CS7642: Project 2

Rory Michelen | rmichelen3@gatech.edu | Git Hash: d3395601f33f5342ad88fe9ccb286f239a2ec495

I. Introduction

This report will explore an implementation of Deep-Q Learning with experience replay and a fixed Q target. It will first discuss the foundations of Deep-Q Learning, including q-learning, function approximation, and stochastic gradient descent. It will then introduce experience replay and fixed target network. Then a theoretical discussion of hyperparameters, followed by an analysis of results using a range of hyperparameter values.

II. Problem Discussion

The objective of this project is to solve the Lunar Lander environment in OpenAI Gym. The Lunar Lander environment is a continuous state space with the objective of landing a drone-like object on a landing pad. The drone has 2 legs and can take four actions at a given time step (1) do nothing, (2) fire left engine, (3) fire right engine (4) fire main engine.

The agent receives rewards for the following: (a) 100 points for coming to rest without crashing, (b) -100 points for crashing, (c) 100-140 points for reaching landing pad at zero speed, (d) 10 points for each leg that makes ground contact, and (e) -0.3 for each engine fire action taken.

States are represented using an 8-tuple. The first 6 values represent the location, velocity, and angular velocity of the agent in the x and y dimensions. The last two values are binaries that equal one if the left or right legs are in contact with the ground.

III. Model Selection

This section will outline the considerations that factored into the selection of the author's final model.

A. Q-Learning

Q-Learning is an off-policy TD Control algorithm. It learns the values of taking a greedy action for each given state by exploring the state-action space. While Q-Learning estimates the value of a greedy action, the algorithm does not necessarily follow a greedy policy. This is why the algorithm is referred to as "off-policy".

B. Continuous State Space

As mentioned earlier a state in the Lunar Lander environment is described using 8 dimensions. 6 of the 8 dimensions are continuous variables. Traditional Q-Learning requires a table of all possible states. Theoretically, an environment with continuous dimensions has an infinite number of states. Representing a continuous space would require discretizing the 6 continuous variables by truncating each of the continuous dimension values.

However, the state space of the Lunar Lander environment is very large. Therefore, to use traditional q-learning would require an incredibly large q-table or a significant loss in precision of the state space.

C. Approximation: Motivation

While using a q-table in Q-learning is the simplest option, it is not practical for large and continuous state spaces. Another option is to use a function approximator to estimate Q(S,A). Function approximation enables an agent to estimate Q(S,A) based on a set of features. This provides us with a generalization of the Bellman Equation:

$$Q(S,A) = F(w^a, F(s))$$

such that $F(s)$ is a mapping of our state space to a set of features and w^a is the weight vector corresponding to these features. Note that the superscript, a, implies that the weights are dependent on the given action. This enables the agent to establish relationships between states with shared features. Therefore, learning at one state can be applied to other states, even if they have not yet been visited. This can significantly reduce computational complexity, since the state space can potentially be described with a set of features much smaller than a list of all states.

D. Stochastic Gradient Descent

Even with an excellent feature set, the values of the features must be learned. Sutton describes the method of Stochastic Gradient Descent (SGD) to improve estimates of W, the feature weights. Similar to Gradient Descent, SGD intends to minimize some function iteratively. In this context, Sutton defines a loss function as:

$$V\bar{E}(w) = \mu(s)[V_\pi(s) - \hat{V}(s,w)]^2$$

Such that $\mu(s)$ is a function describing the relative importance of each state. To improve W, $V\hat{E}$ must be minimized. This can be achieved through gradient descent. However, the loss function is dependent on the optimal policy, which is unknown during learning. This can be resolved by substituting the optimal policy with some approximation via bootstrapping- a technique that is familiar to reinforcement learning. This substitution results in noise, which is why the algorithm is referred to as a stochastic version of gradient descent. Under certain assumptions, SGD will converge to a local optima for W.

E. Divergence

With the introduction of SGD and function approximation, there are now two updates occurring at each time step of the Q-learning algorithm. (1) Q(s,a) is updated based on the bellman equation and (2) W is updated via SGD. However, updating W implies that estimates of the value function are updated for many, if not all other states. Therefore, the update of Q(s,a) and W are dependent on one another. If both estimates are converging towards their true values, there is no issue. However, this is not guaranteed. If our initial estimates for W are not sufficient, this will feed a low-quality update for Q(s,a) which will feed the consequent update of W. If this cycle continues, Q(s,a) will diverge from its true value with each step.

