Lunar Lander Deep Reinforcement Learning

Brad Smith

I. Introduction

Current classical solutions to guidance and control problems require significant training on domain knowledge to implement carefully tuned products typically for specific applications. Reinforcement learning offers a possible alternative approach that benefits from generalization to provide solutions to similar sets of problems. It extends further as a natural framework to automate the tuning, through online learning, of the same vehicle to a different environment, set of flight conditions and/or requirements. The OpenAI Gym LunarLander-v2 agent was used to demonstrate successful reinforcement learning in the observable control design space [5]. Although what is presented is a simple, fully observable, constrained 3 degree of freedom environment, the methods used in this report are scalable to simulations with more complicated dynamic models.

II. Methods

A. The Lunar Lander Agent

The lunar lander OpenAI Gym problem objective is to successfully touch down as close to the middle of the landing pad as possible without sustaining damage (crashing). The lander's state is an 8-tuple, $(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R)$:

1. Horizontal position relative to the landing pad (x)
2. Vertical position relative to the landing pad (y)
3. Horizontal velocity $(\dot{x})$
4. Vertical velocity $(\dot{y})$
5. Roll Angle (θ)
6. Roll Rate $(\dot{\theta})$
7. Left leg ground contact (leg_L)
8. Right leg ground contact (leg_R)

In each state the lander can take one of four discrete actions:

1. Fire no engine $(NoPower)$
2. Fire left side engine $(f_{s,left})$
3. Fire main engine (f_{main})
4. Fire right side engine $(f_{s,right})$

Figure 1 depicts the lander with states labeled according to convention. The external forces that can be applied by the lander via its actions are shown. The side force is equivalent regardless of whether the left or right engine is ignited.

The lunar lander is a Markov Decision Process (MDP) where in each current state, *s*, the agent has the option to take an action, *a*. The action is fed into the Gym Lunar Lander environment, which returns the propagated new state, s' and a reward, *r*. The reward provided by the OpenAI Gym LunarLander-v2 environment is internally computed

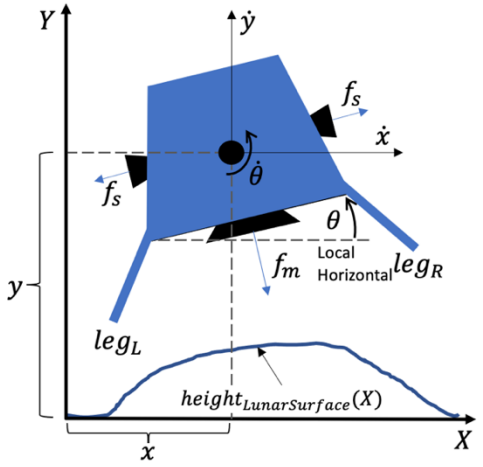

Figure 1: Lunar lander agent with sign conventions applied. The height of the lunar surface varies with each episode. Booleans for each landing leg trigger when the end of the leg intersects this function (and constrained by it). The center of the landing pad is the origin of the X-Y coordinate system.

according to (1). The variables m_{power} and s_{power} stand for the current main engine and side engine power, deducting reward from the lander for fuel usage. The difference of shaping between current and previous state in the "else" clause is *potential based shaping*.

$$\textbf{\textit{if}}\ (\textbf{\textit{landed}}): reward = +100 \tag{1}$$

$$\textbf{\textit{else if}}(\textbf{\textit{crashed}}): reward = -100$$

$$\textbf{\textit{else}}: reward = shaping(state_{current}) - shaping(state_{previous}) - 0.3m_{power} - 0.03s_{power}$$

Potential based shaping is a popular reward function approach in reinforcement learning to boost speeds of learning and help prevent the agent from looping on suboptimal reward feedbacks [6]. The shaping function is given by (2). The agent is only given a positive reward when moving to a state with "less negative" reward.

$$shaping(state) = -100\sqrt{x^2 + y^2} - 100\sqrt{\dot{x}^2 + \dot{y}^2} - 100|\theta + 10leg_L + 10leg_R| \tag{2}$$

Improved reward results when moving closer to the landing point ($\sqrt{x^2+y^2}$) while slowing down ($\sqrt{\dot{x}^2+\dot{y}^2}$). The final term provides shaping for zeroing out the roll angle θ and applies reward if the lander makes ground contact, but applies a penalty if contact is lost [5]. The problem is considered solved when the agent accumulates an average reward of more than 200 over 100 consecutive episodes.

