CS 7642 - Project 2

Austin Lane
alane31@gatech.edu
Commit hash: af89db994aeb166e8a6e90ae46e7c356e226826e (Branch: "main")

***Abstract*—Reinforcement learning (RL) by function approximation with neural networks is an extremely powerful learning technique in environments with continuous state spaces. In this study, the Deep Q-Network (DQN) algorithm and several of its variants were implemented to solve the Lunar Lander environment. Training and evaluation experiments were conducted to study the effect of 3 hyperparameters on agent performance (learning rate α, discount factor γ, and the size of the replay buffer). Additional automated hyperparameter tuning was also performed to identify the best possible set of hyperparameters for each DQN variant. This allowed the performance of each agent configuration to be directly compared, and also provided some insight into how the range of viable hyperparameters changes as the DQN algorithm is extended. The theoretical advantages and disadvantages of each variant are discussed, and finally some opportunities for future work are considered.**

I. Introduction

Autonomous control in continuous, stochastic state spaces represents a grand challenge in artificial intelligence research and epitomizes the full reinforcement learning (RL) problem. To effectively make decisions, RL agents must learn from sequential, sampled feedback which is not validated by a supervisor or a ground truth model of the environment.

Among model-free RL algorithms, Q-learning and its variants have been popular choices for value-based learning since being introduced in 1989 by Watkins. [1] Q-learning has its roots in temporal difference learning, which makes use of successive predictions in time to estimate some future reward. Famously, Q-learning was the basis for the DQN algorithm introduced in 2013 [2], which successfully leveraged deep neural networks to learn sophisticated strategies for playing Atari games with superhuman performance.

Unlike tabular methods, RL algorithms based on neural networks are capable of generalizing in continuous state/action spaces and can approximate arbitrarily-complex value functions and policies. Their utility comes with a cost; convergence to an optimal policy isn't guaranteed. In fact, even basic stability in training is difficult to achieve without carefully modifying the corresponding tabular algorithms. [3]

Q-learning and DQN are both off-policy algorithms, meaning that the policy the agent learns from experience (target policy) is different from the policy used to generate that experience (behavior policy). Off-policy algorithms benefit from their ability to learn from deep past experience, making them more sample-efficient than corresponding on-policy algorithms. In addition, established training methods from supervised learning can be leveraged, like batching for stochastic gradient descent updates. However, to scale up and incorporate neural networks, DQN in particular needs additional strategies for 1) stable training [4] and 2) managing the overestimation of value estimates, a problem inherent to Q-learning. [5] Since the seminal DQN paper, multiple studies have demonstrated additional improvements to the base DQN algorithm.

Sequential experiences are naturally correlated, and small changes in behavioral policy can have drastic impacts on the moment-to-moment data distribution that an agent experiences while learning. To combat both of these issues, the authors of DQN introduced two methods for decorrelating and stabilizing training data: the experience replay buffer and the use of a separate target network. The experience replay approximates the iid nature of supervised learning by randomizing the data used for training, and the size of the buffer is an important parameter for controlling the diversity of data the agent learns from. The target network is used to estimate the maximum over future action-values, and is identical to the original network but only updated periodically, which provides a temporary fixed point for the agent to perform a more stable gradient update.

Shortly after demonstrating the effectiveness of the target network in learning to play Atari games, the same authors also proposed a solution for another issue often seen in Q-learning. Because the TD target is calculated by maximizing over future action-values, the agent tends to learn an overly optimistic value function. In certain environments (like stochastic ones), this can lead to slow learning and "overoptimism in the face of apparent certainty", [5] which is a less-than-desirable strategy for proper exploration and can lead to suboptimal policies. To learn a more balanced action-value function, the responsibility of choosing the state-action with the highest value and reporting the value of that state-action is divided between two networks; in this case, the target network and the online network. This "you cut, I choose" mechanism leads to a more balanced estimation of future Q values and helps reduce bias in training.

