Solving LunarLander-v2 using Double Deep Q-Networks with Experience Replay

Nicholas C. Park
npark62@gatech.edu

Abstract

Reinforcement learning has had success even in environments with a large states such as playing the Atari game of Video Pinball or controlling an aircraft. Many problems in real-life scenarios have an infinite number of states due to continuous state space variables. In this paper, we solve a toy environment from OpenAI gym called LunarLander-v2 to optimally navigate the moon lander onto a permitted land zone with a DQN agent. The key implementation idea is to achieve function approximation of the optimal action values using neural networks via the use of another network called a fixed target and the concept of experience replay. The following hyperparameters of the agent are experimented on with respect to the overall training speed: the network's hidden layer sizes, the decay rate of the exploration probability, the discount factor γ and the target update frequency.

1. Introduction

The LunarLander-v2 is a toy environment from OpenAI gym that embodies the control problem of optimizing the action value function of a spaceship in an environment to land in the landing pad denoted as between the two flags in the picture shown below by starting out in the middle, top of the screen.

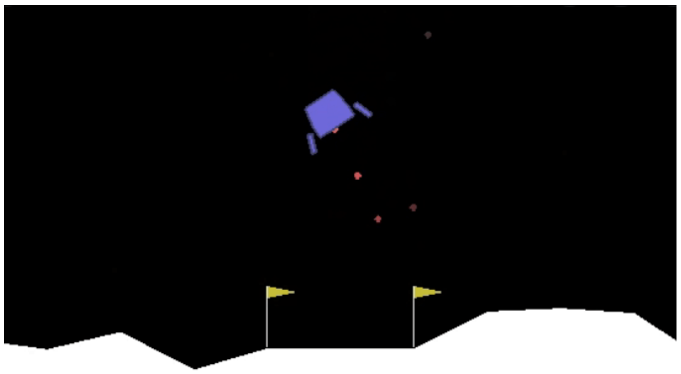

Figure 1. The Lunar Lander Environment

Space	Variables
State	$x, y, \hat{x}, \hat{y}, \theta, \hat{\theta}$,leg$_L$, leg_R
Action	Nothing, Left, Main, Right

Table 1. States and Actions of the Lunar Lander Environment

There are eight state features. The continuous state space variables are as follows: x, y denote the horizontal and vertical positions of the agent, $\hat{x}$, $\hat{y}$ denote the horizontal and vertical speeds of the spaceship, θ and $\hat{\theta}$ denote the angle and angular speed. The discrete variables leg_L, leg_R denotes whether or not the left and right leg of the ship is touching the ground. There are four actions available by the agent. The actions indexed from 0 to 3 are as follows: 0 - do nothing, 1 - fire right (or move left), 2 - fire bottom (main engine) and 3 - fire left (or move right). The reward structure is specified in the original OpenAI gym repository [3].

The control problem is solved by optimizing the trajectory of the spaceship via the function approximation of the action value function. By bringing the action value function to convergence - which maps the set of the eight state features into the values of the four discrete actions - to the optimal value function that maximizes the discounted sum of the future rewards, an appropriate policy can emerge by acting greedily with respect to this function.

2. Approach & Implementation of Double DQN

Naive Q learning is known to face stability issues due to the sequential nature of the reward-state sequence and overestimation of action values due to maximization bias. Here we introduce a Double Deep Q Network agent that adds a second network in order to update the weights in the function approximation, along with experience replay, to converge with better stability.

2.1. Q Learning

The Q Learning control learns action values based on the successive values of the state-action pair derived from a greedy target policy, while actually following an ϵ-greedy behavioral policy to pick an action. Because the behaviour and target policies are different in terms of the presence of exploration, this is an off-policy learning. Given an experience described by S_t, A_t, R_{t+1} and S_{t+1}, the corresponding Bellman update is as follows:

$$Q\left(S_t, A_t\right) \overset{+}{\leftarrow} \alpha(R_{t+1} + \gamma Q\left(S_{t+1}, A'\right) - Q(S_t, A_t)) \quad (1)$$

where A' indicates the action chosen by the target policy. In this case, A' is obtained from the greedy policy π with respect to Q, while the A_{t+1} - the actual action to be selected by the behavioral policy - follows the ϵ-greedy policy with respect to Q.

$$A' \sim \pi(S_{t+1}) = argmax_{a'} Q(S_{t+1}, a') \quad (2)$$

This produces the following Q learning update:

$$Q\left(S_t, A_t\right) \overset{+}{\leftarrow} \alpha(R_{t+1} + \gamma max_{a'} Q\left(S_{t+1}, a'\right) - Q(S_t, A_t)) \quad (3)$$

However, the Q function in this example is not tabular. The action value function involves weight parameters to incrementally update at each step of this control algorithm, which effectively updates the action values. We formulate this Q learning target update using two neural network function approximations.