B. Reinforcement Learning Approaches

Numerous methods for solving the lunar lander MDP agent were explored in search of an applicable and reasonable approach. Tabular Q-Learning methods were deemed insufficient given the continuous nature of the lunar lander's state space. Approximate methods for generalizing the state space include discretization (aggregation), linearization, and Deep Q-Learning (DQN). Discretization involves establishing buckets within which to insert portions of the state space, thus converting the problem into something that can be solved with tabular Q-learning methods. This approach was deemed undesirable due to risk in keeping track of the numerous state combinations and anticipated inefficiencies in training times because of the projected large size of the state space table.

Linearization was implemented with an on-policy approach during initial set up of the lunar lander solver but showed poor initial behavior. No agent learning was detected, given that losses never decreased, nor did accumulated reward returns increase over a long number of episodes.

Deep Q-Learning (DQN) using a feedforward neural network to approximate the lunar lander's action-value function was determined the most viable option for solving the lunar lander. Elements of the environment are dependent on one another, so the problem is nonlinear. Literature on Q-Learning is abundant, such as the work of Mnih 2017, which explores the use of Deep Q-Learning to train an agent to play Atari games [1]. Confidence in this down-selection was further motivated by Stanford research on the lunar lander problem with applied uncertainty where DQN was deemed the most successful [4]. Parameters from both Mnih 2017 and the Stanford uncertainty lunar lander approach were used to assist initial selection in the absence of neural network design experience [1,4].

C. Deep Q-Learning

Deep Q-Learning (DQN) uses temporal difference learning to update a neural network approximation of the action-value function (Figure 2). DQN is classified as a one-step method, where $TD(\lambda = 0)$, because learning is performed on a random sample of previous experiences of state, action, reward, new state combinations. This paradigm is called experience replay and is a major reason for the success of DQN. Experience replay consists of storing state, action, reward, new state, and a terminal flag (to indicate if the episode ended on the transition to the new state) in a circular buffer. At each learning step, a random batch of samples from replay memory are extracted and used to train the action-value neural network to learn the value of each action given a current state. This approach of random sampling provides fast learning, but also prevents correlations between the samples, which might lead to overfitting [2]. The target action-value neural network (with different weights) is used to compute the action-value function for the next state and is updated a predefined rate. This enables stable learning for the network and prevents the primary action-value network from being compared to its own updates [2]. A target update rate of every 5000 training steps was found a sufficient lag time.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2: Deep Q-learning with Experience Replay as defined by Mnih; modified to solve the lunar lander agent [1].

$$L_i(\theta_i) = \left(y_i - Q(s, a;\ \theta_i)\right)^2 \tag{4}$$

$$y_i = reward + \ \gamma max_{a'} Q(s', a';\ \theta_i^-) \tag{5}$$

Equations (4) and (5) are extracted from Figure 2 to emphasize the challenge faced with determining the appropriate method for defining them within PyTorch. This was identified as an element of the DQN algorithm that is unique and problem specific to be experimented with. Another key difference in DQN as implemented for the lunar lander was that preprocessing of experiences was not needed since the original use case required simplification of the Atari game 210x160 pixels for reasonable computations. This is not necessary with the 8-tuple state space of the lunar lander, so the following simplification is made: $\phi(s) = s$.