For this study, four of the most well-known extensions for DQN were implemented for the Gym Lunar Lander environment and evaluated for performance, sample efficiency, and hyperparameter tuning window:

1) Replay buffer
2) Target network with hard update
3) Double DQN
4) Double DQN with soft network update

II. Methods

The basic architecture for the training, evaluation, and tuning experiments was implemented in Python, using the PyTorch library for the construction and training of the neural network. Typical training experiments consisted of 500 episodes, with evaluation runs performed every 50 episodes using a model checkpoint. Hyperparameter tuning was performed initially by hand to find reasonable ranges for each parameter. Automated tuning experiments were then carried out using Optuna. Each automated study typically consisting of 200 trials, 500 episodes each, using 50-100 burn-in episodes and 5 warm-up trials to stabilize the agent performance before pruning. To demonstrate the effect of changing hyperparameter values on agent performance, the best-performing set of hyperparameters for a particular DQN configuation (replay buffer & gradient clipping, no target network) was used as a baseline. Values for α, γ, and buffer size were systematically varied and trained using 5 trials of 500 episodes each.

III. Results and Discussion

A. Baseline DQN configuration

This section of the discussion will consider the DQN variant from the 2013 paper [2] which does not use a separate network for computing the TD target in the training update. This variant was found to perform as good or better than any other tested for this study, and relies on fewer hyperparameters which simplifies analysis. Figure 1 shows the results from a single training run for this DQN variant.

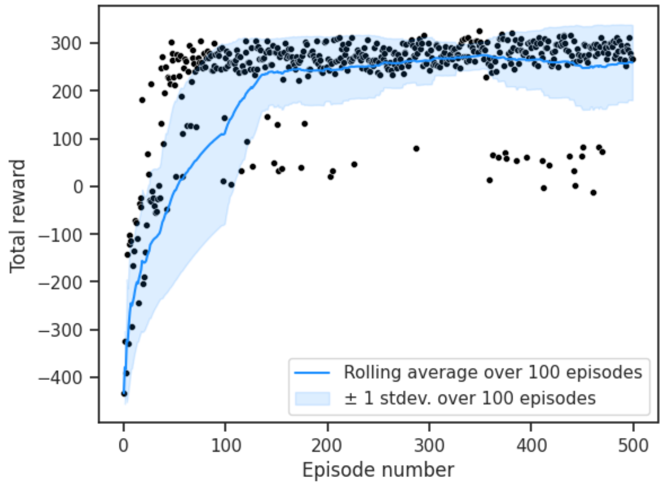

Fig. 1. Training results for base DQN agent. Average and standard deviation are calculated over a rolling window of 100 episodes.

This particular training run was exemplary within this study; most agent configurations and hyperparameter combinations only reached a rolling average of 200 total reward after 200 episodes or more. In fact, Fig. 3 shows that the mean rolling average for this agent (magenta trace, $\alpha = 2e{-4}$ did not perform as well on average. This particular configuration and hyperparameter set is also notable for its stability after solving the environment, not exhibiting any signs of catastrophic forgetting in extended training experiments (>1000 episodes).

To validate the training performance in Fig. 1, a checkpoint of the model was saved every 50 episodes and immediately tested in evaluation (no training or exploration). Results from the best model identified by these evaluation experiments is shown in Fig. 2.

