Project 2 Report, CS 7642 Reinforcement Learning and Decision making

Farhad Batmanghelich (fbatmang3@gatech.edu)

Git hash: e0f4b5529d27b1837c25d318ab3df148d535c8d4

***Abstract*— In this report, solution to the LunarLander-v2 environment from OpenAI Gym by Deep Q-Network (DQN) algorithm is discussed. Along with fundamentals of value function approximation in reinforcement learning, impact of different hyperparameters in the DQN model is experimented.**

Keywords*—*value function approximation, reinforcement learning

I. Introduction

Tabular reinforcement learning updates, e.g. Q-learning, are highly inefficient and/or impractical if the state space is continuous or high-dimensional. In such state spaces, to develop a policy, it is wise to learn a function that approximates the value of each state by taking the state vector as input and returning value for each possible action. Learning a function from labeled examples is called *supervised learning* and we should find a way to incorporate this paradigm into reinforcement learning context for value-function approximation. This is usually done by using the target of a reinforcement learning update $s \mapsto u$, where s is the state being updated and u is the update target, as examples for training the value-function being approximated. For example, G_t(return) in Monte Carlo or $R_{t+1} + \gamma v(S_{t+1})$ in TD(0) are target of the update and thus, along with their input state, can be used as examples to train the value-function. A major difference between supervised learning and value-function approximation in the context of reinforcement learning is that in the former, examples are fully captured and do not change, however in value-function approximation, experience becomes richer as the agent explores the environment further.

II. Concepts

A. Function Approximation

In the supervised learning paradigm, to find the best function that approximates labels given input vector, first a decision must be made on the form of the function. This would define the function space from which the best function would be selected. The notion of "best" is embodied as a cost function which returns a measure of how far our predictions are from the gold-standard, i.e. true labels in the training dataset. What differentiates functions in a function space, are the weights (coefficients) of the basis functions, therefore our task is to find the vector of weights that minimizes the cost function. This minimization problem is easier when we assume a simple function form, e.g. linear but it gets harder to solve in complex function forms, e.g. a neural network. In easier cases, we can set the derivative of the cost function w.r.t the weight vector to zero and find the roots analytically but in complex cases (neural network) we cannot do this analytically and thus we should use an iterative method called *gradient descent*. In this method, weight vector is incrementally moved towards the negative of the gradient of cost function until convergence, i.e. insignificant change in weight vector by further iteration. For example, if we have m examples of n-vector ($\boldsymbol{x}_i \epsilon R^n$) inputs that are labeled ($y_i \epsilon R$), a mean squared error cost function would be:

$$J(\boldsymbol{w}) = \sum_{i=1}^{m} (y_i - f(x_i; \boldsymbol{w}))^2$$

Optimum **w** would be the roots of $\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = [\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_1}}, \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_2}}, \ldots, \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_n}}]^T = 0$. This is easier for simple models but becomes infeasible for complex models such as neural networks. For complex models, as mentioned above, gradient descent methods move the **w** vector towards the negative of $\frac{\partial J(w)}{\partial w}$ with a particular learning rate, α, with the hope that it reaches to a global or local optimum **w**. The iteration continues until convergence. For example, for linear models where **w** has the same dimension as $\boldsymbol{x}_i$:

$$[\boldsymbol{w}_{t+1,1}, \boldsymbol{w}_{t+1,2}, \dots, \boldsymbol{w}_{t+1,n}]^T$$
$$= [\boldsymbol{w}_{t,1}, \boldsymbol{w}_{t,2}, \dots, \boldsymbol{w}_{t,n}]^T - \alpha[\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_1}}, \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_2}}, \dots, \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w_n}}]^T$$

And in general:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \nabla_w J(\boldsymbol{w})$$

B. *Function Approximation in the Context of Reinforcement Learning*

As previously mentioned, in the reinforcement learning context, for non-trivial state spaces, one should approximate value-function similar to supervised learning. Examples required for training the value-function, come from reinforcement learning updates, thus the cost function for a single state transition can be written as:

$$J(\boldsymbol{w}) = [U_t - \hat{v}(S_t, \boldsymbol{w}_t)]^2$$

$\hat{v}(S_t, \boldsymbol{w}_t)$ is the approximated value of S_t computed by the function and U_t is ideally the true value of state S_t but in the absence of such measure, it can possibly be a bootstrapping target using the previously mentioned function, $\hat{v}$. For example in TD(0) update, $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w_t})$ or for TD(0), Q-learning, which is a TD(0) algorithm to learning Q values, $U_t = R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, \boldsymbol{w_t})$ or for Monte Carlo update $U_t = G_t$.

