Git hash: b4983502fecbe7f881d11d865c7f9df52b5d33de

Shooting for the Stars with DQN

Trevor Skelton
CS-7642 Spring 2021
Georgia Institute of Technology, OMSCS
tskelton6@gatech.edu

I. Introduction

At the heart of reinforcement learning is value iteration, Q-learning, and the Bellman equations that describe these algorithms. A common issue in deploying these reinforcement learning algorithms into real-world scenarios is the difficulty of managing an enormous continuous state space that exists for many problems. In this paper, we'll explore one algorithm that has succeeded in making progress to solving this problem as evidenced by their agent's then start-of-the-art superhuman performance on the classic Atari 2600 video games [1]. Mnih et al. at DeepMind integrated a modern deep learning approach with Q-learning into an algorithm they coin a Deep Q-Network (DQN). We'll aim to reproduce a similar DQN and deploy it to solve OpenAI's Lunar Lander Gym reinforcement learning environment.

II. Methods

A. Q-Learning

In building up the DQN algorithm, our first stop is undoubtedly the Bellman equations. The Bellman equations specify for a reinforcement learning problem how to evaluate the optimal value of various representations of the state and action space and given a specific policy [2]. These equations are recursive in nature due to the temporal problem they aim to solve and interweave with one another. Key to the DQN is the following equation for the Q-value of a given state-action pair [2]:

$$q_*(s,a) \quad = \quad \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \quad (1)$$

To summarize, the current Q-value for a given state-action pair is equal to the expectation of the sum of the reward of the resulting state and the discounted Q-value of the following state-action pair, where the following action taken is the action that maximizes the Q-value at that state. R_{t+1} represents the reward received after taking action a at state s. The discount factor is given by the parameter γ. In a stochastic environment, this expectation is equivalent to applying the probabilities of transitioning to a subsequent state to each subsequent state's Q-value and summing the result.

In practice, we can only hope to estimate the optimal Q-value given a policy and a state-action pair. Q-learning is an algorithm to do just that. The update rule for Q-learning is as follows [2]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \quad (2)$$

Here, we aim to update the current Q-value estimate $Q(S_t, A_t)$ for some state-action pair at time t in the direction of the temporal difference (TD) error at some learning rate between 0 and 1 represented by α. The TD error term itself is the difference between what we hope to represent as Q*, given the Bellman equation above, and the current Q-value estimate. What makes the algorithm off-policy is that the action taken A_t does not need to be chosen by the current policy – it can be chosen by some other method. A common choice we will use for the purposes of this paper is ε-greedy policy, where the action has some ε between 0 and 1 probability to choose a random action, and otherwise it will take the action with the maximum Q-value of the next state in accordance with the Bellman equation. This randomness of action-selection is preferred in order for the agent to be forced to explore its environment, and is a key parameter in tuning the exploration versus exploitation behavior of an agent's policy space in reinforcement learning. The key to the algorithm is that the reward received will be discounted and propagate back to the Q-values of the states that led up to said reward.

In thinking about how to practically deploy Q-learning, an issue arises quite obviously: if the state space is continuous, as it often is in the real world, how can we practically store the required data to represent a fully continuous state space? The algorithm has no way to tell which states that may be very close together in a continuous state value are fundamentally similar or different for the problem. Solutions that have been presented include some form of human intervention, function approximation (linear and non-linear), and binning of the states into distinct categories [2]. For DQN, non-linear function approximation is our tool of choice. Luckily for us, we also have this wonderful "new" tool of non-linear function approximation that tends to work very, very well to solve machine learning problems: deep neural networks.

B. Deep Q-Network

It is beyond the scope of this paper to fully explore all aspects of the DQN algorithm itself, inclusive of a full explanation of neural networks and deep learning – basic understanding of neural networks is assumed. However, we will initially present a simple overview of the DQN algorithm and later in the paper explore its experimental results.