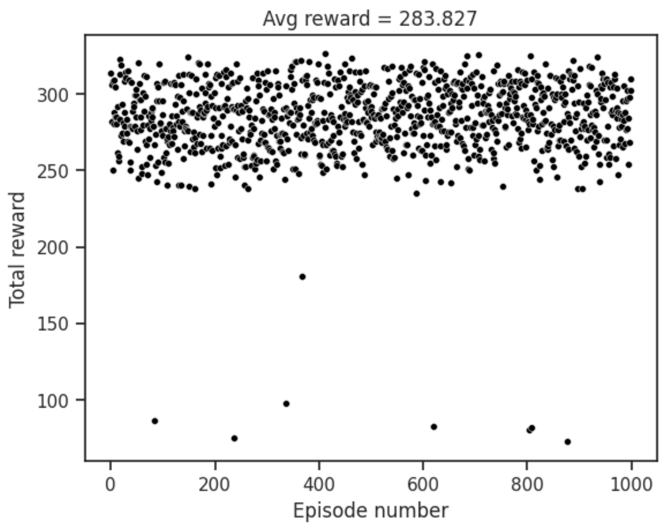

Fig. 2. Evaluation results for base DQN agent, using a model saved from the end of the training run in Fig. 1.

This model exhibited the highest evaluation score of all model checkpoints from this training session, and it happened to be the model saved after all 500 episodes. Remarkably, the agent's average total reward is nearly 284 over 1000 episodes, with less than 1% of episodes earning less than 200 reward. This is characteristic of an agent that has learned to land very quickly with minimal thruster use, which was verified by watching rendered episodes during a separate evaluation session. This type of policy is difficult to learn, since the agent must use its thrusters only when necessary to avoid crashing and efficiently move to the landing pad. Occasionally, the agent still crashes, which can be seen in Fig. 2 as the episodes earning around 100 total reward. The gap of 200 between sucessful and failed landings is the lost +100 reward for landing and the -100 penalty incurred by crashing.

What explains this agent's performance? This question is addressed in the next section by considering the effects of three important hyperparameters: the learning rate, α, the discount factor, γ, and the size of replay buffer. In the last section, the effect of the agent's configuration on training performance is discussed, and some lessons from the hyperparameter tuning process are also included.

B. Learning rate, α

The learning rate (α) plays a central role in the training update loop by controlling the magnitude of the NN parameter updates. Fig. 3 demonstrates the sensitivity of the DQN algorithm to this parameter. The viable range of α values for stable training is small for this agent configuration, possibly spanning just an order of magnitude [e-05, e-04]. Larger values of α learn faster initially, but quickly become unstable. As

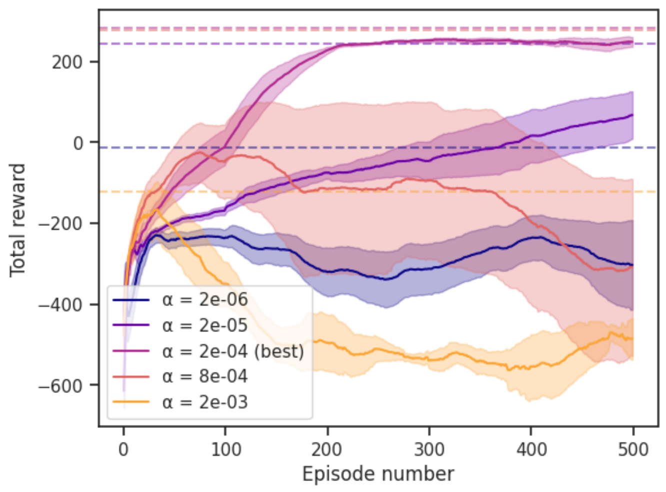

Fig. 3. Effect of α (learning rate) on training performance. Solid lines represent the average over a rolling window of 100 episodes, averaged over 5 consecutive training runs. Shaded regions above & below the mean rolling average represent ± 1 standard error. Dashed lines represent the evaluation score (average total reward over 100 episodes) for the best-performing checkpoint model from the 5 training runs.