2.2. Function Approximation using Neural Network

The function approximation in this implementation uses the state feature vector X(S) that consists of the 8 state variables in Table 1 to output a vector of 4 corresponding action values.

The end goal is to obtain the optimal action values that can greedily produce the ideal policy to improve the landing trajectory. As stated above, an experience is getting a reward and a new state upon an action in a state. By fitting a parametric model, each step of the agent's experience can be selected to incrementally update the model with gradient descent. In Deep RL, the parametric model used is a neural network. Instead of multiclass logistic regressions, deep neural networks are used for its flexibility to represent nonlinearities through the multi-layer architecture. It is this multi-layer structure through which an input data can be hierarchically processed to extract deep, abstract features useful for classification. These abstract features can also be useful for an agent in RL to learn to act optimally in a state.

Model applications with high-dimensional inputs such as tensors of image pixels require greater parameter capacity and typically can reach millions of neurons. However, our state space is 8 variables, and the environment is not stochastically or dimensionally complex. The max width is 512, and the max depth is 3. Also, only linear layers are used. This is unlike in Atari game playing where the screen image is fed into a convolutional neural network that are also generally larger in capacity. This is not necessary here. Each linear layer is followed by a ReLU activation in order to achieve a degree of nonlinearity from the linear transformation and for cheap gradient computation.

Algorithm 1: Target Network with Experience Replay

```
Data: $N$ = (mini batch size), $U$ = (target update
      frequency)
Initialize empty list $M$ (Replay Memory Buffer)
Initialize neural net $Q_1$ (Main Q Function)
Initialize neural net $Q_2$ (Target Q Function)
for NumSteps = 1 ... T do
    Take action $a_t$ with $\epsilon$-greedy policy
    Observe $r_{t+1}$, $s_{t+1}$
    Store experience [$s_t$, $a_t$, $r_{t+1}$, $s_{t+1}$] into $M$
    if NumSteps > N then
        Sample $N$ experiences $E$ from $M$
        $L = 0$
        for i = 1 ... N do
            $current_i = Q_1(s_t, a_t)$
            if $E_i$ is Terminal then
                $target_i = r_{t+1}$
            else
                $target_i = r_{t+1} + \gamma$ max $Q_2(s_t)$
            end
            $L = L + (current_i - target_i)^2$
        end
        $L = L / N$
        Perform gradient descent on $L$ wrt $Q_1$ weights
    end
    if NumSteps % U == 0 then
        Set $Q_2 = Q_1$
    end
end
```

2.3. Experience Replay

In a supervised learning setting, independent and identically distributed (i.i.d.) data are proven to show better convergence behaviors. In RL, each successive experience is highly correlated and thus non-i.i.d. due to the sequential nature of the data, causing divergence. This i.i.d. batch of data can be mimicked in RL by storing each of the agent's experience as an experience tuple in the following format: $<S, A, r, S'>$, and then randomly sampling the mini-batch of tuples with replacement for use during weight updates (i.e., via stochastic gradient descent). This implementation tracks a list of tuples that, when full, removes the oldest and brings in the new tuple in a FIFO manner. This sampling breaks up the correlation. This re-use of data is also useful where obtaining real-life experience is not cheap. In RL, it may also be good to use multiple updates using the same data due to the slow and incremental nature of the algorithm.

