Q-Learning with Function Approximation on LunarLander-v2

Nancy Trinh

Abstract

This report presents a solution to the OpenAI LunarLander-v2 problem using Q-Learning with Function Approximation. The approach is adapted from the deep Q-network specified in Mnih et al. *Human-level control through deep reinforcement learning, Nature 518, 2015.* The effects of select hyperparameters on the agent's learning are also presented here.

1. The Agent

The task is to interact with an environment through a sequence of observations, actions, and rewards, and the goal is to select actions to maximize cumulative expected reward. This is well suited for a reinforcement learning algorithm. The function $Q^*(s, a) = max_\pi \mathbf{E}[r_t + \gamma r_{t+1} + \gamma r_{t+2} + ...|s_t = s, a_t = a, \pi]$ returns the maximum sum of rewards r discounted by γ at each time step, that can be achieved by starting in the state s, choosing the action a, and following the policy π thereafter.

The state space is infinite, so a function approximator is better suited for holding the state-action values than a table. Discretizing the state space is also an option, but I did not have much success with it in my experiments (k-nearest neighbor was attempted with varying values of k). RMSprop was used as the optimizer in the paper and in my experiments.

I used a neural network as the function approximator because neural networks have been shown to work very well in this capacity, and my experiments with linear function approximators and trees were not successful. I used a feedforward network with one hidden layer instead of the multi-layered convolutional network specified in the paper, and (state, action, value, next state) tuples as inputs instead of the pixels in frames of the game. The lunar lander problem is much simpler than the Atari environments and did not require that much complexity.

Online reinforcement learning with nonlinear function approximators and a mean squared error loss function is prone to instabilities for several reasons: (1) the examples are presented in a highly correlated fashion, (2) large error values can have an outsized impact on the network weights, and (3) the target values $r + max_{a'} Q(s', a')$ and the action-values Q are correlated (increasing $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a, which then causes the target to increase). To combat these shortcomings, I used experience replay, an alternative loss function, and a target network.

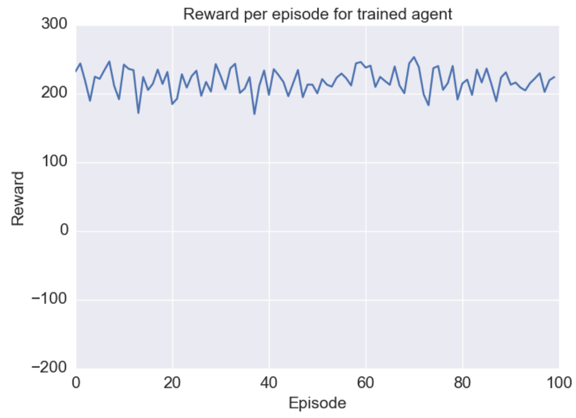
Experience replay reduces correlation in the examples that are presented to the online learner. Experience tuples (s, a, r, s') are stored in a memory bank. Q-learning is performed on mini-batches drawn randomly from the memory bank. I limited the size of the memory bank to 100,000 examples and popped out the oldest example when the bank became full, so that the most recently encountered examples were stored. The authors in the paper use a memory capacity of 1,000,000 examples and mini-batches of 32, but I found that a capacity of 100,000 examples and mini-batches of 64 were sufficient for the agent to solve the problem. When the agent is initialized, it has no experience, and thus, no memory, so I directed the agent to take random actions and store its experiences until its memory bank is full. Only then do I start training the agent.

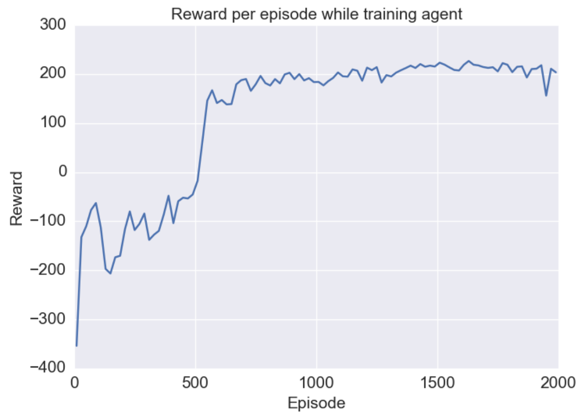
Using the loss function $\sqrt{1 + error^2} - 1$ instead of mean squared error reduces the impact that large error values can have on the network weights by "squashing" it. This function is a smooth approximation to the Huber loss function.

The target network reduces correlation between the target value and the action values. It involves training two networks with identical architecture. Network A is updated frequently while Network B is updated periodically by cloning the weights from Network A. Network B's updates are effectively snapshots in time of Network A. The intervals between updates of Network B should be long enough to allow Network A to converge. I updated the target network every 1000 updates instead of every 10,000, as in the paper, and this was sufficient to obtain a solution.

The agent solves the problem with the following settings for these hyperparameters: learning rate: 0.0001, epsilon between 1 and 0.1 with exponential decay ($\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-0.001t}$), and gamma of 0.99. I maintained the default maximum number of timesteps at 1000, since this provided enough timesteps for the agent to figure out how to land properly, and did not greatly increase training time.

Figure 1 (left) shows the reward per episode obtained by the trained agent. Note that the agent performed relatively consistently per episode. Figure 2 (right) shows the reward per episode while training the agent. The agent learned very quickly; it improved dramatically in the first few hundred episodes. It converged around episode 700 and maintained a high level of performance thereafter.