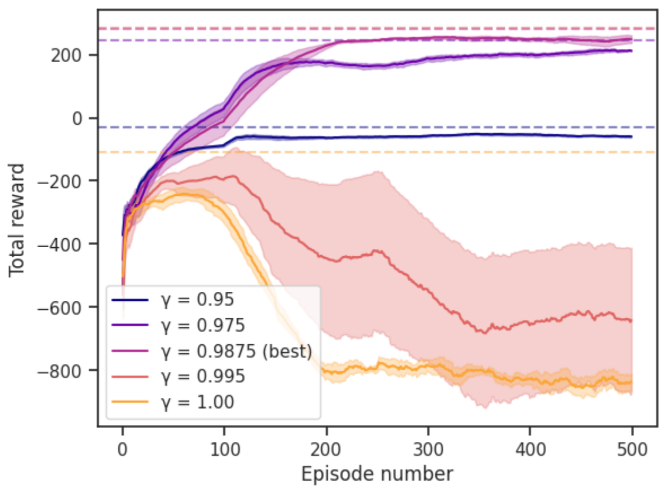

Fig. 4. Effect of γ (discount factor) on training performance. Solid lines represent the average over a rolling window of 100 episodes, averaged over 5 consecutive training runs. Shaded regions above & below the mean rolling average represent ± 1 standard error. Dashed lines represent the evaluation score (average total reward over 100 episodes) for the best-performing checkpoint model from the 5 training runs.

discussed earlier, the near-optimal policy requires a delicate balance between speed and safety of the lander. Large steps in the predicted Q-values may never find this balance, and the final section of this report will discuss why changing the agent's architecture may shift the range of viable α values higher.

However, the data from the larger α values also suggest a path to improve the current agent configuration. A dynamic learning rate schedule could be leveraged to decrease α over time, combining the faster improvement in training performance seen for $\alpha = 8\text{e}{-4}$ with the long-term stability of the more optimal α setting.

Note that the smallest α value also resulted in poor performance that did not improve over time. This could be a manifestation of the learning process becoming trapped in a local minimum, unable to take the step size required to change its distribution of experiences and learn more about the environment.

Finally, Fig. 3 also demonstrates that the parameter space for stable training is not identical to evaluation performance, as demonstrated by the fact that $\alpha = 2\text{e}{-4}$ and $\alpha = 8\text{e}{-4}$ both have similar peak average evaluation scores for a checkpointed model. Despite having a much lower average training performance, the agent trained with $\alpha = 8\text{e}{-4}$ did have one lucky training session out of 5 that produced a stable agent with good performance.

C. *Discount factor, γ*

The discount factor γ is responsible for controlling the agent's perceived value of future rewards. This hyperparameter has important effects in environments where agents are able to receive small, persistent rewards for intermediate behaviors, such as Lunar Lander. Fig. 4 showcases the viable range of γ values for this agent configuration. When γ is too small, the agent is myopic; it does not learn to properly value future rewards, such as the +100 reward for landing successfully. One of the interesting consequences is the hovering behavior, which can be observed by plotting the per episode step count during training (Fig. 5). With the lowest value of γ, the agent consistently fails to land within a 1000 episodes. When $\gamma = 1$, the agent's performance still suffers during training, likely because it ignores the importance of the reward shaping built into the Lunar Lander environment. This short-term feedback is very important for guiding the agent initially during the first steps of the episode, and the algorithm must incorporate some short-term awareness to acquire an effective policy. For this environment, however, it appears advantageous to push gamma as close to 1 as possible to maximize the importance of landing, without completely ignoring the short-terms signals meant to guide landing.

D. *Replay buffer size*

A well-sized replay buffer balances exploration of previously visited states while providing the most relevant information to the agent as training progresses. If the buffer is not large enough, information about past states is eroded too quickly, the diversity of training data becomes stagnant, and the agent may suffer from overfitting on data from its current policy. Make the buffer too large, and the data may become stale; the distribution under the current policy is difficult to learn from due to the noise of irrelevant past experience.

