Deep Q-Learning

Scott Merrill
Georgia Institute of Technology
smerrill7@gatech.edu
git hash: c0c8e87e687b36d5610c91c064fe715cf76fe7e1

I. Introduction

By rewarding desired behaviors and penalizing unwanted actions, Reinforcement Learning applies concepts from classical conditioning to train agents to optimally navigate an environment. In 1992, Chris Watkins presented the proof of convergence of a technique known as Q-Learning whereby agents can be conditioned to identify and exploit the series of actions that maximizes their cumulative reward [2]. Such techniques, however, require finite state spaces and thus are applicable only to select domains. Deep Q-Learning is an extension of the original Q-Learning algorithm that is scalable to infinite state-space environments. While Deep-Q Learning algorithms aren't guaranteed to converge, several strategies can be implemented to make convergence more likely [1].

In this paper we demonstrate the training of a Deep Q-Network to teach a virtual agent to solve OpenAI Gym's LunarLander-v2 environment. We demonstrate the use of experience replay, fixed Q-Targets and ϵ −greedy exploration which help support convergence and additionally explore how hyperparameters can be tuned to enable efficient learning. This work serves to demonstrate the efficiency of properly parameterized Deep Q-Networks and highlight the broadened class of infinite state space problems they can solve.

II. Q-Learning

A. Reinforcment Learning

Reinforcement Learning problems model the environment as a Markov Decision Process (MDP), where at any time *t*, an agent is in some state *s*, and can chose from a set of actions *a*. Based on the selected action, the environment then returns the resulting reward and new state from selecting this action. This basic reinforcement learning framework is shown in Figure 1.

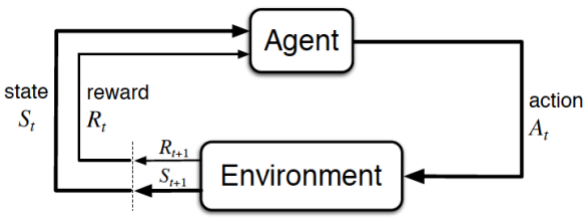

Figure 1

The goal of the agent is to select the set of actions that yield the highest cumulative reward. The underlying properties of the MDP may be known or unknown. If the underlying transition probabilities and reward for each state are known, then it is considered a model-based system and can be solved through planning. However, a model for the environment is rarely known in advance, and oftentimes the agent must learn to maximize reward through trial and error. In these cases, the Reinforcement Learning framework is said to be model-free, which is the focus of this paper.

B. Algorithim

The Q-Learning algorithm is a model-free strategy to identify the optimal policy of an agent. To illustrate the algorithm, consider an unknown environment with state space S and action space A. Let the Q-Function, $Q^*(s,a)$, be the state-action value function that provides the value of being in a particular state *s* and taking a particular action *a*. Since the goal of the agent is to identify the policy $\pi^*(s)$ that selects the action resulting in the largest cumulative reward, the policy is related to the Q-Function by (1).

$$\pi^*(s) = argmax_a Q^*(s,a) \tag{1}$$

Since a model of the environment is unknown, the agent must approximate the optimal Q-Function through trial and error. To estimate this state-action value function, $Q(s,a)$ is initialized to arbitrary random values and updated with each transition (s_t, a_t, r_t, s_{t+1}), where s_t is the starting state, r_t is the reward received for taking action a_t and s_{t+1} is the resulting state after taking action a_t. At each time-step, the current values of $Q(s,a)$ are updated toward the Q-Target – an approximation of the optimal Q-Function defined in (2).

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) \tag{2}$$

In (2), γ is a discount factor used to normalize rewards received in the distant future. Intuitively, the value of any state action pair is the reward for taking some action a_t and the discounted value of being in a new state s_{t+1}. A learning rate parameter α determines the amount with which the Q-Function is updated towards the Q-Target. The complete learning rule is thus presented in (3).

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3}$$

With repeated application of this update rule and adequate exploration of the state space, it has been shown that $Q(s_t, a_t)$ converges to $Q^*(s_t, a_t)$ with probability 1 [2].