2.4. Updates: Double DQN vs. Vanilla Target Network

The weights need to be updated in the direction of reducing the loss, or the MSE, between the ideal and predicted action values using gradient descent. This loss is computed as the following:

$$L(w) = \mathbf{E}[(Q_\pi(S, A) - \hat{Q}(S, A; w))^2] \tag{4}$$

To approximate this $L(w)$, we introduce different action value functions for the weight update: Q_{main} that is updated at every step and used to pick actions in an ϵ-greedy way and Q_{target} whose output action values are compared against those of the Q_{main} for the parameter update of Q_{main}. The ideal action value is substituted as the value using the fixed target weights, Q_{target}, while the predicted action value is obtained using the main function, Q_{main}. The vanilla target network approach that approximates the true action value given the sequence S_t, A_t, r_{t+1} and S_{t+1} is obtained as the following:

$$Q_\pi(S_t, A_t) \approx r_{t+1} + \gamma max_a Q_{target}(S_{t+1}, a; w) \tag{5}$$

The only difference the double DQN has in comparison to this is that, in obtaining the Q-learning target $Q_\pi(S, A)$, the the action selected in S_{t+1} is not greedy with respect to Q_{target} but to Q_{main} [1]. Though the action is selected greedily from the main network, the target network is used to estimate the action value.

$$Q_\pi(S_t, A_t) \approx r_{t+1} + \gamma Q_{target}(S_{t+1}, argmax Q_{main}(S_{t+1})) \tag{6}$$

In both cases, the target function is implemented as a lagged version of the main function. The target function's parameters are the exact hard copy of the every-step-updated main function's, except the hard copy of the weights from the main to the target takes place every kth step. This k is set as a positive integer hyperparameter "*target_update_freq*" in this implementation so that, when set to 1, there is no distinction between the Q_{main} and Q_{target} at any step (i.e., reduces to the vanilla target approach). At every step, the loss function from the minibatch of N samples of experience tuple $<S_i, A_i, r_i, S'_i>, \forall i \in [1..N]$ is then the following:

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} (r_i + \gamma max_{a'} Q_{target}(S'_i, a') - Q_{main}(S_i, A_i))^2 \tag{7}$$

At every step, the gradient descent is taken on this computed loss with respect to the weight parameters of the Q_{main}.

3. Training and Testing

First, the double DQN approach and vanilla target network approach has shown no noticeable superiority over one another in training time. The plots generated below uses the Double DQN. Initially, the agent required more episodes to solve the environment since the effects of every hyperparameters have not been revealed enough to enable tuning for faster training. The first successful training is achieved with the following reward curve for each training episode observed. The moving average of the 100 past episodes are plotted in magenta to visualize the rate of learning with respect to the number of episodes, denoted as "SMA" for "simple moving average" over some number of episodes, e.g. 50 or 100. The run plotted in Figure 3, which took 1718 episodes

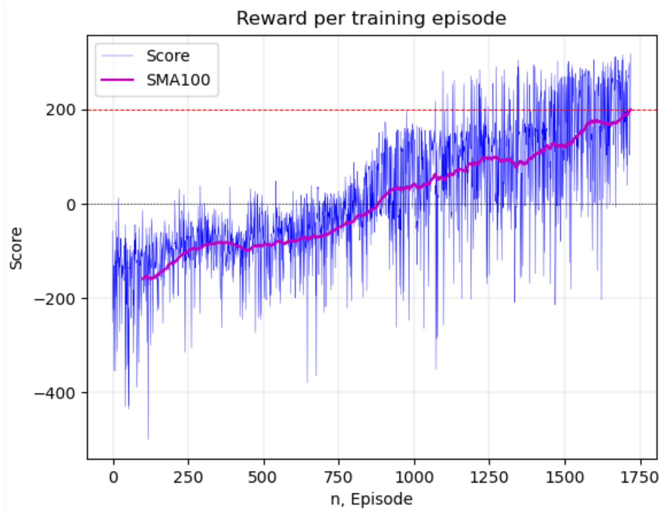

Figure 2. Training Score Plot: Pre-tuning

and 2.566.57 seconds, has the hyperparameters set as follows: ***ϵ-decay=0.999, γ=0.99, hidden layers=(64, 64), batch size=64, learning rate=1e-4, regularization=1e-6, momentum=0.99, target_update_freq=1, double_dqn=True***. The initial half of the score plot shows slow reduction in the frequency of poor termination of the episode. The performance at each episode seems to depend most on the current epsilon value. Also, the minimum ϵ value is set as 0.1. Therefore, the prolonged slacking of the 100SMA compared to the passing score 200 and the continued high variance as the episode progresses indicate too much chance of exploration for the most part of the training. Also, there appears to be an increase in training speed with the increased width of the neural network. Other hyperparameters are tuned with experimentation using grid search of the minimum time to solve the environment as well as testing the effects of one hyperparameter by searching within the unidimensional space while holding all else equal.