F. Linear Function Approximation

One special case of function approximation is the use of linear functions. In this paradigm, the value function is estimated as a linear combination of the weight vector, W. This simple approximation method has two drawbacks. Firstly, it requires the feature set to be manually defined. It requires domain knowledge and creativity for a human to identify the correct features to accurately describe the relationship between states. Secondly, the relationships between states may not be linear, therefore linear approximation will not be sufficient has could lead to the divergence scenario described above.

G. Deep Q Learning

Another special case of function approximation is the use of a neural net to define the weight vector W. A neural net is a method of mapping a set of inputs to a set of outputs. This mapping is achieved by a network of hidden layers that use potentially nonlinear functions to map inputs to outputs.

In the context of Q-learning, we can define a neural net such that the input nodes describe the current state of the environment and each output node represents the value of taking a specific action at that state. With multiple hidden layers each consisting of many nodes, this method is capable of describing the complex relationships between states that is present in environments such as the Lunar Lander. Further, it can do so without manually defining features to be considered. This method is referred to as a Deep Q-Network (DQN).

H. Experience Replay

Recall that in Q-learning, an update is made to Q(s,a) based on the latest action taken. After this information is incorporated into the estimate of Q(s,a) (and W), this experience is forgotten. In the DeepMind reinforcement learning course, Silver discusses the use of experience replay in order to make more efficient use of the experience at each time step. Experience replay introduces two new concepts to the DQN algorithm. Firstly, the agent stores the latest experiences in a cache instead of forgetting them after the update. Secondly, instead of using the latest experience to update Q(s,a) at the given timestep, a random batch of past experiences is used to update several state-action pairs.

This is of particular use for DQN and other function approximation methods since an update at a single state impacts all other states. Therefore, for each update, it is important to consider the impact on many states, not just the state last experienced.

I. Fixed Target

Another modification to the DQN algorithm described by Silver is the use of a fixed target. Recall the bellman equation (modified for function approximation).

$$Q(s,a,w) = Q(s,a,w) + \alpha(\gamma \underset{a'}{Max} Q(s',a',w) - Q(s,a,w))$$

As described earlier, it is possible for this function to diverge from the true value of Q(s,a) since W and $\underset{a'}{Max} Q(s',a',w)$ are interdependent. To avoid this risk, Silver suggests the use of a fixed Q target. That is, replace $\underset{a'}{Max} Q(s',a',w)$ in the update with an estimate of Q(s`,a`) that remains fixed for some period of time. After a certain number of iterations the target can be updated. Silver recommends using a q-value from some previous iteration as the fixed target.

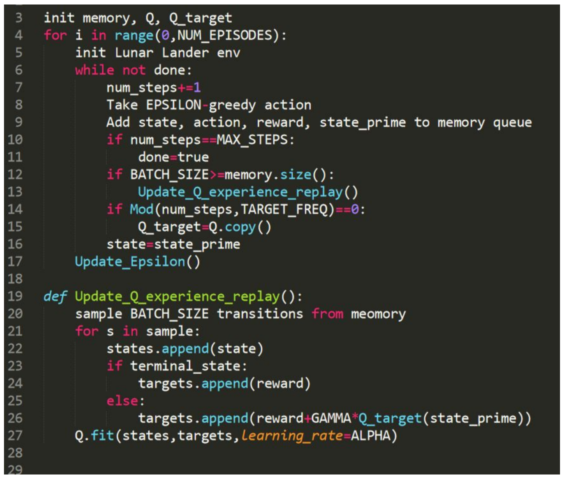
This substitution should not come as a surprise since the original, $\underset{a'}{Max} Q(s',a',w)$ rule was itself a substitution for the true value of Q(s`,a`).

IV. Implementation

Based on the above discussion, the author experimented with an implementation of the DQN algorithm using a fixed target and experience replay. The pseudocode is provided in the figure below with hyperparameters in capital letters

Due to the long run-time of this algorithm, proper understanding of each hyperparameter is critical. Their impact and selection process is described below.