D. Neural Network Topology

A neural network structure needed to be defined to accomplish DQN with the lunar lander. A mapping from the state vector to estimated value of each action was established. The idea behind this was chosen primarily out of practical efficiency. When running on greedy policy, the agent feeds the current state through the network, receives a relative grade for each action, and choses the action with the highest grade. This was preferred over the other considered concept where a suggested action and state vector would be provided to the input layer and a single estimated action-value would be provided in the input layer. These were seen as mechanically the same solution, but the second approach would require $(number\ of\ actions)(feed\ forward\ comp.time)$ at each step. Although not significant in the 4-action lunar lander, in more complex action spaces, this would be inefficient, so the topology decision was made primarily in principle.

Hidden layers in neural networks allow for nonlinearities to be approximated [6]. The extent of research into hidden layer number halted when target reward values were achieved after moving from a single hidden layer to two hidden layers,

so the two hidden layer structure was held constant for hyper-parameter tuning.

The Rectified Linear Unit (ReLU) was chosen as a simple nonlinear activation function to account for the nonlinear nature of the lunar lander environment. Without a need for baked in binary comparisons of the value function, no activations were used on the output layer (such as the sigmoid).

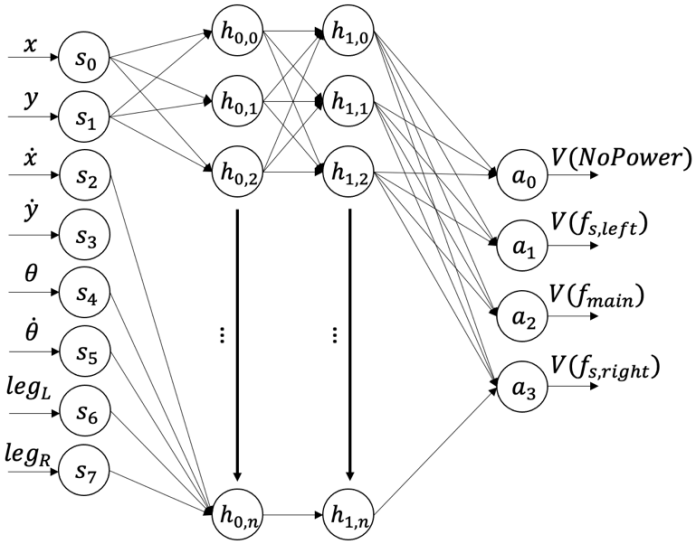

Figure 3: Lunar Lander Value Function Approximation Neural Network. The input layer is the 8-tuple state vector; the output layer is the estimated value of each action, given the current state. The hidden layer cell count, n, was chosen as a hyper-parameter to vary.

E. Implementation

The agent was implemented in python using the LunarLander-v2 gym module. The feedforward neural network was designed using the PyTorch Python library, a common tool used for neural network implementation.

III. Experiments

Numerous studies were conducted in development of the lunar lander. Informal studies resulted in the higher-level scoping of the design to show action-value function learning. This included loss function and ϵ-greedy scheduling, with the intent of validating neural network learning. Formal trade studies explored agent hyper-parameter optimization: learning rate (α), neural network hidden layer cell count, replay memory sample size, and discount rate (γ). In all formal studies, 600 episodes were used for training; derived from baseline results for what was inspected and qualified as convergence. In each plot, cumulative rewards are averaged over consecutive sets of 10 episodes to indicate trends more clearly. The primary metric for comparison was the average reward over the final 100 episodes, shown in parathesis in each figure legend. For each hyper-parameter study, the baseline was $\alpha = 0.001$, $\gamma = 0.99$, hidden layer cell count of 128, and batch size of 128. The baseline exists in each hyper-parameter figure and serves to provide the reward at each training episode while training the chosen agent design.

A. Neural Network Loss Function

PyTorch documentation was investigated for a loss function to accomplish the mechanics as defined in DQN from (4). Mean Square Error (*MSELoss*) was chosen since it represented the same error computed in (4). Implementation of *MSELoss* proved difficult because it wasn't clear at first how Mnih intended batch samples to be fed through the DQN loss function, which was never explicitly specified to be the mean of the square over all batch sample errors [1].