In Sutton's book [1], it's mentioned that the goal of learning a value-function is to find a better policy and this does not necessarily mean that we have to minimize the cost function. However, since there is not yet clear what other alternative objective we could use, we can focus on minimizing above cost function for value-function approximation in the context of reinforcement learning. Optimal value-function has a **w** vector that minimizes $J(\boldsymbol{w})$, we can use gradient descent method to find the optimal **w** and since we are moving **w** at each online example, this method is called *stochastic gradient descent*.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla_w J(\boldsymbol{w})$$
$$= \boldsymbol{w_t} + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla_w \hat{v}(S_t, \boldsymbol{w}_t)$$

Another note here is that since U_t is a bootstrapping estimate of the true value written in terms of $\hat{v}$, it is a biased estimate of the true value meaning that its expectation is not necessarily equal to the true value. Therefore, if bootstrapping targets are used in the above gradient descent update, we are not truly looking at an instance of gradient descent. We assume the bootstrapping target to be independent of **w** vector at time t and thus its derivative w.r.t $\mathbf{w_t}$ is 0. Therefore, a more suitable name for this optimization algorithm is *semi-gradient methods*. Although semi-gradient methods do not guarantee to converge as reliably as gradient descent methods, they have many advantages namely, reliable and fast convergence in important function forms and also they enable learning to be online.

From now on, whenever I mention value-function I mean the action-value function, Q.

C. *Deep Q-Network (DQN)*

The algorithm I used to solve the LunarLander problem is called DQN by Mnih et al [2]. At a high level, this paper uses neural network as the value-function form in TD(0) Q-learning updates (Figure1). Then stochastic gradient descent is used to optimize the cost function, where cost function is the squared difference between TD(0) Q-learning target bootstrapped from the value-function and the value predicted by the value-function. However, simply implementing above would cause the value-function to diverge because of the correlations between consecutive samples and non-stationarity of targets. Mnih et al, addressed these instabilities by including a biologically inspired mechanism called *experience replay* which removes correlation between consecutive observations by storing agent's experiences and randomly selecting from the pool of experiences for each value-function update. The other mechanism for imparting more stability into the DQN algorithm is to use two neural networks called model and target model respectively. Target model is used to generate the TD(0) Q-learning updates and model is used for value prediction. The reason that this modification is helping stability is that if the model used to generate Q-learning update is the same as the model for value-prediction, it is highly probable that an update that increases $Q(S_t, a_t)$ also increases $Q(S_t, a)$ for all the actions which can result in instabilities and divergence. The way DQN is implemented in this report is captured in Algorithm 1:

Algorithm 1: DQN implementation in this report

```
Input: LunarLander-v2 Gym environment
Output: a tensorflow.keras neural network model that
given a state returns action-value for each of the 4
possible actions
Initialize replay buffer = []
Initialize tf.keras action-value model $Q(s,a;\boldsymbol{w})$; $s \to Q[a]$
Initialize tf.keras action-value target model
$Q(s,a;\boldsymbol{w}^{-})$; $s \to Q[a]$ initially $w^{-} = w$
Initialize counter $t = 0$ and C
for episode = 1:500 do
    reset Gym environment
    for step = 1:500 do
        pick action $a_t$ epsilon-greedily
        Take $a_t$ and observe next state($s_{t+1}$) , reward
        and done (flag whether episode has finished)
        Add this experience tuple ($s_t$, $a_t$, $r_t$, $s_{t+1}$, done)
        to the replay buffer
        Randomly select a minibatch of size batch_size
        from replay buffer
        for each experience tuple in minibatch do
            if experience tuple is terminating
                $y_j = r_t$
            else
                $y_j = r_t + \gamma \max_a Q(s_{t+1}, a; \boldsymbol{w}^{-})$
            Do gradient descent step on cost function
            $[y_j - Q(s_t, a_t; \boldsymbol{w})]^2$
        $t += 1$
        if $t \% C == 0$
            $\boldsymbol{w}^{-} = \boldsymbol{w}$
epsilon = epsilon*epsilon decay rate
```