Epsilon is the hyperparameter that determines the agent's tradeoff between exploration and exploitation. If epsilon is close to 1, the agent will favor exploration. However, just like in life, exploration is only needed when there is uncertainty. A lot of exploration is needed at the beginning of training whereas at the end of training, the agent can afford to exploit its previous learnings. To accomplish this, we introduce a new hyperparameter, *EpsilonDecay* which changes the value of epsilon after each episode. With this model, an initial epsilon value is supplied and after each episode it is multiplied by EpsilonDecay, which is a number between 0 and 1. Values close to 1 for EpsilonDecay will decay epsilon slowly and will enable the agent to explore the environment for more episodes. This additional exploration will build confidence in its estimates of Q(s,a), but will come at the cost of lower reward values and slower convergence.



```
init memory, Q, Q_target
for i in range(0,NUM_EPISODES):
    init Lunar Lander env
    while not done:
        num_steps+=1
        Take EPSILON-greedy action
        Add state, action, reward, state_prime to memory queue
        if num_steps==MAX_STEPS:
            done=true
        if BATCH_SIZE>=memory.size():
            Update_Q_experience_replay()
        if Mod(num_steps,TARGET_FREQ)==0:
            Q_target=Q.copy()
        state=state_prime
    Update_Epsilon()

def Update_Q_experience_replay():
    sample BATCH_SIZE transitions from meomory
    for s in sample:
        states.append(state)
        if terminal_state:
            targets.append(reward)
        else:
            targets.append(reward+GAMMA*Q_target(state_prime))
    Q.fit(states,targets,learning_rate=ALPHA)
```

Figure 1: Pseudo-code for the author's implementation of the DQN algorithm

Gamma is the rate at which future rewards are discounted. A low value of gamma will encourage an agent to collect immediate rewards at the expense of future rewards. Since the process of landing is slow, if gamma is too small, the agent may not be incentivized to learn to land and may instead find a quicker way to achieve a smaller reward (such as learning to hover indefinitely).

Alpha is the learning rate. It controls how much of a current iteration's learnings are incorporated into the approximation of Q(s,a). With alpha close to 1, the current iterations learnings are weighted as equally important as everything learned by the agent previously. This can result in non-convergence since each iteration is subject to randomness. However, a small lambda loses out on the opportunity to learn more quickly, therefore finding a balance is needed.

For experience replay there are two hyperparameters, the size of the cache, and the size of each random sample (or batch). Too small of a cache was observed to be insufficient because it did not store any successful landings or near landings.

The size of the neural network could also be considered a hyperparameter. For this implementation, two hidden layers with 64 nodes were used.

With consideration of the following factors, review of existing literature, and trial and error, the final agent was trained with the following hyperparameter values:

Alpha: 0.002, Gamma:0.99, Epsilon: 1, Epsilon Decay: 0.995, Memory Size: 100,000, Batch Size: 64, Number of training episodes: 1,500.

V. Discussion of Results

A. Trained Agent

Figure 2 shows the training process of the agent. Note that the agent satisfies the criteria for solved (average score above 200 for 100 consecutive trials) at approximately trial 900.

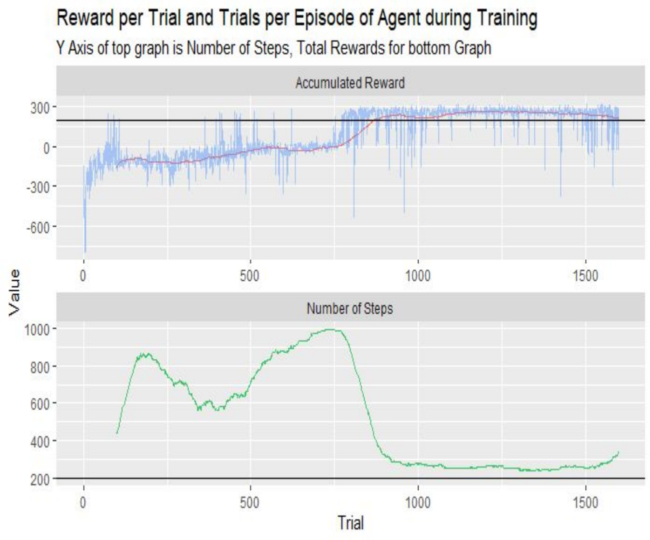


Figure 2: The top chart shows the total reward per trial during training in blue. The red line shows the 100-episode trailing average. The bottom chart shows the 100 episode trailing average number of steps per episode.

Note that at around episode 750 the agent reaches an average trials per episode of nearly 1,000: the maximum number of trials per episode. This immediately precedes the period that the agent reaches the 200-point reward threshold. After observation of the agent during this period of training, it appears that the agent avoided crashing by simply hovering above ground near the landing pad. Since action selection isn't deterministic, it is epsilon-greedy, the agent occasionally chooses actions that enable it to land (with a few crashes). The agent quickly learned that a successful landing earns more points than hovering. As a result, the number of steps per episode quickly decreased whereas the average score quickly increased.

Figure 3 shows the results of the trained agent. Note that there are still several episodes with crashes despite satisfying the criteria for solved. There are at least four causes of this behavior. (1) The criteria for "solved" is not strict enough to guarantee a higher success rate, (2) the stochasticity of the initial state of the environment, (3) network used for approximation was very large and, with the current hyperparameter values, the training time was not sufficient for a higher success rate.