Initially, losses were computed on a per sample basis, and neural network loss never decreased, rapidly oscillating, and diverging in some cases. Eventually this detail of Mean Square Error was realized, since the intent of the loss function is to average the square of the errors over multiple samples into a single scalar value. The agent training implementation was modified to provide the entire batch of losses to the loss function. Thereafter, neural network losses as a function of episode count decreased and lander accumulated reward began to increase as well (generally). Although experience replay is helpful in speeding up learning and specifically decorrelating samples by taking arbitrary samples from a bank of experiences fundamentally, using a loss function that averages the losses applies an additional smoothing to capture general loss state of the neural network's accuracy [2].

B. Episode Length and ϵ-Greedy Schedule

The balance of exploration and exploitation in the lunar lander was studied to determine an appropriate scheduling method. In a ϵ-greedy learning philosophy where ϵ is a probability ($0 \leq \epsilon \leq 1$), such that a random action is taken at this probability, otherwise the highest valued action is taken (from neural network feedforward). By decaying ϵ in consecutive episodes, the agent transitions from taking random actions to explore the state space and rewards, while operating closer to optimal policy at later episodes. Exponential and linear decay were investigated with various curve parameters modified. In both cases, initial ϵ was set to 0.99 to encourage random actions early on in training.

With exponential decay, decay rate and the lower limit were tuned, and although signs of agent learning and improvement in the later stages of 600 episodes was detected, average accumulative reward did not approach the goal. The size of the lunar lander's state space prompted shifting focus to augmenting initial exploration. This was pursued with linear decay, since a more gradual decrease in learning would result in more exploration compared to exponential decay (more area under the curve = more random decisions = more exploration), Lowering ϵ by 1% each episode until a constant value of 0.01 yielded the most improved results for agent training and was selected as the ϵ-greedy method moving forward with agent optimizations [4].

C. Hyper-parameter – Learning Rate

Learning rate controls the magnitude that the neural network's loss is backpropagated back through to the weights. It is generally thought of as a measure of "how well the error is trusted". From Figure 4, the learning rate was varied by orders of magnitude and illustrates two key features: suboptimal convergence and learning oscillations. When $\alpha = 0.01$, the cumulative reward follows an oscillatory motion and never converges. This is because the learning rate is too aggressively applying losses in backpropagation to the neural network weights and continuously bouncing back and forth. On the other extreme with $\alpha = 0.0001$, convergence to a very suboptimal value function is seen, showing that a naïve learning rate doesn't allow for significant enough weight changes to exit suboptimal regions. Splitting the difference

was the most promising learning rate value of 0.001, which aided in achieving an average accumulated reward in the last 100 learning episodes of just under 200.

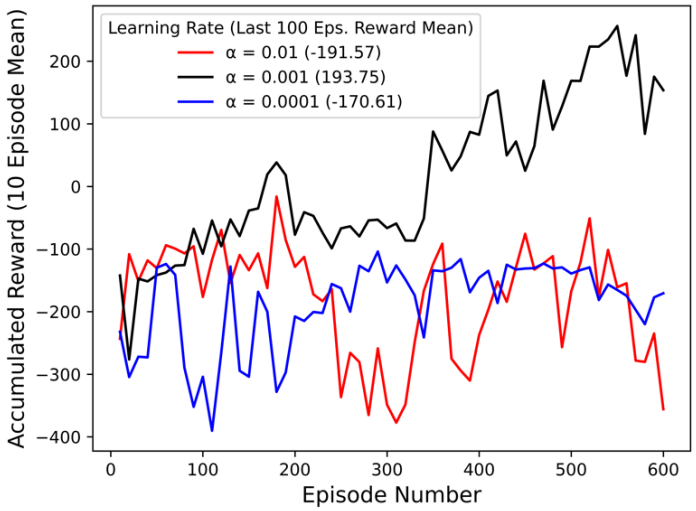

Figure 4: Learning rate variations over 600 episodes. The baseline design was $\alpha = 0.001$.