D. *Lunar Lander Gym Environment*

Lunar Lander is an OpenAI Gym environment where each state is an 8-dimensional vector with 6 continuous and 2 categorical variables. Four categorical actions are possible as follows, do nothing, fire the left orientation engine, fire the main orientation engine and fire the right orientation engine. The landing pad is located at

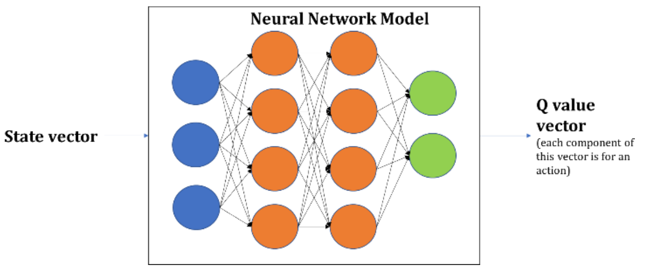

Figure 1. DQN prediction model architecture. Target model and additional details are not illustrated

coordinate (0,0) and the objective is to successfully land the lunar space module to these coordinates. An episode is considered done, if the lunar lander crashes or comes to rest at the landing pad with rewards -100 and +100 respectively which is added to the base reward accumulated up to that point. At each step, firing engines entails a small penalty which is -0.3 for the main engine, and -0.03 for right or left orientation engines. In this project, the objective is to train the value-function of an agent such that it achieves a score of +200 or better on average in the last 100 consecutive episodes during its training. State is represented as an 8-vector: $[x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R]^T$ where x and y are horizontal and vertical coordinates, $\dot{x}, \dot{y}$ are horizontal and vertical speeds, $\theta, \dot{\theta}$ are angel and angular speed of the lunar lander, leg_L, leg_R are binary values indicating whether the left or right leg of the lander is touching the ground. In this project, DQN algorithm is implemented to achieve the objective that is outlined above.

III. Experiments and Results

Figure 2 demonstrates agent's performance during 500 episodes of learning. Each episode has a maximum number of steps of 500. As can be seen, overall, as agent goes through more episodes, score values i.e. sum of rewards at each step, tend to get more positive, this indicates that learning is happening as more experience is accrued by the agent. However, learning does not happen at all combinations of the hyperparameters as will be seen later which commands the importance of tuning hyperparameter values to get good performance. As can be seen in Figure 2, the last 100 consecutive episodes has an overall positive score, with multiple scores >+200 and also multiple negative scores. The average was ~ +140.

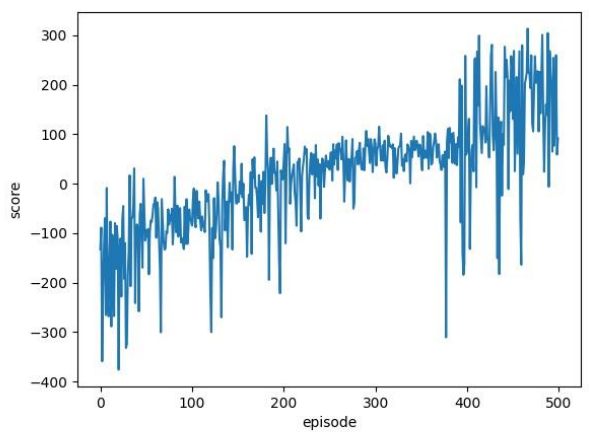

Figure2. Agent's score per episode during training

After the training phase, performance of the agent was tested. In the testing phase, the prediction model (not the target model) was used to compute action-values of each state and then the action with highest action-value was taken at each step. Scores in the testing phase are all positive except for a few episodes, which suggests that further hyperparameter tuning is required for this agent (average ~ +150)