Neural networks have been utilized to extraordinary success in very difficult supervised learning problems such as computer vision, natural language processing, tabular data prediction, and more. In fact, with sufficiently large number of total neurons, it has been shown that a neural network is a universal function approximator [3]. Thus, the seem a natural choice for to approximate a non-linear function as we expect to have. For our reinforcement learning problem, we seek to utilize the neural network's flexibility to serve as a function approximator for Q in our Q-learning algorithm. The neural network can be built to take as input either a state or some state-action pair and output Q values for each action or simply the state-action pair itself, respectively. In this case of OpenAI Gym's Lunar Lander with discrete actions, the network will function in the former case, taking as input any possible state vector representing the environment and outputting a vector of Q-values, where each element of the vector represents one of the possible discrete actions able to be taken at that state. This is how Mnih et al.'s DQN was constructed, and as they state, it provides additional benefits in that a single pass through the network can compute all possible Q-values for a given state.

Meshing Q-learning and a neural network is not quite as easy as it may seem at first glance. For those who have understood the concepts discussed so far well, you may have noted that in a reinforcement learning problem, we simply don't have labels for our input data. Thus, how can we train a neural network to learn? Recall in Q-learning that we effectively need not know the optimal Q-value as long as we have enough experience receiving rewards from the environment. Using the Q-learning algorithm for our neural network's loss function means we need not know at the onset of learning what the true values are – with enough experience, the agent will receive enough rewards to learn a good representation of the Q-value for the state space.

However, this very concept introduces a problem or our neural network. Q-learning uses successive temporal states to update Q-values – in other words, it utilizes future rewards (estimated by future Q-values) to estimate the current Q-value. If the current Q-value is being updated by a neural network, then the Q-values for past states are also, in effect, being changed. This can lead to divergence of the algorithm. To solve this, Mnih et al. introduced a novel approach of utilizing a copy of the neural network to calculate the Q-value of the next state in the Q-learning update step. This copy, called the target network, will only be updated after a set number of weight updates, which means the current Q-value update will not immediately change other targets for Q-learning updates, and will lead to better stability of the network.

The final ingredient to the DQN is the use of what Mnih et al. call a *replay buffer.* While training, all sequences of state-action pairs and rewards are stored in the replay buffer, which the DQN will sample from in order to perform a step of stochastic gradient descent. This will cause the samples to be more independently distributed than a sequence of states, which are highly correlated.

C. *Experiment Setup*

We finally have all the pieces of the DQN algorithm. We aim to replicate the DQN in a simple setting and test a few of the algorithms hyperparameters. OpenAI's Gym – Lunar Lander is the perfect environment to train our DQN agent. The goal of this problem is for an agent in a 2D environment to fly and land successfully on the ground surface below it. The ground has a small landing pad at coordinates (0,0), but otherwise the ground can be quite uneven. The agent receives the following information about its current state: horizontal and vertical coordinates, horizontal and vertical speed, angle and angular speed, and binary values that represent whether or not the left and right legs of the agent are touching the ground. The agent can take one of four possible actions in any state, which are do nothing, fire the left engine, fire the main engine, and fire the right engine. The reward structure consists of a variety of factors. If the agent lands anywhere successfully, it receives 100 reward. If it crashes, it receives -100. Moving toward and away from the landing pad receives a small positive and negative reward respectively of equal value. Each action of firing any of the three engines incurs a -0.03 penalty. Finally, each episode will time out and end prematurely if the agent takes longer than 1,000 time steps (state transitions). The agent has "solved" the environment when it can achieve a mean of 200 reward over at last 100 consecutive episodes. More specifics about the Lunar Lander can be found at OpenAI's Gym resource website [4].

A special note on the timeout of an episode – there are a number of ways to address this state given there is no special reward given to the agent for a timeout. In the case of this implementation, we chose to have the agent estimate the Q-value as if the episode would have continued. In other words, a timeout was not treated as a terminal state, despite the fact that the episode does end at that state. There are different methods as to how to handle this, which would be worth exploring further.

A variety of hyperparameter values were tested, but not quite a full exhaustive grid search. The best found were utilized for the training of the agent and described here, while more detailed analysis of hyperparameter selection will be discussed further on.