D. Hyper-parameter - Hidden Layer Cell Count

Although a native CPU was used for lunar lander agent development, NVIDIA GPU efficiency recommendations for hidden layer cell counts to be divisible by powers of 2 (64 to 256) was followed as a general guideline [3]. This idea was backed by an approach taken to solve the lunar lander with applied uncertainty [4]. Figure 5 shows the effect of varying hidden layer cell counts below advisable sizes for successful generalization by training with a hidden layer cell count of 32.

This study showed that in terms of trends, the overall generalization achieved through a multi-hidden layer neural network was not *significantly* impacted by the cell count, which was somewhat surprising. The 32-cell hidden layer had a lower cumulative reward over the last 100 episodes, but the agent was improving. Although determined a less sensitive hyper-parameter, the 128 and 256 cell counts were down selected as the top candidates to be compared going forward.

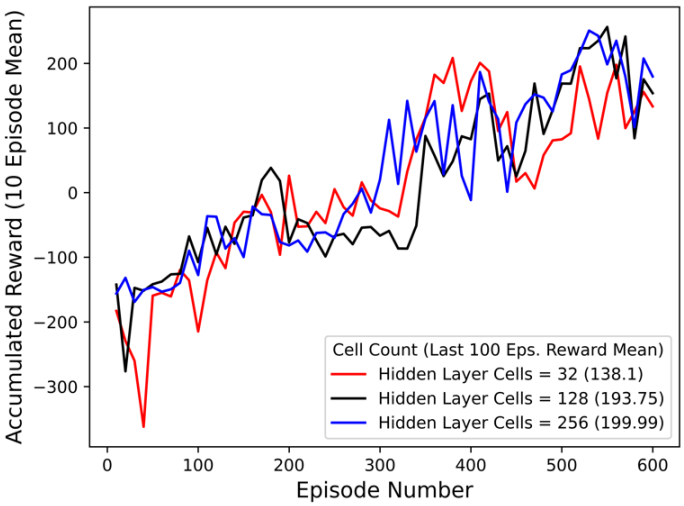

Figure 5: Hidden layer cell count variations over 600 episodes. The baseline design was the hidden layer cell count of 128.

E. Hyper-parameter – Replay Memory Batch Size

As mentioned, a crucial element is DQN is replay memory. Once a stable solution was found, a sensitivity trade was completed to look at the effects of varying the batch size to train the neural network on. The size of replay memory was held constant at 10,000. When varying batch size according to Figure 6, a smaller size of 32 imposed substantial degradation to performance of the agent and exhibited oscillating rewards. Sampling over episodes showed a maximum number of steps in an episode to be 500. The belief for worse performance at smaller batch sizes seems to be very problem specific. The belief is when the fraction of memories to number of possible steps in an episode gets too small, representation of the full scope of the state-action space is lost, and the ability to train is obstructed by a lack of information.

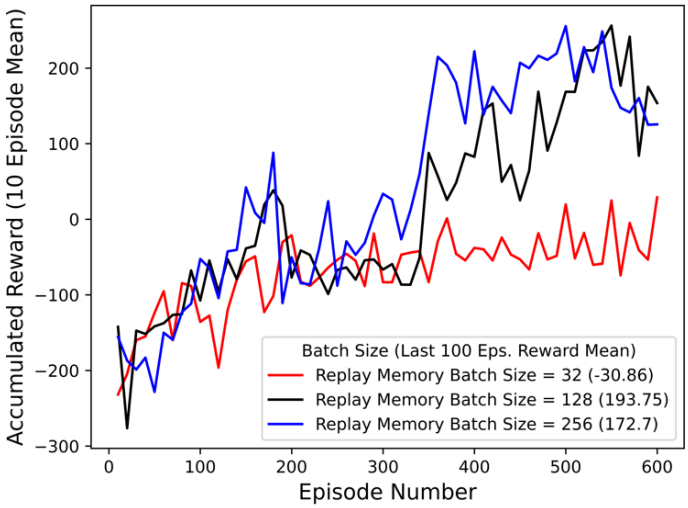

Figure 6: Batch size variations over 600 episodes. The baseline design was a replay memory size of 128 samples.