Upon further tuning of each hyperparameter, it is realized that for decreasing the ϵ-decay down to a value $\leq$ 0.99 resulted in a much faster solving of the environment in terms of the number of episodes required and the total training time needed. The run plotted in Figure 4, which took 271 episodes and 471.68 seconds, is achieved after tuning the ϵ-decay to 0.95 and changing the minimum ϵ to 0.05 and increasing both hidden layers of the network to (512, 512). Within around 97 episodes, the epsilon reached the minimum baseline, yet the environment was solved within the. A rapid epsilon decay seems to be helpful for this environment. By maintaining the presence of the minimum exploration possibility at 0.05, there is no need to stall the decay of this probability for a

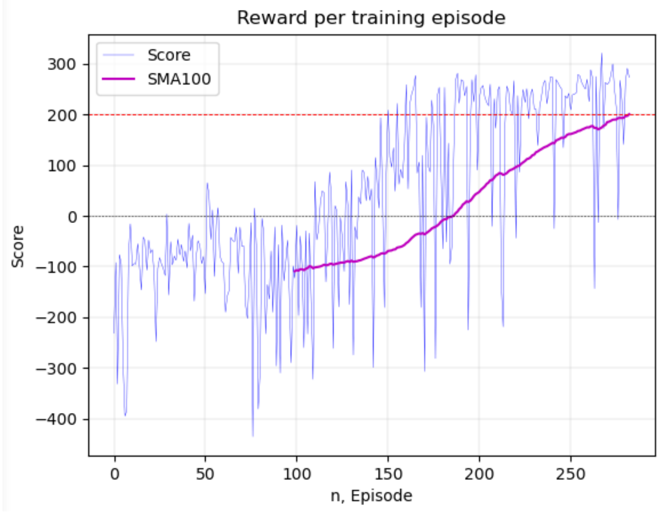

Figure 3. Training Score Plot: Post-tuning

prolonged number of episodes to achieve the environment solving action value function in this simple environment. This suggests that harmonic decay over a minimum baseline ϵ might be a good choice. Also increasing the parametric capacity of the neural network seems improve the speed of training. This emphasizes the need to learn from exploitation as well without forcing too much exploration.

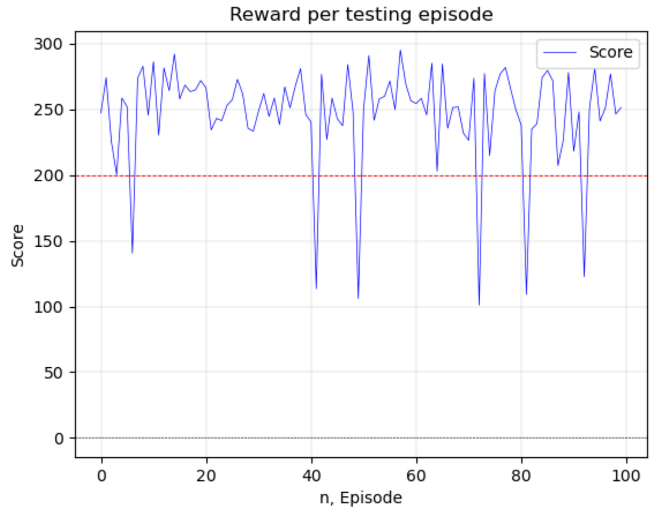

Figure 4. Testing over 100 consecutive episodes

When testing the trained agent, the exploration probability ϵ is set to 0. It turns out that the author's definition of solving the environment in this example is not sufficient to guarantee the consistent landing of the spaceship with score > 200. This definition indicates a relaxation of the convergence criterion towards the optimal action value that could be tightened through requiring a higher average score or increasing the average window.

4. Hyperparameter Analysis

Among the 20+ hyperparameters present in the DQN agent, the following section will selectively discuss the effects of ϵ-decay, hidden layer sizes of the fully connected network, γ (the discount factor) and the target network update frequency. To understand which hyperparameters contribute to an efficient training, the rate at which the agent trains with respect to the computation time is considered as well as to the number of episodes. It is noticed that training the agent can vary considerably in the total computation time depending on whether a step was taken with exploration vs. with exploitation (greedy).