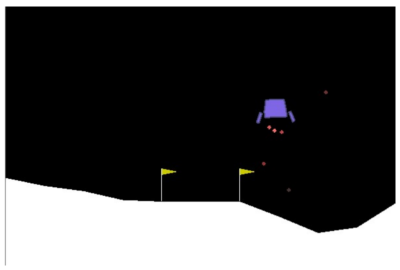

Fig. 1. An image of OpenAI Gym's Lunar Lander environment, where the Lander (purple) must carefully descened and land on the surface below

For this problem, a simple neural network with two hidden layers with ReLU non-linearities of size 128 and 30 neurons were used. The input layer was size eight for the eight different state parameters, and the output layer was of size four representing the Q-values of the four possible actions of the agent. All weights were initialized randomly. RMSProp was used as an optimizer with no momentum. A simple mean squared error loss was used for the training network. Stochastic gradient descent was performed with a batch size of 32. The target network utilized identical architecture and was set to copy weights from the training network every 100 weight updates. The exploration parameter ε was decayed linearly over the number of training episodes, starting at 0.7 and ending at 0.05. The learning rate for the neural network was set to 0.001, and the discount rate γ was set to 0.995.

As for the replay buffer, the buffer was filled with 50 episodes of data from a uniform random policy (the agent selecting actions randomly) before training began. The replay buffer was set to hold a maximum of the 10,000 most recent states. Finally, the agent trained during 200 additional training episodes. Every 10 episodes after training episode 100, the agent was evaluated by simulating episodes with a greedy policy and its mean and median reward over 25 episodes were recorded. If these were over 200, the agent was then evaluated again over 100 episodes. If its mean was over 200 again or better than any previous evaluation, the model recorded its weights and evaluation mean. The final model returned the weights of the most successful evaluation run with a mean over 200, or the final training's weights if no successful evaluation was run. For the purposes of our hyperparameter analysis, all of the above will be the *baseline* hyperparameters.

III. Results

A. Training

We present the agent's training represented by the rewards received for each episode of the agent's training in Figure 2.

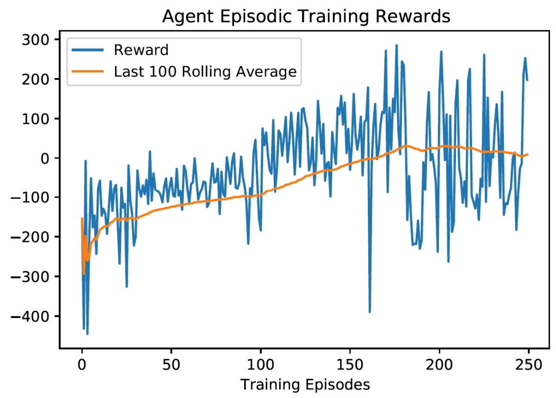

Fig. 2. Reward received each episode during the agent's training alongside the rolling average reward of the past 100 training episodes.

Following training, we ran the trained model through the environment over another 100 episodes with a greedy policy and recorded the results in Figure 3.

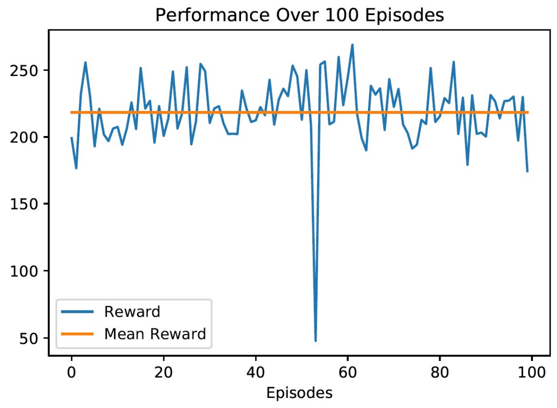

Fig. 3. Reward received each episode during the trained agent's evaluation utilizing the Q-greedy policy.