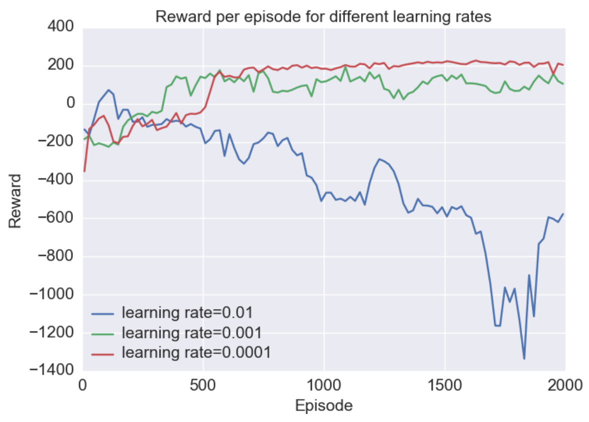




2. Effect of Hyperparameters

Figure 3 (left) shows that a learning rate that is too high can lead the agent to learn a disastrous policy (learning rate=0.01). A learning rate of 0.001 led the agent to learn faster than the agent with the learning rate of 0.0001 (episodes 400 to 500), but it converged to a worse policy than the agent with the lower learning rate (episodes 700+). It was important to choose a learning rate that was small enough to find a good policy but not so small that it would get stuck in local minima.

Figure 4 (right) shows that the initial (max) value of epsilon did not have a large impact on the agent's ability to solve the problem. A very low value of max epsilon (0.1) seemed to make the agent more unstable than higher values, but the difference between an agent with max epsilon of 0.5 and 1 is not great. This may be because I directed the agent to take completely random actions until its memory bank is full. Each episode takes approximately 100 timesteps, so it takes around 100 episodes to fill up its memory bank (capacity 100,000). Note that the learning curves are identical for the first 100 episodes. 100 episodes of pure exploration were enough for the agent to eventually learn to solve this particular problem.



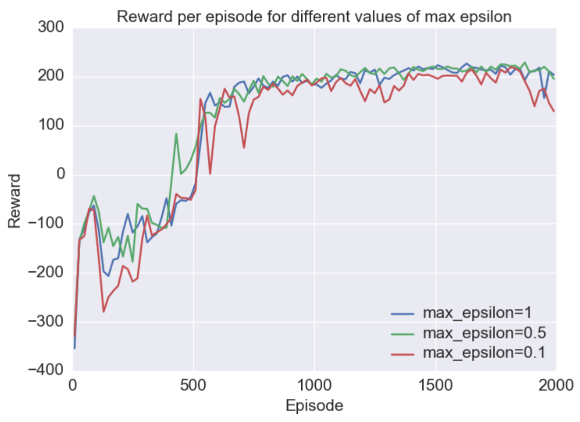
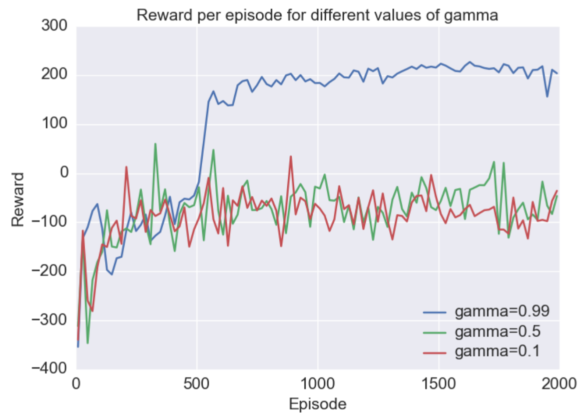
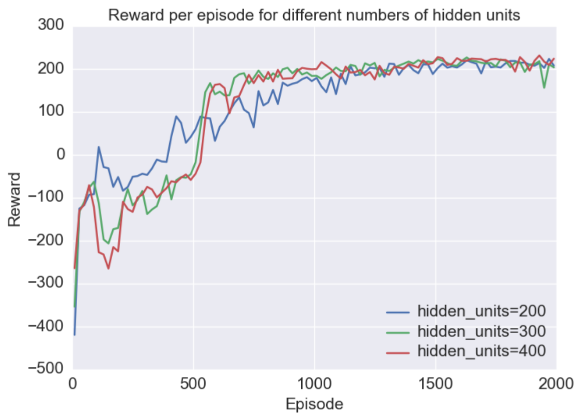


Figure 5 (left) shows that gamma values that are set too low affect the agent's performance considerably. When gamma was set to 0.5 and 0.1, the agent failed to solve the problem entirely. If the gamma factor is set too low, the agent fails to plan far enough into the future to make the right decisions. A high discount factor sets a high weight on future outcomes, and helps the agent with long-term planning.

Figure 6 (right) shows that when the number of hidden units is set in the neighborhood of 300, the agent solves the problem after approximately 1000 iterations. It is encouraging that the agent is fairly robust to this parameter, and it demonstrates the power of the approach. The biggest challenge for this problem was finding an approach that worked. Discretizing, using a tree-based approach, and a simple feedforward network without the target network and experience replay all failed. The DQN-inspired approach provided the requisite complexity and stability to solve the problem after some parameter tuning, using the values specified in the paper as defaults.


Reward per episode for different values of gamma
Reward
Episode
gamma=0.99
gamma=0.5
gamma=0.1


Reward per episode for different numbers of hidden units
Reward
Episode
hidden_units=200
hidden_units=300
hidden_units=400