Fig. 6 illustrates the effect of tuning the size of the replay buffer over a large range. Completely coincidentally, the data in Fig. 6 bears a striking resemblance to the data shown in Fig. 5. This provokes a natural question about the similarity of the two hyperparameters, although there are important differ-

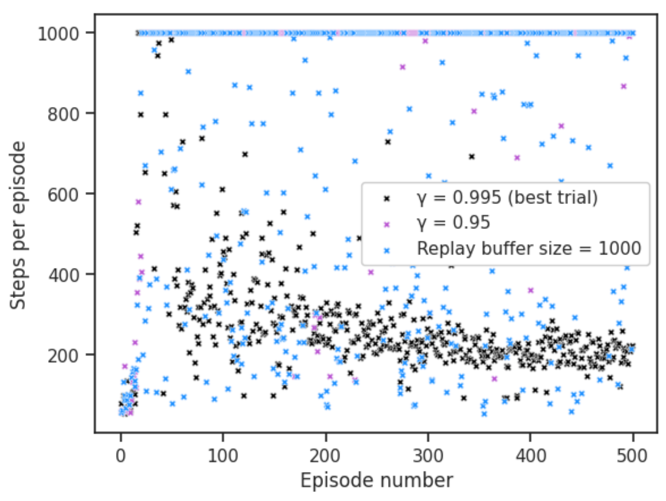

Fig. 5. Effect of γ and replay buffer size on episode length (in timesteps). Episodes are automatically truncated after 1000 episodes, and repeated timeout behavior indicates that the agent has learned to hover indefinitely instead of land.

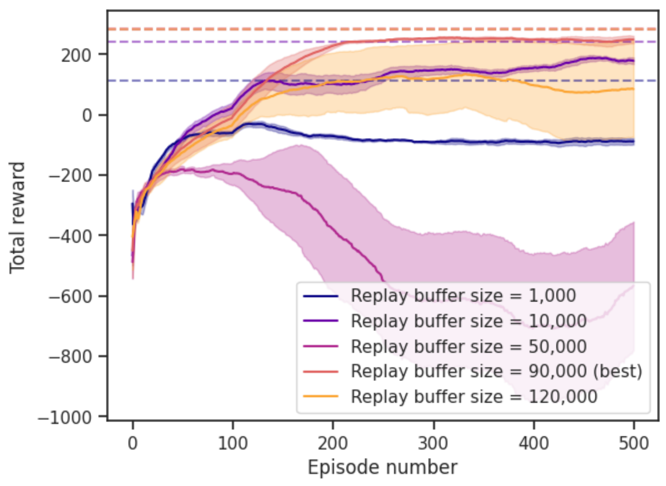

Fig. 6. Effect of the replay buffer size on training performance. Solid lines represent the average over a rolling window of 100 episodes, averaged over 5 consecutive training runs. Shaded regions above & below the mean rolling average represent ± 1 standard error. Dashed lines represent the evaluation score (average total reward over 100 episodes) for the best-performing checkpoint model from the 5 training runs. The size indicates the maximum number of experience tuples that can be stored at one time (state, action, reward, next state, termination criteria).

ences between the two plots. One significant difference is the relatively inconsistent performance trend as the replay buffer increases in size. Values of 10,000 and 90,000 experiences perform the best in this experiment, but performance seems to degrade for the middling value of 50,000. It is difficult to rationalize why 90,000 experience capacity promotes a much more stable training environment compared to 50,000; assuming an average episode length of 500 steps, it would take nearly 100 episodes for the agent to even notice a difference between the size of the two replay buffers. This could be a case where the 50,000 buffer size training simply had several unlucky runs, as indicated by the large standard error in the rolling mean. The initially faster learning rate of the 10,000-size replay buffer compared to 90,000 possibly suggests that the smaller buffer size helps the agent escape its initially random action policy faster, which allows it to improve more quickly. However, the limited buffer size also reduces the diversity of experience later in training, which causes the agent to stagnate. The 1,000-sized buffer is so small that it seems to quickly becomes saturated with hovering experience, and this is verified again by observing the steps per episode in Fig. 5.

