CS 7642 - Project 2 - Lunar Lander

[*]Commit hash: abcdefghi

1st Jeremy Martinez
Dept. of Computer Science
Georgia Institute of Technology
Seattle, WA
jeremy.martinez@gatech.edu

***Abstract*—This document covers a deep Q-network (DQN) implementation to solve OpenAI's Lunar Lander[2] environment. This algorithm was modeled after A DQN algorithm found in Minh et al.[1]. A Q-learning approach is supplemented with a SGD neural network. We scrutinize the performance of DQN's on such a problem space with hyperparameter tuning to find the most optimal set of parameters. Finally, we discuss what these parameter values mean and the effect they have on the agents ability to learn the MDP.**

I. Introduction

The Lunar Lander in OpenAI's gym environment poses a new problem for Q-learning that we have yet to encounter. In homework 4, we leveraged Q-learning to generate an approximate value function to behave in the taxi environment. A crucial difference between Lunar Lander and Taxi is the state space. Taxi's state space was discrete, which allows us to select an ϵ-greedy action with relatively low computation requirements. Lunar Lander's state space is continuous, which poses a problem with this approach. In order to address this issue, we combine Q-learning value function approximation with a neural network. The deep Q-network (DQN) algorithm was developed by DeepMind[1]. We follow their implementation here in solving the Lunar Lander problem. The Lunar Lander is considered solved when the agent achieves a score of 200+ over 100 episodes.

II. Q-learning

Q-learning is at the heart of this off-policy TD learning algorithm. This is considered an off-policy approach because it is evaluating/improving a policy that differs from that used to generate and simulate training experiences. On-policy algorithms learn the value of the policy executed by the agent (OpenAI's env being the agent in this case). We implemented on-policy learning using SARSA in HW3, which constitutes a short digression.

A. On-policy learning

In HW3, we used on-policy learning in the form of the SARSA algorithm. This stands for state, action, reward, next_state, next_action (S,A,R,S',A'). SARSA selects the next action according to the optimal policy (optimal policy so far) and updates it's Q-values accordingly. This works well for the Frozen Lake problem for two reasons. First, the frozen lake state space is relatively low. Second, the state space and action space are discrete. These two details are important to implement SARSA because of how computationally expensive it is to generate S' and A' from an optimal policy.

B. Off-policy learning

In HW4, we transitioned from on-policy to off-policy learning. Our Q-learning solution to OpenAI's Taxi environment differs from our SARSA in the way it updates it's Q-values, which is the distinction between on-policy and off-policy learning.

Our agent in HW4 estimates rewards for the next-step action from the current Q-values by applying argmax across actions in next state. SARSA does not estimate this argmax, and instead updates it's policy off of epsilon-greedy next action. In this project, we follow an off-policy Q-learning approach (augmented by a neural network, which we will cover later).

III. Off-Policy Methods with Approximation (Sutton & Barto)

Off-policy learning seeks to learn a value function for a *target policy* π, given data due to a different *behavior policy* *b*. We seek to learn action values $\hat{\text{Q}} = \text{Q}_{\pi}$. π acts as the greedy policy with respect to Q, and the target action value $\hat{\text{Q}}$ is learned incrementally, occasionally reset to the greedy policy. This occasional resetting of the policy is to ensure that the semi-gradient methods are guaranteed to converge. This is needed since the updates in the off-policy approach is not according to an on-policy distribution.

A. Semi-gradient Methods

Applying an off-policy method with function approximation, similar to that which we used in HW4, takes place in the Q-value update. Instead of updating an action-value array, we apply the update to a weight vector.

$$target\_w_{\text{t+1}} = target\_w_{\text{t}} + \alpha * (\hat{Q}_{\text{t}}(s_{\text{t+1},\ \theta^{-}}) - Q_{\text{t}}(s_{\text{t},\ \theta})) \quad (1)$$

B. Stochastic Gradient Descent

This weight update applies stochastic gradient descent (SGD) with a step-size parameter α (shown in equation 1). We then apply this weight update and adjusted Q-values to our Sequential model from Tensorflow. SGD applies incrementally small updates to the weight vector in an attempt to subtly

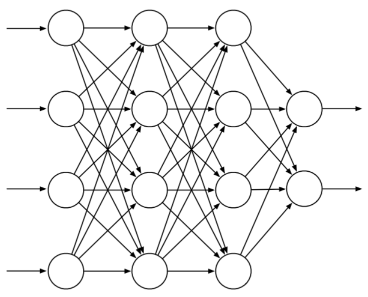

Fig. 1. A generic feedforward ANN with four input units, two output units, and two hidden layers

approach optima. This could potentially result in converging to local optima, which is why we reset out target policy to the greedy policy Q every C steps.

