. Deep reinforcement learning that matters. 2015.
[5] H. van Hasselt et. al. Deep reinforcement learning and the deadly triad. 2018.
[6] C. J. C. H. Watkins. Learning from delayed rewards. manuscript, 1989.
[7] e. a. Ziyu Wang. Dueling network architectures for deep reinforcement learning. 2016.