III. Deep Q-Learning

While Q-Learning provides a very simple framework and update rule, its limitations become clear as state and action spaces grow large. The Q-Learning algorithm described above is considered tabular since it requires a lookup table to store the values of each state-action pair. In continuous state spaces, a lookup table becomes unfeasible and approximation methods are necessary to generalize similar state spaces. In this paper we consider Deep Q-Networks (DQN) which use Artificial Neural Networks (ANN) to approximate the Q-Function enabling Q-Learning techniques in infinite state space models.

A. Feed Forward Artificial Neural Network

ANN are a supervised learning technique based loosely on the human brain. Feed Forward Networks (FFN) are the simplest form of ANN in which information moves in a single direction through multiple layers of neurons to produce predictions. FFN consist of an input layer, optional hidden layers and an output layer. Information is passed sequentially as shown in Figure 2, such that a neuron in layer i+1 is the weighted sum of all neurons in layer i.

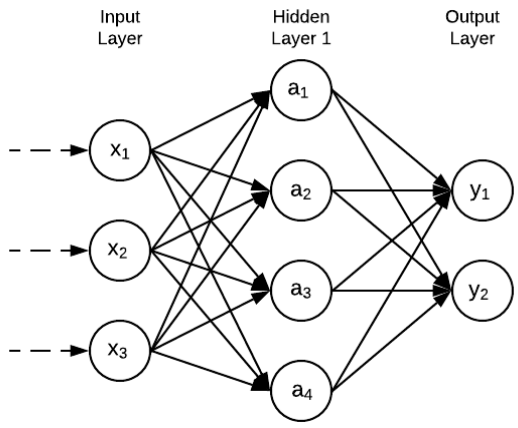

Figure 2

To illustrate this, let x_1 denote the first neuron in the input layer, and a_1 to denote the first neuron in the hidden layer. Additionally, let $w_{ij}^{(1)}$ be the weight vector connecting the input layer to the hidden layer, such that element ij in the vector corresponds to weight connecting ith neuron in the input layer to the jth neuron in the hidden layer. Thus, x_1 affects every neuron in the hidden layer proportional to the weight $w_{1j}^{(1)}$. Or, equivalently stated, neuron a_1, is affected by all neurons in the input layer by the corresponding weight $w_{i1}^{(1)}$ connecting them as shown in (4).

$$a_1 = x_1 * w_{11}^{(1)} + x_2 * w_{21}^{(1)} + x_3 * w_{31}^{(1)} \quad (4)$$

Some representations of FNN use activation functions to further frame the value of each neuron. A simple activation function is the rectified linear (relu) function defined as $f(x) = \max(0, x)$. The calculation of a_1 using the relu activation function is shown in (5).

$$a_1 = \max\left(0, \; x_1 * w_{11}^{(1)} + x_2 * w_{21}^{(1)} + x_3 * w_{31}^{(1)}\right) \quad (5)$$

The values of each neuron are calculated sequentially in this manner to produce the predictions in the output layer. The weights of a feed forward network are trained on labeled data as to minimize a loss function.

B. FFN Q-Function Approximator

In the context of Q-Learning, a FFN Q-Function approximator $Q(s_t, a_t\, \theta_i)$ with weights θ_i is trained by adjusting the weights at each iteration i to minimize loss between the optimal Q-Function. Since the optimal Q-Function is unknown, it is modeled by a second FFN with weights θ_i^-. The Q-Target is thus approximated by $y = r_i + \gamma \max_a Q(s_{i+1}, a_{i+1}\, \theta_i^-)$. Stochastic gradient descent is then performed to adjust the weights θ_i to minimize the mean squared-error between the Q-Function and the Q-Target.

$$L_i(\theta_i) = \left(r_i + \gamma \max_a Q(s_{i+1}, a_{i+1}\, \theta_i^-) - Q(s_i, a_i\, \theta_i) \right)^2 \quad (6)$$

C. Stability Issues

It is known, however, that when non-linear function approximators such as FFN are used to estimate the $Q^*(s, a)$, convergence is not guaranteed because of the correlation between consecutive (s_t, a_t, r_t, s_{t+1}) transitions and the correlation between $Q(s_i, a_i\, \theta_i)$ and the update target $r_i + \gamma \max_a Q(s_{i+1}, a_{i+1}\, \theta_i^-)$ [1]. Two techniques are employed to reduce these correlations.