4.1. Exploration vs. Exploitation: ϵ-decay

This decay parameter is the most crucial aspect of the DQN agent in properly facilitating the balance of exploration and exploitation. Although there can be different forms of decays such as a harmonic or an arithmetic decay, this geometric decay seemed to work well for the most part in ensuring sufficient exploration in the initial episodes of the runs.

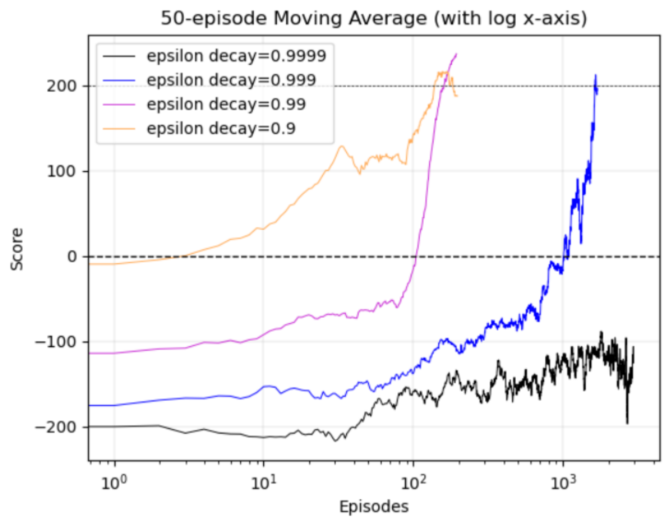

Figure 5. Varying ϵ-decay: Scores vs. # of episodes

Consider Figure 5 which shows the 50-SMA of the various ϵ-decay values. While all curves suggest an upward trend as the episode progresses, too much exploration is shown to result in the failure to achieve successful training within 3,000 episodes. The plot encourages a rapid decay of ϵ from the fact that ϵ-decay = 0.9 which reaches the minimum $\epsilon = 0.1$ in just 22 episodes trains faster than ϵ-decay = 0.99. This is likely due to the minimum ϵ that guarantees a minimum amount of exploration during training.

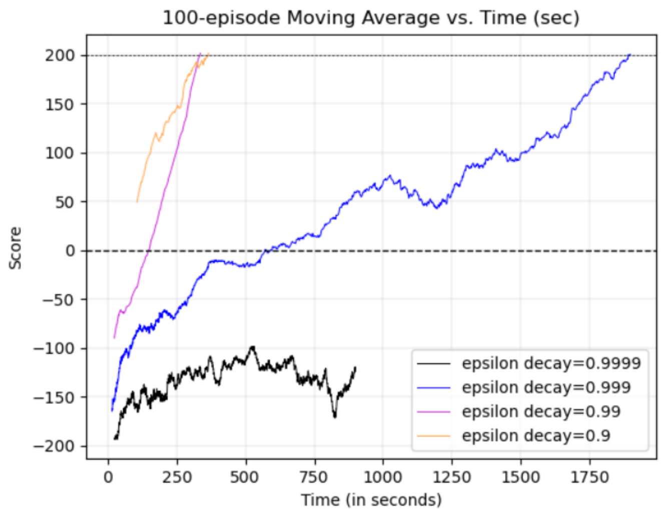

Figure 6. Varying ϵ-decay: Scores vs. time (seconds)

Consider the plot of ϵ-decay = 0.999 vs 0.9999 in Figure 6. The blue curve (ϵ-decay = 0.999) has taken 2,127 episodes, and the black curve (ϵ-decay = 0.9999) has taken 3,000 episodes (i.e., failed to solve). The difference in the time taken highlights the difference in computation costs between exploration and exploitation. Even after 3,000 episodes, the ϵ remains at $0.9999^{3000} = 0.7408$, suggesting that most of the episodes is spent exploring. This indicates that exploration, which involves a random action selection, is much cheaper than exploitation which involves a forward pass of the current state in Q_{main} and greedy selection.