The Sequential model from Tensorflow utilizes a neural network to calculate the optimal weights across our continuous state space. The neural network utilizes an input layer the size of the state space, two hidden layers (the size of which, we study as a hyperparameter), and an output layer of the action space.

The objective of this neural network is to approximate the function to minimize mean square error (mse) on the value function. The *Mean Squared Value Error*, denoted $\overline{VE}$, can be seen below in equation 2:

$$\overline{VE}(w) \doteq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s,w)] \qquad (2)$$

C. Nonlinear Function Approximation

Expanding for a moment on the neural network leveraged from Tensorflow, we see a generic artificial neural network depicted in figure 1. The neural network we use if also a feedforward network, meaning no unit's output can influence it's input. As mentioned before, the ANN used here consists of four layers, one input, two hidden, one output. These layers dimensions are 8, 64, 64, 4, respectively. The input and output layers of this ANN are fixed to match that of the state space and action space (input: state, output: action to take). This follows intuition if we treat the NN as a black box. We put a current state in, and expect to get a prediction (action to take) out.

Where we have some ability to optimize performance comes with the number of hidden layers we use, the size of the hidden layers, the optimizer function used, and the learning rate of this optimizer. While it would be valuable to measure our agents performance with hyperparameter tuning each of these, due to time and resource constraints, we explore only the hidden layer dimension as a hyperparameter later in this paper.

IV. Deep Q-Network (DQN)

In algorithm 1 we see the full DQN algorithm, as implemented in project 2. This algorithm was implemented according to the DQN developed by DeepMind[1].

Algorithm 1: deep Q-learning with experience replay

```
Init replay memory D to capacity N;
Init action-value function Q with random weights $\theta$;
Init target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for episode = 1, 2000 do
    Init state;
    for t = 1, 500 do
        With probability $\epsilon$ select random action $a_t$
        Otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$;
        Execute action $a_t$, observe $r_t$ and next state $s_{t+1}$;
        Store transition ($s_t$, $a_t$, $r_t$, $s_{t+1}$, is_terminal) in D
        Sample random minibatch of transitions from D
        if Every C steps then
            for ($s_t$, $a_t$, $r_t$, $s_{t+1}$, is_terminal) in minibatch
            do
                if is_terminal then
                    $y_j = r_t$
                else
                    $y_j = r_t$ + gamma * $\max_{a'} \hat{Q}(s_{j+1}, a'; \theta)$
                end
            end
            Perform gradient descent step;
            Reset $\hat{Q} = Q$;
        else
            do nothing
        end
    end
end
```

This algorithm has parts that look and feel much like the Q-learning we implemented in HW4. However, Q-learning has a tendency to become unstable when a nonline function approximator (such as neural network) is used to represent the action-value function. DeepMind addressed this with two novel approaches, which we have recreated in our own DQN. The first is to reduce correlations present in a sequence of observations by accumulating state/action/reward pairs over N number of steps across M number of epochs (capped at some arbitrary value) and storing them in a *replay memory*. Then, uniformly selecting a random subset from the *replay memory*, we apply a Q-learning off-policy weight update. Second, to avoid a tendency in the target-action-value function to diverge, we periodically reset it to another action-value function *Q*.

V. Lunar Lander

The Lunar Lander, developed by OpenAI Gym, is a RL environment in which the agent attempts to land a space craft on the moon. The state space includes 6 continuous variables, and two discrete: *(x, y, x-velocity, y-velocity, angle, angle-velocity, left-leg, right-leg)*.