To address the correlation between consecutive transitions, a technique known as experience replay can be used to train a FFN. Experience replay refers to the storage of (s_t, a_t, r_t, S_{t+1}) transitions for later use. When training a FFN, random samples from replay memory are used thus reducing the correlation between training samples and the risk of diverging solutions.

Additionally, fixed Q-Targets are used to address the correlation between the Q-Function and the Q-Target. With fixed Q-Targets, two representations of the FFN are stored; one with weights θ_i and one with the target weights θ_i^-. After C gradient decent updates, the target weights are reset to the

current Q-Function weights. Thus, the Q-Learning update rule bootstraps the Q-Function toward a frozen representation of itself. This technique limits the correlation between the Q-Function approximate and the Q-Target and prevents the unstable situation in which updating the weights will change both the Q-Function and the Q-Target.

IV. Lunar Landing Problem

A. Description

We now demonstrate an application of a DQN to solve OpenAI Gym's LunarLander-v2 environment. The goal of the environment is to teach a lunar lander to successfully land on randomly generated surfaces of the moon. The state space $S \ni (x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R)$ includes 6 continuous variables and two binary flags. The variables x, y correspond to the agent's vertical and horizontal position; $\dot{x}$ and $\dot{y}$ indicate vertical and horizontal velocity while θ *and* $\dot{\theta}$ provide the agent's angular position and velocity. The binary leg_L and leg_R flags indicate whether the agent's legs are in contact with the moon. Given any state, the agent can choose from four actions; firing the main engine, firing the left engine, firing the right engine or turning the engines off.

The agent is continuously rewarded up to a maximum of 140 points as it moves closer to the landing pad. Any move away from the landing pad will result in an equivalent negative reward. In addition, the agent is rewarded 100 points for a successful landing and deducted 100 points for crashing. For each leg that finishes in contact with the moon, the agent is rewarded 10 points. Finally, the agent is penalized 0.3 points for each fire of the main engine and 0.03 for each fire on the left or right engines. The problem is considered solved when a score of 200 points or higher is achieved on 100 consecutive episodes.

B. Deep Q-Learning Implementations

To solve this problem, we implemented a DQN consisting of two hidden layers and a linear output activation function. The first and second hidden layers consisted of 1000 and 250 nodes respectively each with relu activation functions. A learning rate of 0.00025 was selected with Adam's adaptive learning rate optimization to minimize the mean squared error loss function. The network was trained with 100 random samples drawn from replay memory at each iteration. The weights of the Q-Target FFN were reset to the weights of the Q-Function's FFN after 25 iterations.

Our agent learned off policy following an $\epsilon - greedy$ exploration strategy in which it would act randomly ϵ percent of the time and act greedily $1 - \epsilon$ percent of the time. ϵ was initially set to 1.0 such that our agent would act randomly 100%. With each iteration, the value was decayed by a factor of 0.999 subject to a minimum value of 0.01. Complete pseudocode is shown in Algorithm 1.