B. Hyperparameter Testing

To investigate further how varying the hyperparameters of the algorithm affected learning, we reran the same experiments as described above tuning one hyperparameter from baseline and recorded the results. Note that these particular results can vary with any random run of this model, but the overall shape of the results should hold. Note that all agents are identical to the baseline hyperparameters except specifically when mentioned.

First, we investigated how varying the replay buffer size would change the results. We trained agents with a size of 200,000 (large enough to capture all state-action pairs), 50,000, and 10,000. The results are shown in Figure 4.

Next, we varied the "exploration" parameter ε for selecting random versus greedy actions. We trained agents using an ε start at 1 decaying to 0.05, an agent starting at 0.7 and decaying to 0.05, and finally an agent starting at 1 and decaying to only 0.2. The results are shown in Figure 5.

Finally, we examined γ, the discount factor, to analyze how discounting the rewards would impact the model's performance. We decided to try the values 0.998 on the higher

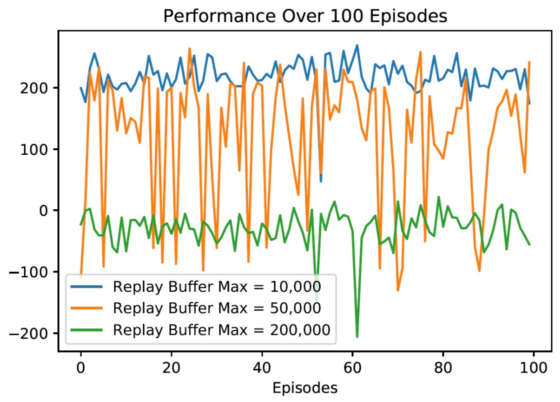

Fig. 4. Reward received each episode during trained agents' evaluation utilizing the Q-greedy policy with a buffer max size of 200,000, 50,000, and 10,000.

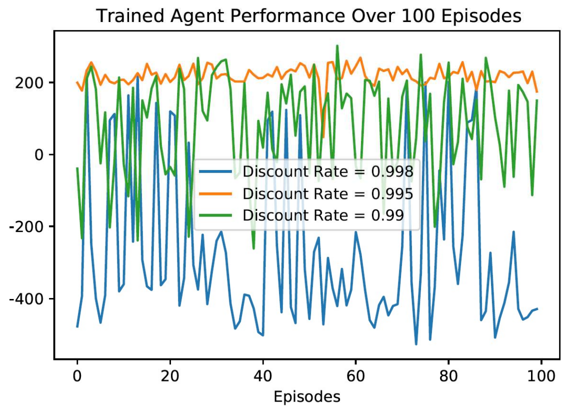

Fig. 5. Reward received each episode during trained agents' evaluation utilizing the Q-greedy policy with a discount rate γ of 0.998, 0.995, and 0.99.

end, 0.995 in the middle, and 0.99 on the low end, keeping in mind that each episode can last as long as 1,000 time steps.

IV. Discussion

A. Implementation

I utilized PyTorch for this implementation of DQN and found it fairly easy to learn and utilize. At first, I found training to be very slow, and only after a few days did I really examine each line of my implementation and think about how best to vectorize the operations I was performing. Narrowing down on the inner loop of training over each batch, I completely eliminated the loop by reformatting the code in a vectorized way, utilizing PyTorch tensors for the operations, which resulted in a 10x increase in the speed of training each episode. A key element of my implementation which I found critical to analyzing results was proper evaluation monitoring, as described in the experiment setup where the greedy policy was simulated fairly often for a small number of runs to identify candidate solution weights.

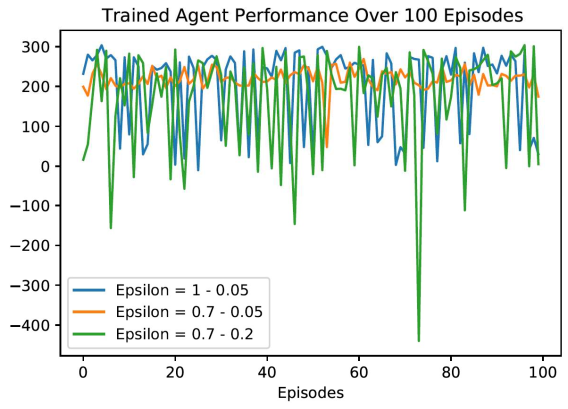

Fig. 6. Reward received each episode during trained agents' evaluation utilizing the Q-greedy policy with an ε decay starting at 1 and decaying until 0.05, versus starting at 0.7, versus decaying only 0.2.