The first two variables, *x* and *y* are the horizontal and vertical coordinates of the space craft in the 2-dimensional map. The next two, *x-velocity* and *y-velocity* show the velocity with respect to the first two variables. *angle* and *angle-velocity* describe the upright angle of the space craft, and the velocity

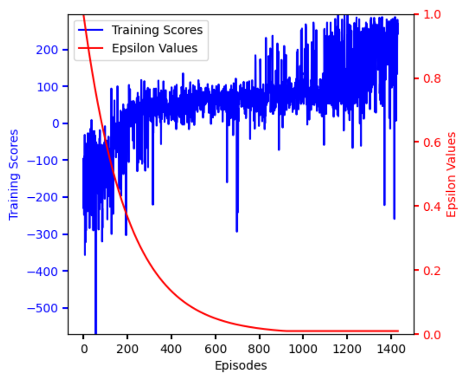

Fig. 2. Training scores and epsilon values mapped against episodes

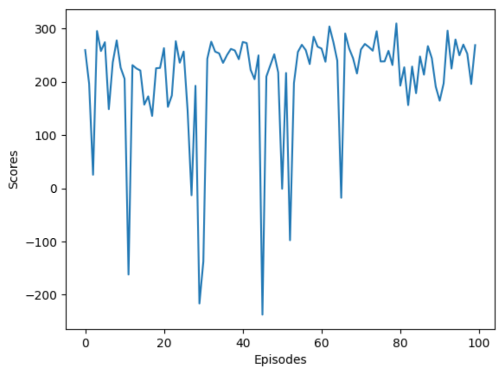

Fig. 3. Testing trained-agent on 100 episodes

at which it is turning (positive value denotes clock-wise, negative denotes counter clock-wise). The last two discrete variables, *left-leg* and *right-leg* are boolean values indicating if the left/right leg are touching the ground or not.

The reward structure of this environment is as follows. The space craft receives between 100..140 points for moving from the top of the screen to the landing pad (ending with zero speed). If it moves away from the landing pad, it loses equivalent reward. The episode terminates when the lander either crashes or comes to a rest, receiving an additional -100 or +100 points. Each leg contact is +10 points. Finally, firing the main engine is -0.3 points each frame.

The action space consists of four discrete actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

VI. Training/Testing the Agent

The agent took 145 minutes to train. The agent ultimately passed the environment with an average score of 200 over 100 consecutive iterations. The agent converged to this in 1,431 episodes (as seen in figure 2). However, with different seeds, I did see this agent converge under 1,200 episodes as well. Ideally, with unlimited computer processing, I would like to run this across 10 cores with 10 different seeds, average all of the processes scores, and then plot that as a representative of the DQN's performance. However, given this limitation, this plot will suffice.

The agent tested very well (as seen in figure 3) with an average score of **207.105** and an average episode duration of **2.619 seconds**. The default parameters used in training/testing can be seen in table VI

VII. The Effect of Epsilon-Decay on Policy Convergence

Epsilon decay was the most promising looking hyperparameter among the three. I expected this to have such a strong influence over the learner because of it's influence in HW4. I was able to reduce the required number of episodes to train in HW4 to one fifth without ϵ-decay at all. I only used three values in testing different ϵ-decay values: *0.003, 0.005, 0.007.*

TABLE I
List of parameters for Agent training/testing

γ	0.99
α	0.003
update frequency	4
sample minibatch size	64
max step size	500
ANN hidden layer dimension	64
ANN learning rate	0.0005
Tensorflow optimizer	Adam
Loss function	Mean Squared Error
replay memory size	1000000
ϵ-decay	0.005
ϵ	1

Figure 4 shows the value that epsilon takes over the course of 1,000 episodes. In the DQN, a minimum 0.01 was enforced for the epsilon value.

This hyperparameter controls the exploration/exploitation of the agent in training. When selecting ϵ-greedy actions, the lower ϵ-decay values will encourage exploration for longer. This is important, and proved to be most optimal among the three (as seen in figure **??**). The reason higher values for ϵ-decay yield slow convergence rates is because the agent begins to take ϵ-greedy actions too often without exploring enough of the state space. Whereas, on the contrary, lower ϵ-decay values yield a higher ϵ in later episodes encouraging the agent to learn more about the state space. This knowledge in turn allows them to select more optimal greedy actions when the ϵ eventually reaches the minimum.