E. *Agent configurations*

As stated in the introduction, several DQN variants and extensions were implemented as part of this study. Each variant was tuned to find the best hyperparameters possible, with the goal of comparing 1) training/evaluation performance and 2) the process window of viable hyperparameter values for each variant. Fig. 7 shows the mean rolling average & peak evaluation performance for the six implemented variants.

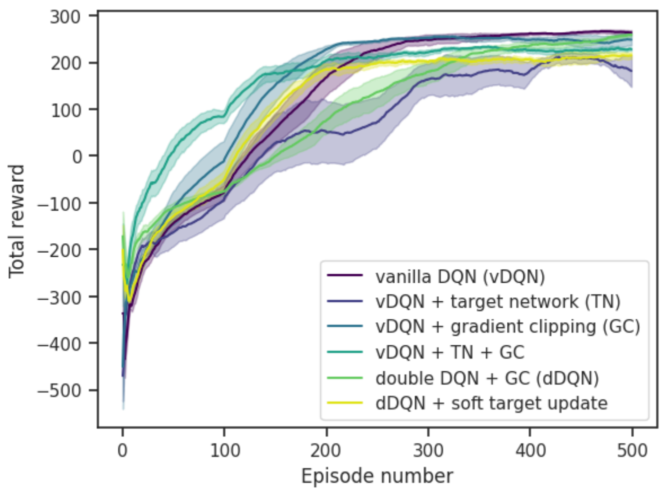

Fig. 7. Training performance for each of the 6 DQN variants tested in this study. Solid lines represent the average over a rolling window of 100 episodes, averaged over 5 consecutive training runs. Shaded regions above & below the mean rolling average represent ± 1 standard error. Peak checkpoint evaluation scores for each variant were all in the range of 270-283 total reward averaged over 100 episodes.

Target network updates and Double DQN were originally proposed as ways to stabilize and improve the base Q-learning algorithm. In practice, however, the best training performance was obtained just by using a replay buffer and gradient clipping. It was also noticeably more difficult to find high-performing hyperparameter combinations for agetns with separate target networks. When good combinations were found using variants with either the hard or soft target network updates, the hyperparameters were always close to full network updates on each step (essentially having no target network at all). For hard updates, this mean target network updates every 20-50 steps (compared to every 10,000 steps in [4]); for soft updates + double DQN, the best value of τ was close to 1.

These results may be best explained by understanding the hyperparameter tuning process in Optuna. To save time