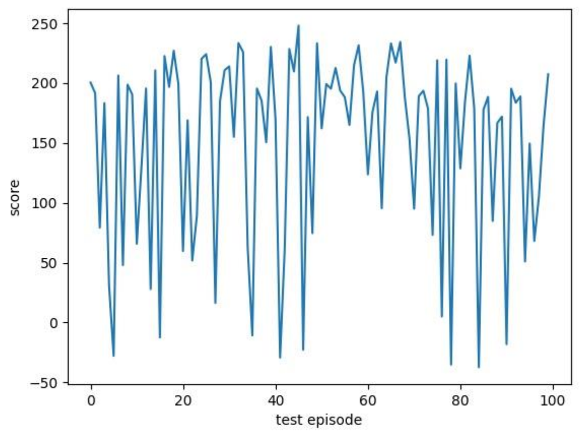

Figure3. Agent's performance during testing, i.e. no model weight update, only predict method of the model is called

A. *Impact of Batch Size*

Batch size that was used to generate figures 2, 3 was 32 but since this learning rate takes a long time to run, hyperparameter experiments were conducted using batch size 16 as default. Figure 4 illustrates the impact of batch size as a hyper parameter on the performance of the agent. Batch size is the number of (s_t, a_t, r_t, s_{t+1}) experience tuples that are randomly selected from replay buffer at each step. As outlined in Algorithm 1, at each step, a batch of experience tuples are randomly selected from replay buffer, and for each experience, target model is used to compute $y_j = r_t + \max_a \gamma Q(s_{t+1}, a_t; \boldsymbol{w}^-)$ and model is used to compute $Q(s_t, a_t; \boldsymbol{w})$ and then the square difference between these two terms would be the cost function to minimize by stochastic gradient descent. As is depicted in Figure 4, batch size16, seems to have a higher score overall compared to batch size 8 and 1. One advantages of sampling from replay buffer (pool of experiences) is that each experience is potentially used in many weight updates which is a more efficacious use of the available experiences. Another point of the experience replay approach is to avoid potential correlations among consecutive experiences. Experience replay approach also causes the behavior distribution to be averaged over many of its previous states which avoid oscillation and divergence.

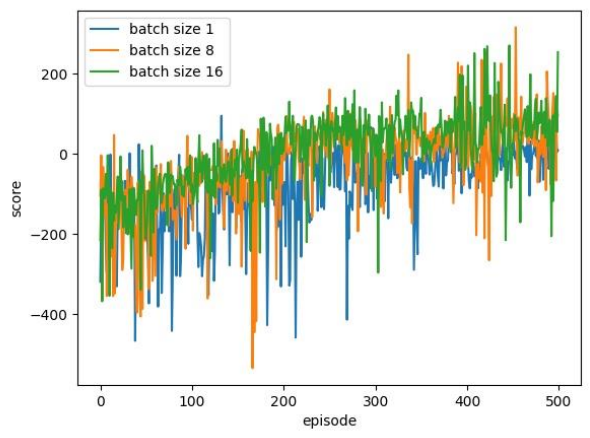

Figure4. impact of batch size on agent's performance

B. *Impact of Learning Rate*

The impact of learning rate on agent's performance is depicted in Figure 5. As can be seen, by decreasing the learning rate, agent's behavior becomes much smoother and overall scores gets more positive. The impact of learning rate can be better understood in the context of the stochastic gradient descent method that is used to train the prediction neural network model. The higher the learning rate is, the easier it is for the optimizer to miss optimums and therefore the model's weights do not specify the best function from the function space.