4.2. Neural Network Architecture: Hidden Layer Sizes

The optimal parametric capacity of the neural architecture is searched with respect to training time holding other hyperparameters governing regularization and learning rates equal. The environment is not particularly complex with regards to stochasticity and dimensionality compared to image object detection or NLP classification in the sense that we feed 8 uni-dimensional variables to simply output a vector of 4 action values in a 2D environment. Hecht-Nielsen and Ismailov posit that any continuous or discontinous multivariate function can be implemented using a certain kind of three layer neural network. [2]. For the control of this toy example, the deep neural network model to use can be relatively trivial in width with at most 512 nodes and only involve at most 3 (and at least 2) fully connected layers for an appropriate function approximation.

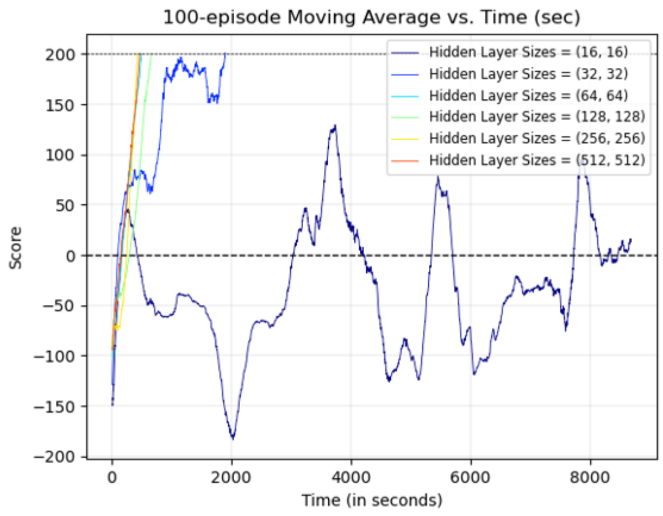

Figure 7. Varying the Network Width: Scores vs. time (in seconds)

The architecture can include the depth of the network or adding different activation functions or having ascending or descending depth pattern (e.g., 32→128 vs. 128→32). But the experiments here focus on the sheer parametric capacity by keeping depth = 2 and identical widths. Figure 7 shows that networks with at least 128 nodes per layer will significantly outperform than those with less nodes per layer in terms of the number of episodes and training time taken. Beyond 128, with every doubling of the parameters per layer there appears to be a marginal improvement as well. Also, having too few parameters shows difficulty in convergence according to the purple plot (16, 16).

This does not conclude that increasing the width of the neural net is better in general since the batch size, learning rate and regularization factors are kept the same. Reducing the batch size and learning rate can help address exploding gradients for larger linear layers. This experiment result however does indicate a general benefit of exploiting parallelism which can help solve the environment in less amount of time given efficient backpropagation via the GPU in the weight updates to converge faster to the optimal action value function.

4.3. The Discount Factor: $\gamma \in [0, 1]$

The gamma value of the variants of the Bellman update controls how much to discount future values towards the current value of the state-action pair. When $\gamma = 0$, the focus becomes only on the immediate reward of the action, while $\gamma = 1$ causes the algorithm to over-value the future rewards resulting in infinite runs or divergence. It appears there is a correct region of this "far-sightedness" to use depending on the environment, because it effectively must appropriately represent the degree of uncertainty future rewards can be in this lunar lander environment.

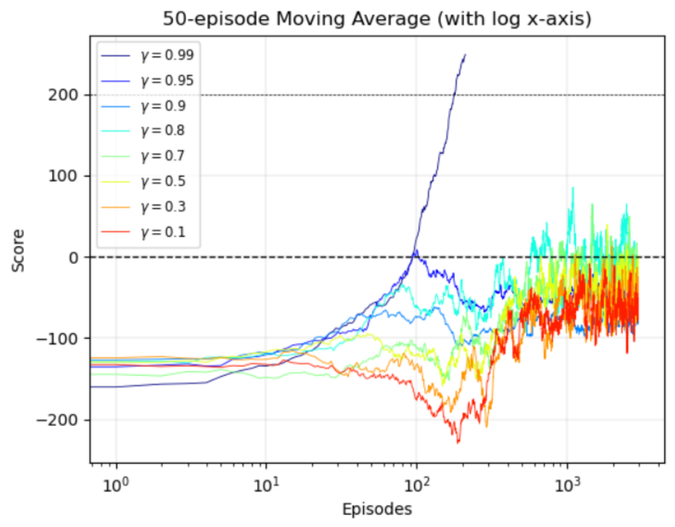

Figure 8. Varying the discount factor γ: Scores vs. # of eps.