B. Algorithm Analysis

One of the challenges I had attempted to reimplement this algorithm is how much randomness I found there was in the training process itself. While the environment itself is deterministic, as an off-policy learning algorithm using an ε-greedy action select, there is inherent randomness in which states are visited and what actions are taken. There is another layer of randomness in the initialization of the neural network weights themselves that can produce different results. Finally, we are randomly sampling the states visited themselves to perform updates for the Q-values. While many of these effects are minimal asymptotically, practically when testing the effects of these various hyperparameters in a reasonable timeframe, it becomes much more difficult to see clear patterns when the results have a significant variance.

C. Hyperparameter Analysis

A main change from the original DQN described in Mnih et al. that I have implemented is the selection of state-action pairs to sample from for a gradient descent step is limited to only the 10,000 most recent state-action pairs – a maximum of 10 episodes. In limiting the gradient descent update to calculate losses on only more recent samples of data, we a recency bias on the direction of the weight updates, the idea being that once the learner is on the right track, that more recent data is best to update on. With only 250 training episodes to work through, 10,000 outperformed 50,000 and 200,000 significantly, with 50,000 being in the middle but with a great deal of variance. I suspect with higher and higher training episodes, the larger buffer sizes would work to solve the problem as well. Comparatively, Mnih et al.'s model used a replay buffer size of 1,000,000 frames – which seemed to be more of a memory limit than a particular sample bias choice. Their model also needed to be extremely robust because the same model was deployed across many different Atari games with completely different environments, so such a simple heuristic used here may not produce generalizable results.

With *γ*, there are a few factors at play. It's important to consider that almost all of the initial episodes used for training involve crashing into the ground. Thus, too high a discount factor, and the model will actively avoid the ground to avoid the potential of crashing. This is primarily why 0.998 performed the worst. If the discount factor is too low, the positive rewards of touching the ground and landing won't propagate through the prior states' Q-values enough. As Fig. 5 shows, the middle-ground of 0.995 works best in this environment.

With ε decay, I found empirically the range between decaying 0.7 to 0.05 performing best. With a higher ε start, the beginning of training is too saturated with random data (given that the buffer is already filled with random data) and slows learning. Decaying only down to 0.2 means the model never gets to effectively learn in state ranges it will be seeing when it enacts its policy.

I did also try decaying the learning rate from 0.001 down to 0.0001, and while this seemed to work, I didn't see any marked change. Normally with neural networks, decaying learning rate over time is a good thing to do because the initial weights are completely random, but with reinforcement learning we are dealing with somewhat biased data compared to normal supervised learning. Maintaining the learning rate throughout meant later training data was not being weighted less than earlier data, which made sense to me.

D. Additional Thoughts and Experiments

I implemented the main change for the Double DQN algorithm as described in Hessel et al.'s Rainbow DQN paper [5] in a previous version of the model – selecting the action that maximizes the Q-value of the next state as specified by the training DQN instead of the target DQN – and found it did not substantially improve the results and computation time for an additional pass through the training network.

In terms of future investigations, the replay buffer itself I feel could be better optimized than was initially detailed in Mnih et al.'s initial paper, and if I had more time I would have liked to experiment even more with this concept. I see a fundamental problem in that if a particularly important state-action sequence is only seen late in training, it is unlikely that it will be sampled enough for the network to update the weights properly. Tuning this hyperparameter seems to be a tradeoff between sacrificing consistency over the entire state space with a faster convergence into a good policy on average, but I'm sure it is possible to create a more advanced sample selection, such as where Q-values are the largest magnitude from the target.