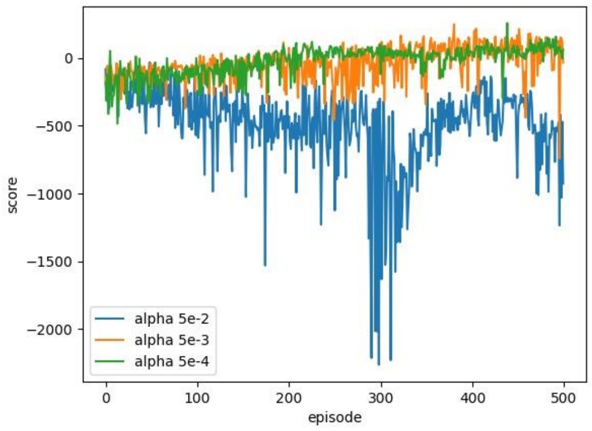

Figure5. impact of model learning rate on agent's performance

C. *Impact of Epsilon Decay*

The impact of epsilon decay rate as a hyperparameter was assessed and is depicted in Figure 6. As it is evident, without decaying epsilon value, i.e. epsilon decay = 1, performance is worst which gets better as epsilon gets slightly decayed more (0.995) and exacerbates again as epsilon decay rate is increased too much (0.9). Value of epsilon determines the trade-off between exploration and exploitation. Acting based on the model is not ideal in the initial stages of learning as it reduces the exploration of the environment. As episodes pass by, it's better to increase epsilon decay and thus rely more on the model for taking actions. The initial epsilon value in these experiments is 1 and when the epsilon decay rate is 1, it means that the agent selects all of its actions randomly and not according to the value-function. Therefore performance is worst and all negative for epsilon decay 1. As epsilon decay rate is increases slightly (0.995) agent would use the value-function with a higher probability as episodes progress, resulting in much better performance. At the very initial episodes, all three agents take actions randomly and more or less perform the same, however agent with epsilon decay 0.9 starts to rely on the model sooner and seems to perform better than agent with epsilon decay 0.995. However, this slightly better performance is temporary and near episode 200, the agent with epsilon decay 0.995 takes over. This could indicate that the agent with epsilon decay 0.995 has managed to reach a better tradeoff between exploration and exploitation as this agent relies less on the value-function model for a greater number of episodes.

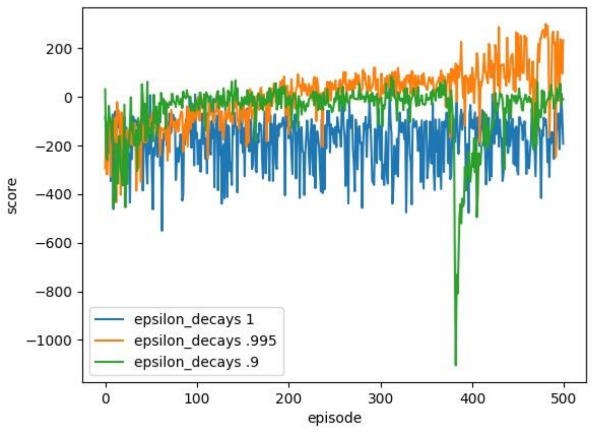

Figure 6. impact of epsilon decay on agent's performance

IV. CONCLUSIONS

In this report, DQN algorithm was implemented to train an agent capable of solving OpenAI Gym LunarLander-v2 environment. It was shown that for environments with continuous state spaces, tabular methods are inefficient and/or impractical and thus action-value function should be learned following a supervised learning paradigm where reinforcement learning updates are used as examples to train the action-value prediction model. The core concept in DQN algorithm is the usage of a neural network model for the action-value function prediction. TD(0) Q-learning updates are used as examples to train aforementioned model. To overcome oscillation and instabilities in the training of the prediction model, another neural network, target model, is used to generate TD(0) Q-learning updates. Furthermore, at each step, agent's experience is added to a replay buffer and prediction model weight updates are done using a randomly selected minibatch from the replay buffer. This approach has shown human level control in the original paper and is capable of solving the Lunar Lander problem.

V. REFERENCES

[1] R. S. Sutton, A. G. Barto, Reinforcement Learning, An Introduction, 2nd edition, MIT Press, (2020)

[2] Mnih, V., Kavukcuoglu, K., Silver, D. *et al*. Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015)

[3] Lectures of Georgia Tech’s CS 7642 (2021)