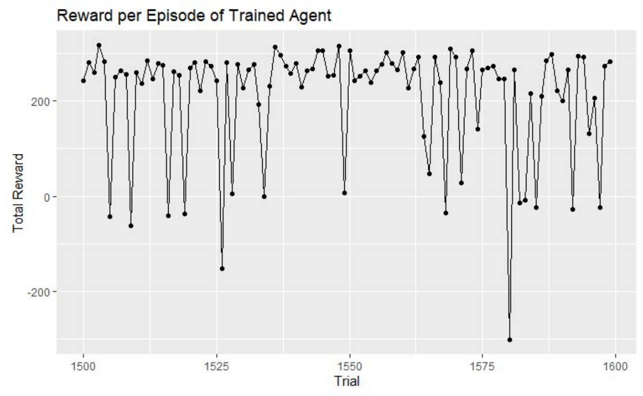


Figure 3: Reward per episode of trained agent.

B. Hyperparameter Experiments

All experiments were conducted by varying one hyperparameter at a time with respect to a default network. The default network was similar to the agent described in the above sections, however a learning rate of 0.0005 was used instead of 0.002. This was due to time limitations.

The breakthrough discussed in section 5A may not have been possible with a smaller learning rate. Figure 4 shows the agent's performance holding all hyperparameters equal except learning rate. The largest learning rates, 0.002 and 0.01 reached a 200 point average the quickest. However, the 0.01 learning rate drops below the 200 point threshold near the end of training. This is because large learning rates can overfit the observations and mistake randomness for true value. The learning rate of 0.0005 never reaches the 200 point threshold and experiences a reward spike much later than the other learning rates. This is because information is incorporated into the agent much more slowly than the other models.

Varying gamma also demonstrated interesting effects on the model, as seen in Figure 5. Gamma values of 0.99 and 1 enabled the agent to experience a quick spike in rewards per episode. Smaller values of gamma appeared to never learn how to land. To understand this phenomenon, consider two agents, one with gamma = 0.9 and another with a gamma=0.99. Assume that both agents are hovering and would require an additional 20 time steps to successfully land. If this landing results in an additional 100 reward points, the agent with a lower gamma value would only consider 12% of this reward. However, the agent with gamma=0.99 would consider 80% of the total reward in its value estimates. When these agents balance the tradeoff of landing successfully with the risk of crashing, it is more likely that the expected reward of the gamma=0.9 agent will be negative. Therefore, it should come as no surprise that the gamma=0.9 agent never learns to land successfully. In other words, it is too worried about the present in order to value the future opportunity of landing.

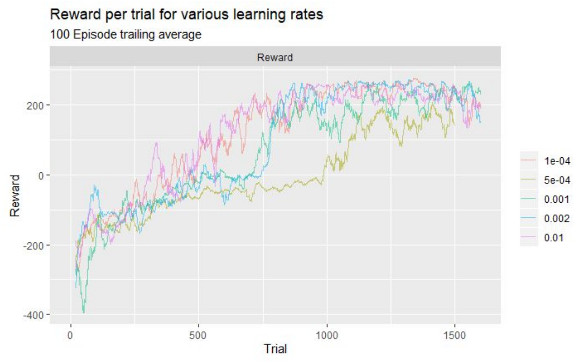


Figure 4: The 100-episode trailing average reward for agents with various learning rates.

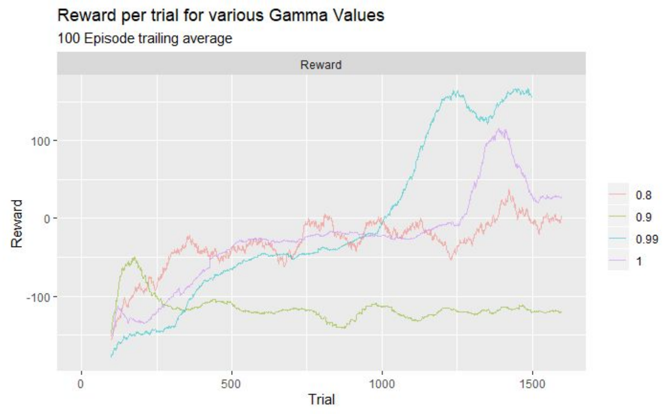


Figure 5: The 100-episode trailing average reward for agents with various gamma values.