Algorithm 1: ϵ-greedy Deep Q-Learning

Initialize replay memory of size N
Initialize Q-Function Approximator with weights θ_i
Initialize Q-Target with weights θ_i^-

For each episode:
  Initialize s_i to starting state
  For each transition:
    Select a random action with probability ϵ, otherwise select $argmax_a Q(s_i, a_i)$
    Observe new reward r_i and state s_{i+1}
    Store the transition (s_i, a_i, r_i, s_{i+1}) in replay memory
    Randomly sample N observations from replay memory

Set Q-Target: $y_i = \begin{cases} r_i & if\ episode\ terminates \\ r_i + \gamma \max_a Q(s_{i+1},\ a_{i+1}, \theta_i^-)\ otherwise \end{cases}$

    Perform gradient descent on $(y_i - Q(s_i,\ a_i,\ \theta_i))^2$
    Anneal ϵ by decay factor
    After C steps set $\theta_i^- = \theta_i$
  End For
End For

The results for our model with these parameters are shown in Figure 3. Overall, these agents perform quite well, solving the problem and converging to the optimal solution in 155 training episodes.

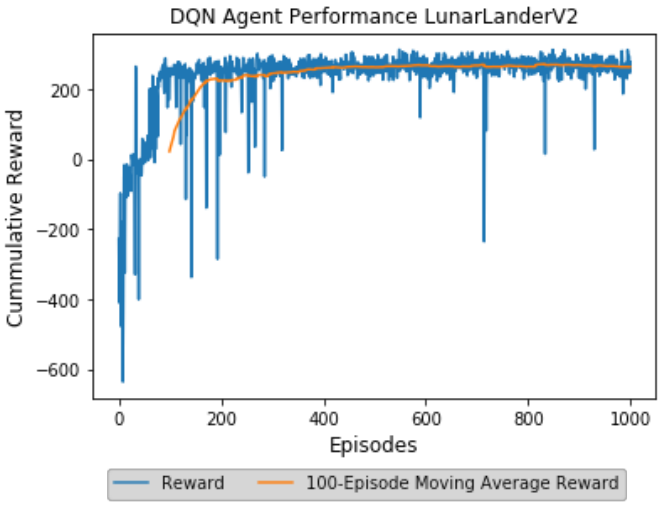

Figure 3

V. Experiments and Hyperparameter Tuning

The performance of our DQN was proven to be highly sensitive to the input parameters. Thus, several experiments were run to determine the optimal hyperparameters.

A. Optimizing Neural Network

Our first experiment attempted to optimize the overall configuration of our FFN. In total 24 different configurations were tested. We considered a network with an input layer, two hidden layers and one output layer. The number of

hidden layers was chosen strategically to allow for the modeling of complex boundaries while also avoiding overfitting. While each hidden layer used the relu activation function, 4 different activation functions were considered on the output layer; linear, relu, sigmoid and tanh.

While the linear activation function doesn't bound the output value, the relu, sigmoid and tanh are all bounded. The relu function prevents negative values, the sigmoid function bounds the output to between 0 and 1 and the tanh is defined from -1 to 1. Each output activation has its theoretical advantages, but the linear function was the only one that produced consistent convergence. The results for the linear activation function on several network sizes are shown in Figure 4. Surprisingly, the relu activation function converged and performed quite well for only one network with hidden layers of size 400/200.

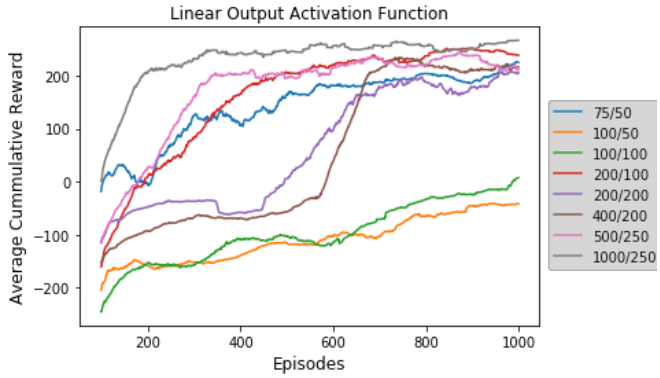

Figure 4

Since our FFN is modeling a Q-Function which can take an infinite number of possible values, the linear function does make the most sense. Furthermore, since the scale of the reward ranges from -100 to 100, bounding the Q-Function between -1 and 1 results in the immediate reward dominating the update rule in (6). Thus, with the sigmoid and tanh output activation functions decisions are made largely on maximizing one step reward rather than maximizing cumulative reward. The reward would need to be scaled if either of these activation functions are to be reconsidered in the future.