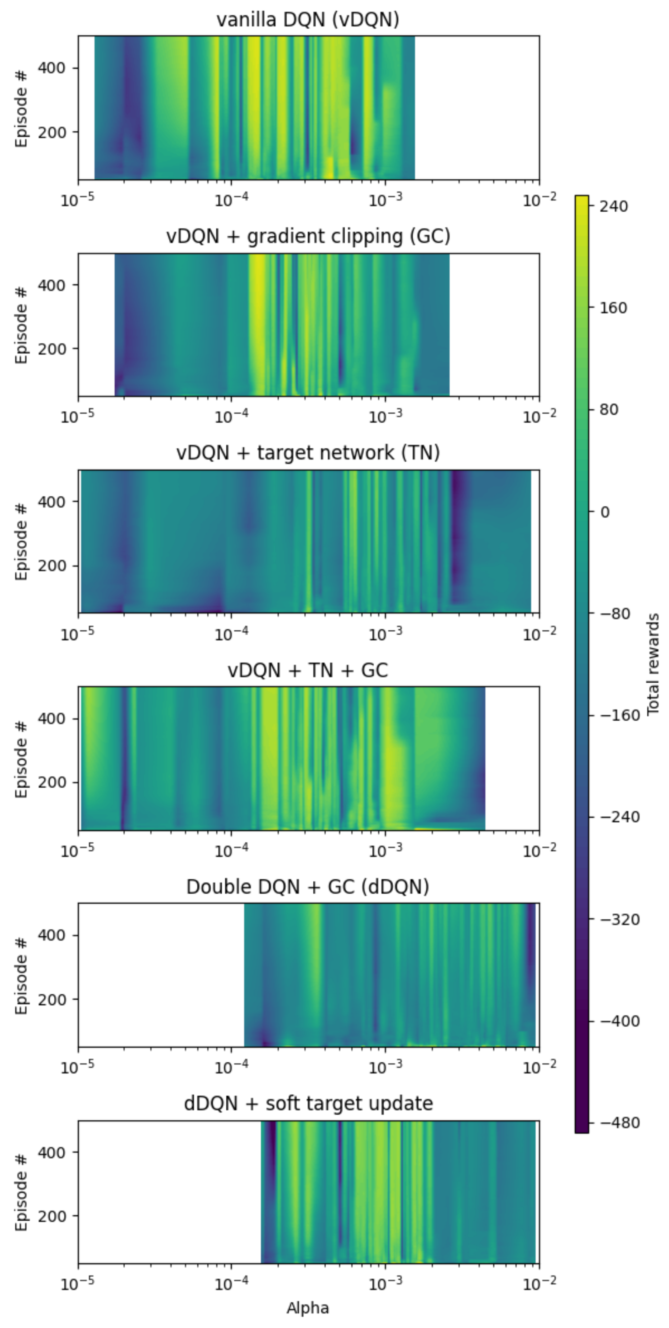

Fig. 8. Tuning results for the learning rate, α, across multiple agent configurations. A single tuning trial can be visualized as a vertical line at the value of α for that particular trial. Values are linearly interpolated between values not sampled, and white areas indicate values that were either below or above the lowest/highest selected value for the entire tuning study.

when running the experiment, Optuna provides a pruning functionality, which can end a trial prematurely if the intermediate performance does not meet a certain threshold. Training performance can be noisy in the beginning, especially for DQN variants with target network updates, since performance will not start to improve until a sufficient amount of updates & learning have taken place. If the pruning conditions are not set up carefully, the pruning algorithm may incorrectly cancel trials with objectively good hyperparameter combinations early on, resulting in false negatives and a less reliable model of the parameter space. Anecdotally, it was observed that tuning on variants with target network updates were pruned more agressively. One remedy to this problem is increasing the number of "burn-in" episodes for each trial, such that early pruning decisions are based on more data, but this comes at the cost of longer experiment runtimes. If time permitted, it would have been interesting to perform a more in-depth study on the hyperparameter tuning process, its variability and sensitivity to its meta-parameters.

The data collected by each tuning study can be difficult to decipher, especially since the optimization process changes every hyperparameter for each trial. However, by plotting a heatmap of the training performance for each agent configuration, it's possible to appreciate the impact that each configuration has on the range of individual hyperparameters that produce good training performance. For instance, the data in Fig. 8 suggests that higher learning rates are favored as more extensions are added to base DQN that promote stability during training. This satisfies the intuition that more agressive learning can take place when more "safeguards" are put in place to prevent divergent behavior.

F. Future work

Given more time, it would be interesting to implement more of the DQN extensions that culminated into Rainbow DQN. [6] However, given the performance similarity of the variants that were already used in this study, it's not clear that Lunar Lander has enough complexity or difficulty to really observe the full potential of each algorithm. I would also be interested in exploring the MuZero algorithm [7] which incorporates planning by using a learned model to predict and evaluate future actions before taking them.

References

[1] Watkins, C. J. C. H. "Learning from delayed rewards." (1989).
[2] Mnih, Volodymyr, et al. Playing Atari with Deep Reinforcement Learning. arXiv (2013).
[3] van Hasselt, Hado, et al. Deep Reinforcement Learning and the Deadly Triad. arXiv (2018).
[4] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.
[5] van Hasselt, Hado, et al. Deep Reinforcement Learning with Double Q-Learning. arXiv (2015).
[6] Hessel, Matteo, et al. Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv (2017).
[7] Schrittwieser, Julian, et al. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. Nature 2020, 588 (7839), 604–609.