With large batch sizes in relation to the max total steps in an episode, this effect is not seen, and the action-value approximator learns. Along with a larger average over the last 100 episodes in comparison, the batch size of 128 was selected over 256 partially to decrease training time.

F. Hyper-parameter - Discount Rate

The discount rate (γ) determines the present value of future rewards for the MDP [6]. Shown in (5), γ scales the maximum of the target neural network's output for the next state in a replay memory tuple, controlling the horizon for how far into the future value information is propagated. The closer γ is to 1.0, the greater the importance of future potential reward. From γ variations seen in Figure 7, the gains from maintaining a far horizon are made evident. Agent performance quickly degrades from $\gamma = 0.99$ to $\gamma = 0.75$ and even lower for $\gamma = 0.5$. The agent becomes more myopic, meaning it will accept smaller rewards that don't take as long to get to than considering the greater picture of the reward for landing successfully [6].

The benefit of a high discount rate makes sense because the lander's greatest reward is achieved at the end of the episode if it successfully touched down at the landing platform (+100). The best reward for the lunar lander comes at termination, so it is important to maintain a farsighted agent for the specific case of the lunar lander. This thought process was used in initially setting the baseline $\gamma = 0.99$, and the

training data result from Figure 7 support this hypothesis, so the baseline design choice was maintained.

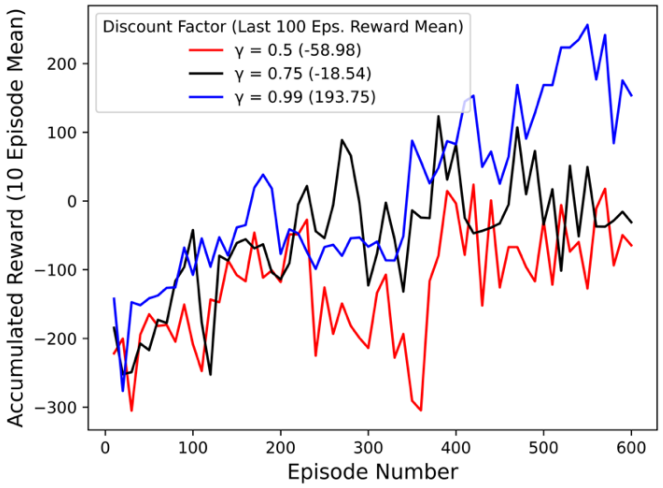

Figure 7: Discount rate variations over 600 episodes. The baseline design was $\gamma = 0.99$.

IV. Results

The goal of the lunar lander problem is to train the agent to be successful in generating an average cumulative reward of >200 over 100 consecutive episodes. The experimental trade studies introduced in the previous section narrowed down the following hyper-parameters: $\alpha = 0.001$, $\gamma = 0.99$, and $BatchSize = 128$.

Hidden layer cell count was not pulled directly from the experiment results in Figure 5 as the best choice. Further comparisons of 128 and 256 count hidden layer cells when running a 100-episode test set showed that 9/10 times, the former yielded higher average cumulative rewards (+27 on average across the 10 seeds). It is believed that overfitting resulted in the larger cell count case such that it was not well equipped to learn from samples not included in replay memory.