Varying the rate at which epsilon is decayed also has significant impacts on model performance. Figure 6 demonstrates that an epsilon decay factor of 0.75 enabled the agent to reach a 200-point 100-episode average very quickly. This is because the agent switched from exploration to exploitation of the model much faster than the larger decay factors. While a decay factor of 0.995 eventually reached 200 points, hundreds of trials were wasted on exploration when they could have been spent exploiting previous learnings. This observation is highlighted by the average number of steps of each agent. With large epsilon decay, hundreds of episodes were spent reaching the maximum number of steps (1,000) whereas with a smaller epsilon decay, the agent never reached 1,000 time steps.

Although not tested, epsilon decay values smaller than 0.75 could potentially improve the model. However, they may also decrease performance by switching from exploration to exploitation too quickly. In this case, the agent's approximation is poor and the actions that it is exploiting do not have accurate estimations. Additionally, smaller epsilon decay factors could prohibit the agent from ever discovering a successful landing.

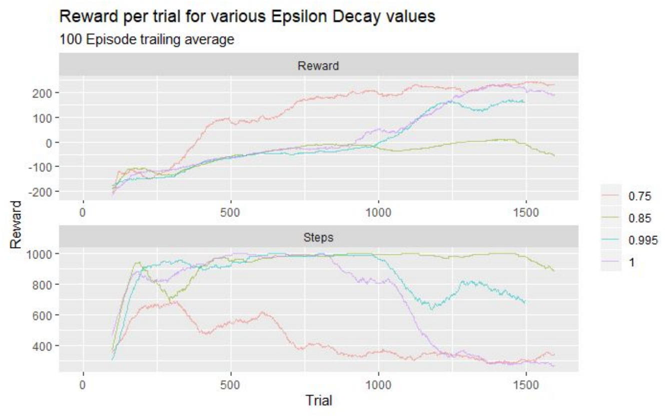


Figure 6: Agent performance for various epsilon decay factors. The top chart shows the 100-episode trailing average reward score. The bottom chart shows the 100-episode trailing average number of steps per episode.

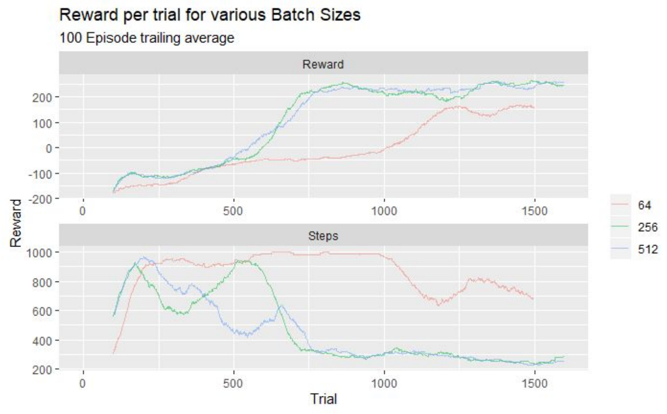


Figure 7: The 100-episode trailing average reward for agents with various batch sizes for experience replay.

Figure 7 demonstrates the impact of various batch sizes for experience replay's impact on the model. A batch size of 64 learned much more slowly than batch sizes of 256 and 512. This does not come as a surprise since a larger batch size makes more efficient use of the data by reusing past experiences. As a result, the agent spent significantly more trials hovering with a batch size of 64 compared to the larger batch sizes.

VI. Conclusion and Next Steps

This report outlines the implementation of a successful DQN algorithm for solving the Lunar Lander environment. It makes use of experience replay and a fixed target to achieve this feat. The impact of 4 hyperparameters are discussed and evaluated.

While the algorithm was successful, there are several future experiments that could improve the number of episodes required to train. Firstly, Silver discusses an improvement of experience replay, called Prioritized Experience Replay (PER) which applies priority-based sampling instead of uniform sampling from the memory cache.

Most importantly, is hyperparameter tuning. Larger batch sizes (>512), smaller epsilon decay values (<0.75) and intermediate learning rates (between 0.002 and 0.01) should all be explored. Additionally, future research should investigate interaction effects of hyperparameters. This analysis conducted all hyperparameter changes on the same default network. However, these results may not hold if multiple hyperparameters are simultaneously varied. In particular, the relationship between learning rate and epsilon decay should be explored. While this study recommends increasing learning rate and decreasing epsilon decay, the combination of these tuning recommendations may differ than their individual impacts.

VII. References

[1] Sutton R. S., & Barto A. (2018) Reinforcement learning: An introduction, Cambridge MA: The MIT Press

[2] Silver D. "RL Course by David Silver - Lecture 6: Value Function Approximation," YouTube, May 15, 2015. Available:https://www.youtube.com/watch?v=UoPei5o4fps. [Accessed: March 15, 2020].

[3] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Humanlevel control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.