The ϵ value reaches the minimum of 0.01 for the three ϵ-decay values represented in figure **??** after *655 episodes* for a decay of 0.007, *918 episodes* for 0.005 and *1,532 episodes* for 0.003. It is also worth noting that the ϵ-decay of 0.003 solved the Lunar Lander problem in 972 episodes and an ϵ-decay value of 0.007 solved it in a record speed of 908 episodes in hyperparameter testing.

VIII. The Effect of Gamma on Policy Convergence

Refer back to algorithm 1 when considering gamma's impact as a hyperparameter on the DQN. The gamma discount rate is applied to the Q-learning step in the sampling of the

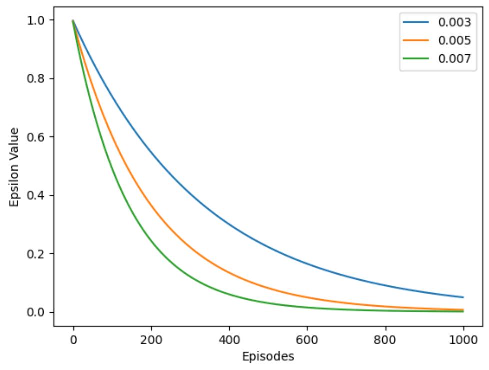

Fig. 4. Rate at which epislon decays over three values for ϵ

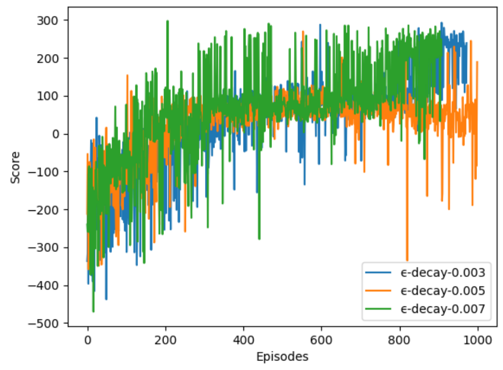

Fig. 5. Agent training convergence with tuning ϵ-decay as a hyperparameter

minibatch. This parameter, similar to the ϵ-decay is not new to this project. We once again see the impact this parameter has over the Q-learning by analyzing the agents ability to train with three different values at 1,000 episodes: *0.79, 0.89, 0.99*

Taking the last 100 episodes from this subset and averaging them, we get *-5.97, -30.049, 44.412*, respectively. This is hard to analyze, and perhaps would benefit from multiple iterations and then average over those iterations (given unlimited processing power). However, it does highlight that a gamma value of 0.99 is most optimal. Gamma discounts the argmax action from our target-action-value function. A value of 1 suggests that every subsequent action is *just as* important as the current state-action value. A value *close to* 1 being most optimal makes sense in this problem because the state space is continuous and the number of steps (500 steps max) is very large. With a gamma value of 0.99, this would mean that the 100 reward at the end of the epoch holds a value of 0.657:

$$y_0 = Reward_0 + \gamma^{500} * Reward_{500} = 0.657 \qquad (3)$$

Manipulating this hyperparameter forced me to contemplate it's influence on the q-values in relation to the state space and overall step size. I hypothesize that, given an agent with a smaller step size and state space, a lower gamma will be

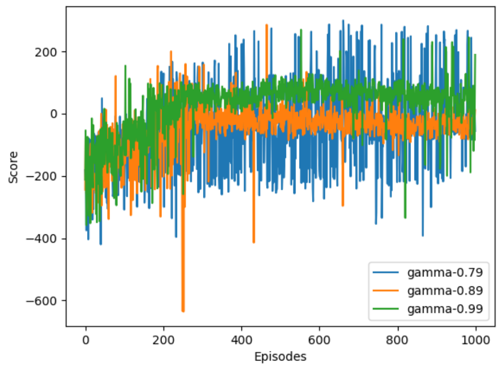

Fig. 6. Agent training convergence with tuning gamma as a hyperparameter

optimal. A fun experiment to run on gamma would also be to chunk episodes into fifths and increasing the gamma value. That way, the discount factor is higher, but only applies to the next 100 steps, then it drops to 0. This would essentially be translated to the agent as to tell it, that anything that happens in the top 20 percent of the state space (y-value) is unaffected by anything actions in the next 20 percent, and so on.