It is also possible that because of the larger number of weights, small variations in state that might not have different true optimal actions result in variations of action-value approximations just enough to select non-optimal actions. This can be thought of as a potential issue of resolution. Figure 8 shows the best observed results for the 128-cell hidden layer, in which only 6 episodes crashed and 75% of the episodes resulted in a cumulative reward above the target of 200, successfully achieving the goal of the OpenAI Gym Lunar Lander Reinforcement Learning agent. Because of this analysis and result, the 128-cell hidden layer neural network was chosen for the final lunar lander design.

The successful lander is a substantial result considering that no domain knowledge was implemented in the lander action selection logic and mapping from states to actions were achieved purely on Deep Q-learning using experience replay. Some future work is still required to further optimize and improve the lander agent, including investigating automated optimization for hyperparameters and examining neural network topologies and activation functions that might more accurately correlate the nonlinearities of the lunar lander environment to clear up the cases where failures did result when running the trained agent. A study into methods for adding learning rate decay might help with faster agent training. Lastly, more complex exploration/exploitation strategies beyond ϵ-greedy could yield benefits, such as looking into ways to provide more episodic level feedback to augment the action-value functional approximation.

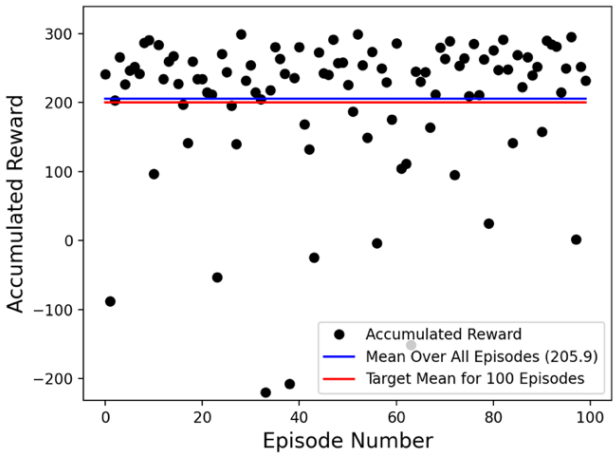

Figure 8: Average accumulated reward for the selected trained lunar lander hyper-parameter values. Target mean is 200.

V. Conclusion

Deep Q-learning, formalized by researcher Volodymyr Mnih, was originally applied to learning how to play Atari games [1]. This report illustrated that applying the same core concepts of DQN, can, with modifications to neural network structure and hyper-parameter values, achieve a valid solution in the context of a completely different domain, with a different reward structure, states, and actions. The lunar lander was the means to showing this powerful result. This agent further displays reinforcement learning as viable solution for success in aerospace guidance and control problems. A future version of the lander may require different release altitudes, uneven landing platforms, or an upward pointing thruster to close the altitude error quicker. Standard approaches to increases in design scope would require extensive trade studies and possibly time-limited subject matter experts to re-optimize the lander. Reinforcement learning in guidance and control therefor not only can provide unique and optimal solutions to problems; it is an inherent framework within which to re-optimize the agent given design changes, and much of the process is automated.

References

[1] Mnih et. al. *Human-level control through deep reinforcement learning*. Macmillan Publishers Limited. (2015). https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf.

[2] Roderick, M., MacGlashan, J., Tellex, S.. *Implementing the Deep Q-Network*. Human To Robots Laboratory. (2017). https://arxiv.org/pdf/1711.07478.pdf

[3] *Deep Learning Performance Documentation*. NVIDIA. (2021). https://docs.nvidia.com/deeplearning/performance/dl-performance-getting-started/index.html.

[4] Gadgil, S., Xin, Y., Chenzhe, X. *Solving the Lunar Lander Problem under Uncertainty using Reinforcement Learning*. IEEE Southeast2020. (2020). https://arxiv.org/pdf/2011.11850.pdf.

[5] *Lunar Lander V2*. OpenAI Gym. [accessed 2021 June]. https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py

[6] Richard S Sutton and Andrew G Barto. *Refinforcement Learning: An Introduction.* Second Edition. MITpress. (2020). http://incompleteideas.net/book/the-book-2nd.html