Another fundamental problem I see in the initial population of the replay buffer is that a uniformly random policy over the *actions* may produce state-action pairs that are not at all uniformly random over the *state space* from which the agent is attempting to learn and approximate a Q-function for. In the case of Lunar Lander, the lander has three different actions for movement and one action to do nothing. Choosing from these actions in a uniformly random way leads to the agent exploring the environment in a somewhat biased fashion. While the environment is seemingly wide open for the lander to move wherever it wants, it will explore the space nearby its starting position far more than the state space near the ground, because the ground is a terminal state and it will continuously crash when taking random actions near the ground. There may be adjustments to the algorithm that can be made to minimize the effect of this so that the agent can learn from a uniformly random sample space or from a prioritized sample space. There are some additional heuristics that would be good to test: for example, what would be the ideal size for the initial replay buffer size that is sampled from (heavily representative of the Q-values updated during early training)? What proportion of the possible state-action space should it attempt to cover?

I'd like to think more about how to separate the process of data generation (the agent actually acting in the environment) and the process of actual learning (updating the weights of the DQN through gradient descent) – or at least better understand how to ensure they can harmoniously coexist without unintentional interactions. They are separate processes, but in the algorithm described by Mnih et al., they are detailed to be interwoven with one another because each timestep per episode, a gradient descent update is performed. I could envision this being problematic for learning. For example, longer episodes with many state changes will perform more gradient updates. On average this means the agent will actually perform better on longer episodes than shorter episodes, which slightly biases the data. States from longer episodes will also take up a larger proportion of the replay buffer than shorter episodes, meaning state-action pairs from longer episodes are more likely to be sampled for updates. In problems where episode duration can vary by a large number of time steps, this may be problematic if each episode is equally important to learn from.

This interacting effect of experience generation and learning has consequences on other algorithm parameters such as the probability to select a random action ε, the target network update (which is set in number of weight updates, not episodes), the maximum size of the replay buffer (set in state transitions), and the number of training episodes required for good performance and convergence in general. In particular, I would like to experiment further with an ε that directly responds to the evaluation metric – in this case the agent's performance over some representative number of episodes setting ε equal to zero. The idea being that if the agent is performing well, its exploration parameter should be fairly low so that it can focus on learning about the states nearby where it is consistently seeing. Meanwhile, if the agent is doing poorly, its exploration parameter should be set fairly high so it can escape its current suboptimal policy and learn more from the environment. Utilizing this metric may reduce the time to find an optimal policy considerably, at the cost of not properly learning about the entire possible state space. Depending on the problem, this may be highly desirable if the agent does not need to perform well on other outlier states, as long as its current policy for the problem is sufficiently good – in other words, if the environment possesses little randomness that could bring the agent into a state it has not frequently observed during training.

All in all, while DQN itself is very flexible to many kinds of RL problems due to neural networks themselves being extremely flexible, it comes at the cost of requiring a great deal of tuning of the sheer number of hyperparameters to achieve optimal performance. In that regard, I would speculate that DQN may be not a highly portable algorithm and not ideal in cases where the problem space changes frequently. However, fine-tuning this algorithm to a particular problem that suits it can produce extremely impressive results and make many challenging reinforcement learning problems much more tractable.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* "Human-level control through deep reinforcement learning," *Nature,* vol. 518, pp. 529–533, Feb 2015.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2nd Ed. MIT press, 2020. URL: http://incompleteideas.net/book/the-book-2nd.html

[3] Hornik, K. "Approximation capabilities of multilayer feedforward networks," *Neural Networks,* vol. 4, no. 2, pp. 251–257, 1991.

[4] "LunarLander-v2". Gym.openai.com. https://gym.openai.com/envs/LunarLander-v2/ (accessed Mar. 19, 2021)

[5] Hessel, M., Modayil, J., van Hasselt, H., *et al.* "Rainbow: Combining Improvements in Deep Reinforcement Learning," in *the AAAI Conference on Artificial Intelligence (AAAI)*, 2018.