IX. The Effect of CNN Hidden Layer Size on Policy Convergence

The hidden layer of the neural network seemed to have performed similarly with a dimension of 16, 32, and 64 (as seen in figure 6). The average of the final 100 episodes across the three values are *47.118, 80.968, 44.411* with a stand deviation of *26.094, 45.081, 71.899*, respectively. This suggests that at 1,000 episodes of training, an ANN with two hidden layers of 32 dimensions yields, on average, the highest convergence. However, it also suggests that a lower dimensionality yields a more stable/steady trend toward convergence. It would be my theory that, given the stability of a 16 dimension hidden layer, this could surpass 32 if extended over 1,500 - 2,000 episodes.

A high dimensionality is known to help avoid overfitting in non-linear function approximation, and so it would be pertinent to bear this in mind when evaluating the trade-offs of a lower dimensionalitys stability/training performance score to a higher dimensionalitys slightly lower training score. In order to compare their tendency to overfit, a similar comparison should be evaluated with the agent's tests scores. We are omitting this here due to computation and time constraints.

X. Pitfalls & Problems

I encountered constraints in terms of computation power/speed. To train the agent took several hours. A couple times, I had to run nightly builds, and then check the results the next morning. This posed a challenge since I was not able to receive immediate feedback on optimizations I would make to the codebase. I was unable to fully vectorize my codebase, too. There still exists two for loops that could potentially be vectorized using numpy, which would have a dramatic increase in performance. The two for loops exist in the sample

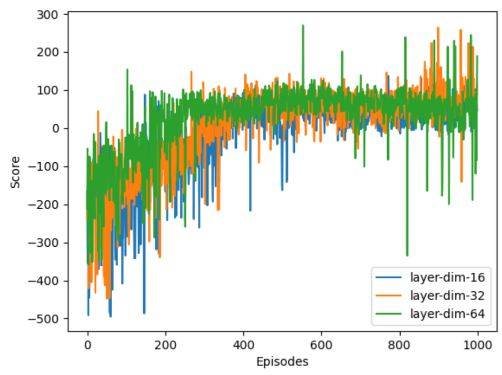

Fig. 7. Agent training convergence with tuning dimension of hidden layer dimension in ANN as a hyperparameter

minibatch computation in regard to the Q-learning step with the target-action-value function and the weight update.

I tested this algorithm, in the early stages of development, against the CartPole-v0 environment (also provided by OpenAI's gym). This was ideal to develop against in the early stages because it alleviated the issues of computation power and speed. I was able to receive quicker feedback that my algorithm was on the right track with this environment. Once I solved CartPole-v0, I knew my DQN was ready to move on to the more complicated problem of LunarLander-v2.

XI. Further Research

Given more time, I would look into parallelizing this code more efficiently. Once this code is parallelized efficiently, I would select more hyperparameters and test all permutations of these against one another. I am particularly interested in how tuning parameters related to the ANN affect the agents overall ability to converge, and the speed at which the agent converges.

One final note, as I wrap up my hyperparameter testing. I am noticing some inconsistent performance across different iterations of agent training. I am reusing the same seed for numpy and the gym env on every iteration. So this variability seems to suggest that CPU availability or processing power may play a factor in getting the agent to converge. In order to quiet down standard deviation between consecutive invocations, I would like to run all of these experiments multiple times (using something like GNU parallel) and average the results across all.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, et al. (2015). Human-level control through deep reinforcement learning. Nature, p529-533. doi: 10.1038/nature14236

[2] https://gym.openai.com/envs/LunarLander-v2/

[3] Sutton, R. S., Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[4] Hado Van Hasselt, et al. Deep Reinforcement Learning and the Deadly Triad. 2018.