Bounding created similar issues for the relu activation function. By setting the minimum Q-Function output to zero, states that are considered extremely bad and only moderately bad are not differentiable. Since all bad states are treated the same, extremely bad states aren't actively avoided leading to sub-optimal decisions. Overall, in the context of the problem, the linear activation function makes the most sense to model the Q-Function and empirical results support this fact.

B. *Adaptive Learning Rates*

Our second experiment was conducted to find the optimal learning rate parameter for our given network. 7 different learning rates were considered as well as well as 3 different adaptive learning rate algorithms; RMSProp, AdaGrad, and the Adam optimizers. Each of the adaptive learning rate algorithms dynamically adjust the learning rate for each weight parameter in the network. AdaGrad sets larger learning rates to parameters that are updated more infrequently, RMSProp updates learning rates based on magnitudes of recent changes to network weights and the Adam's optimizer updates learning rates based on recent magnitudes and variances of weight updates.

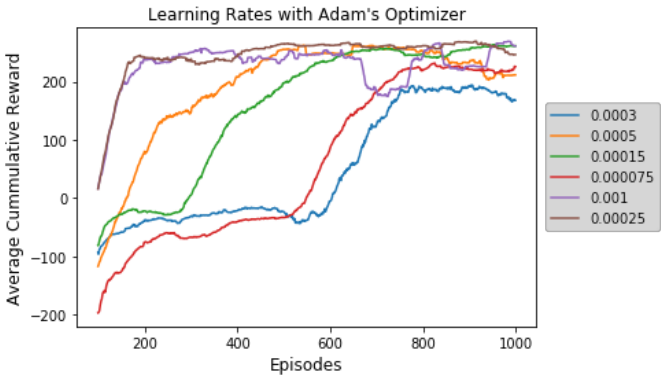

Figure 5

The best overall results were achieved using the Adam's optimizer, however the RMSProp algorithm performed well for select initial learning rates. The AdaGrad algorithm failed to converge for every learning rate tested. We believe this is partly due to the overwhelming size of the network. With two hidden layers of size 1000 and 250, there are a total of 260,254 tunable weights. Given the size of the network, it's likely the case that every weight is updated each time the network is trained; changing one weight will likely create a chain reaction of updates to other weights in the network. And, since AdaGrad adaptively decreases the learning rate for parameters that are updated frequently, such will result in deflated learning rates and slow convergence.

The RMSProp and Adam's optimizer don't face the same problem as AdaGrad. These algorithms update the learning rate based on the average change and average volatility of each particular weight. Moreover, when a particular weight begins to converge on some value, learning rates will automatically decay. Lowering the learning rates as parameters converge is the ideal behavior. In gradient descent, there's the inherent tradeoff between convergence speed and the optimality of the solution; the higher the learning rate, the quicker the learning but also the potential to overshoot the global minimum. The lower the learning rate, the slower the convergence but the more likely we are to find the global minimum.

C. *Loss Function*

Our third experiment considered three loss functions; the mean-squared error (MSE), mean absolute error (MAE) and mean squared logarithmic error (MSLE). The results for each loss function are shown in the Figure 6. Both the MAE and MSE performed quite well. The MAE found the optimal

reward quicker, however it is not monotonic, nor did it converge over 1000 episodes.

Figure 6

Mathematically, the MSE and MAE are quite similar. Unlike the MAE, the MSE squares the difference between the predicted output and actual output making it more sensitive to outliers which may explain the initial outperformance of the MAE. However, the major issue with the MAE was its lack of convergence. After careful investigation, this can be explained by looking at the gradient of MAE. Being a piecewise function, the gradient of the MAE is constant, and thus the same for small and large losses. Ideally, larger gradients for large losses and smaller gradients for smaller losses are preferred. This constant gradient prevents the MAE from converging even with adaptive learning rates. Unlike MAE, the MSE is parabolic and thus operates efficiently with smaller gradient updates for smaller losses.