Surprisingly, the best representation of uncertainty towards the future rewards in this model is when $\gamma = 0.99$. We can see from Figure 8 that any values of the discount factor less than or equal to 0.95 results in a failure to solve the problem. We can also see that values less than 0.9 even causes divergence in terms of the episodic rewards indicating a failure to learn the environment, which hints the necessity of learning with proper γ.

4.4. Target Network Update Frequency

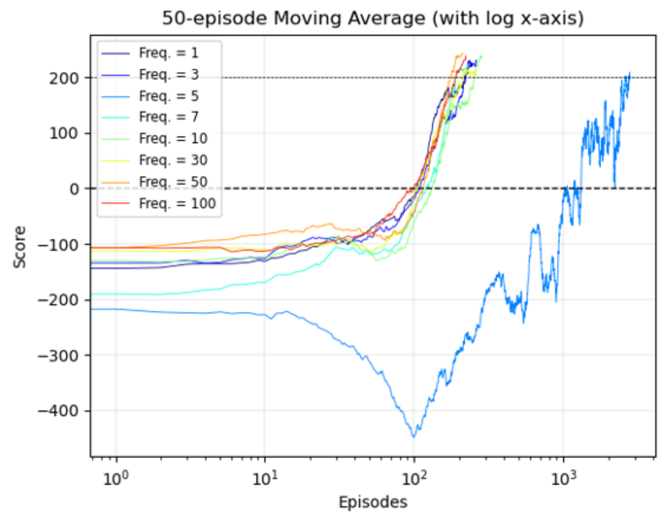

Figure 9. Varying the target update frequency: Scores vs. episodes

Figure 9 shows the hard copy can take place at a frequency over a widely ranging positive integers. The score plots during training is very similar to one another. However, freq.=5 consistently results in an initial dip in the score before the eventual success and suggests selecting the frequency based on experimentation.

5. Challenges & Pitfalls

Before discovering that the ϵ-decay was the key to solving the environment, the large number of hyperparameters involved in this DQN agent was too overwhelming to troubleshoot the ineffective training. It was important to start testing with a simple version of the DQN agent with less hyperparameters involved to observe the gradual improvement. Furthermore, even with just two hidden layers of the fully connected neural network, with maximum number of nodes to $2^9 = 512$, there still were 81 combinations of numbers of nodes to test with respect to the time required to solve the problem when the width = 2^n, $n \in [1..9]$, The variance in performance in the re-run of the same architecture made it harder to compare and contrast the effects of a given architecture factor such as width or ascending/descending numbers of nodes. Increasing the capacity the DQN agent to allow three hidden layers in the network, required a re-tuning of all other hyperparameters in order to find a competent candidate.

6. Areas of Improvement

If more time was given, the agent can be trained, tuned and tested on a privately modified version of this toy environment. We can introduce more complexity or stochasticity to the environment such as lunar wind or uneven terrain. A controlled experiment to reveal the various interdependencies of the hyperparameters (e.g., between regularization and batch size) could be helpful in developing an intuitive understanding of the training improvement. There were ambiguous assumptions in the adjustment of the learning rate with respect to the batch size of tuples per step.

Also, the agent can adopt qualitative distinctions in the training methodology such as decaying the epsilon harmonically or arithmetically instead of geometrically. The agent can also introduce bias in the way that samples are selected from the replay buffer to neglect more of the initial exploration. Furthermore, we can try action picking methods other than ϵ-greedy, for instance, via a softmax policy. The possibility of exploration is still always on the table throughout the episodes due to the presence of non-zero probability in the non-argmax indices. The temperature of the softmax can be increased to eventually selecting actions greedily in the limit. Finally, there is room for algorithmic improvement such as to adopt Polyak averaging (soft updates) when updating the target network, instead of a hard copy.

References

[1] Hado van Hasselt. Deep reinforcement learning with double q-learning. 2

[2] Vugar Ismailov. A three layer neural network can represent any discontinuousmultivariate function. 4

[3] Oleg Klimov. gym/gym/envs/box2d/lunar$_lander.py$. 1

$ *git log -1 –format="%H"*
>> 134095796fa9721e62e85f1a130b1826fdfb775c

Table 2. Latest Git Commit Hash