The MSLE is akin to MSE, however it is concerned with percentage error in estimations. The MSLE was the worst performing loss function which is likely due to the bounded nature of the formula preventing extremely large losses. The MSLE also suffers the same gradient issues as the MAE. Being a logrithmic function, large and small errors are treated approximately the same.

D. Epsilon Decay Speed

Our fourth experiment tested the decay speed of ϵ. As our implementation is an ϵ-greedy off policy DQN, the value ϵ determines how often our agent acts randomly and how often it acts greedily exploiting the best policy. ϵ was initialized to 1.0 and decayed by the decay factor after each transition. In general, higher decay factors outperformed lower ones as shown in Figure 7. However, extremely high decay factors such as 0.999999 performed the worst. This makes sense as with such a high decay factor our agent is selecting most actions at random for thousands of episodes. Considering there are on average 100 transitions in an episode, after 1000 episodes with a decay speed of 0.999999 our agent is acting randomly 90% of the time ($0.999999^{100*1000}$). The optimal decay speed and that which was used in the final model was found to be 0.999, indicating that after 20 episodes, the agent is acting greedily roughly 86% of the time (0.999^{100*20}).

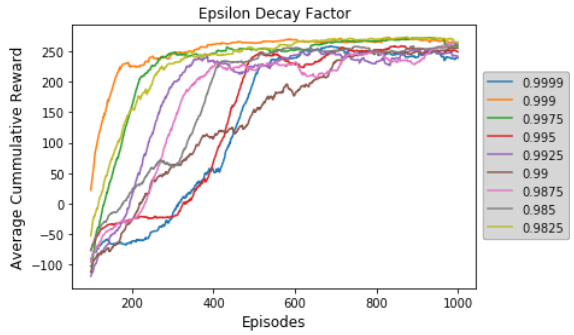

Figure 7

E. Discount Factor Selection

Our final experiment considered 14 discount rates. The results for this experiment are shown in Figure 8. The discount factor producing the quickest convergence was found to be 0.995. Interestingly, any discount factor higher than this prevented convergence. Such high gamma values encouraged the lander to land as quickly as possible instead of taking an infinite stream of slightly negative rewards. The priority with which the agent placed on landing quickly prevented it from ever properly landing and converging on a solution. In the other extreme, discount rates below 0.98 resulted in the lander taking its time to land. With many failed attempts at landing and receiving a -100 reward, the agent decided the optimal decision was to remain in the air indefinitely and accepting a reward of -0.3; this makes sense as the value of remaining ascent is an increasing function of the discount rate.

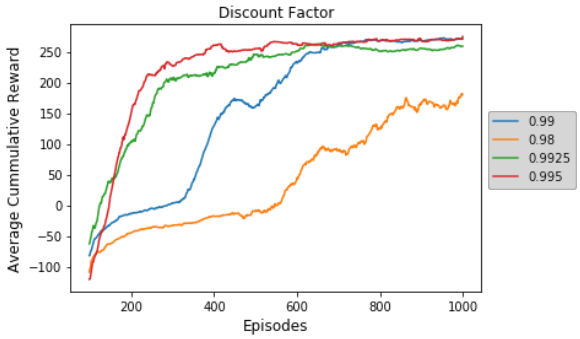

Figure 8

VI. Conclusion

Solving AI Gym's LunarLander-v2 with a DQN proves the usefulness of FFN approximators for the Q-Function. Additionally, strategies such as experience replay, and Q-Target fixing are useful in aiding the convergence of non-linear Q-Function approximators. While basic parameter tuning of the DQN was enough to solve the LunarLander-v2 environment, more complex problems may require more sophisticated tuning. Thus, more efficient strategies to train parameters simultaneously remains to be investigated.

References

[1] Mnih, Volodymyr, "Playing atari with deep reinforcement learning." (2013)

[2] Watkins and Dayan, "Q-Learning." (1992)