Report

December 25, 2020

1 Project 1: Navigation

This project solves the single agent Banana Collector Unity environment.

1.1 Environment details

The Banana Collector environment involves collecting bananas in a large, square world.

Each yellow banana collected gives +1 reward, and each blue banana gives -1.

There are 37 continuous dimensions in the state space including the agent's velocity, direction, and ray-based object perception.

The action space consists of four discrete values: move forward, move backward, turn left, and turn right.

The environment is considered solved when the agent gets a score of +13 over 100 consecutive episodes.

1.2 Implementation

The code borrows heavily from the code in the Deep Q-Networks lesson.

The code trains a Deep Q-Learning agent to solve the banana collection environment. DQN involves using a neural network (rather than a table) to represent the optimal action-value function of the agent. Using this functional approximator makes it possible for the agent to learn in this environment, which has a continuous state space. Once an optimal action-value function Q with weights w* has been obtained, it can then be used to select actions in each state that will maximize reward, namely argmax_a of Q(s,a,w*).

The initial parameters of the network are set randomly. Then the agent improves its performance from its experiences: for each batch of (s, a, r, s') tuples [1], the agent compares Q(s,a,w) with the bootstrapped value computed using actual reward and next state applied to the bellman expectation equation, namely `r +  * Q(s', a, w-)`. The gradient of the difference of these values with respect the the weights is used to update the parameters the reduce the difference. The resulting update rule is: `Δw =  * (r +  * Q(s', a, w-) - Q(s,a,w)) *  w Q(s,a,w)`.

To minimize oscillation and divergence (which DQN is prone to), the agent also utilizes experience replay and fixed Q targets.

With experience replay, the agent does not immediately use its experiences for learning, but first stores the experiences in a replay buffer. After every T time steps (in this case 4), the agent draws batches of experiences (S, A, R, S' tuples) of size 64 from the replay buffer, and performs the

learning step mentioned above. This reduces any sequential correlation between experiences that can skew the approximations toward an invalid direction.

The agent also implements "fixed" Q targets (actually slow moving Q targets) to minimize unnecessary oscillations that occur when the same network is used for the TD targets and the updates. To do this, a separate target network is maintained in addition to the main network to compute TD targets, and this network's parameters are updated very slowly - only .001 the amount of the main network).

[1] s: current state, a: action chosen at s, r: reward obtained by taking action a at state s, s': state resulting from taking action a at state s

1.2.1 Summary of hyperparameters

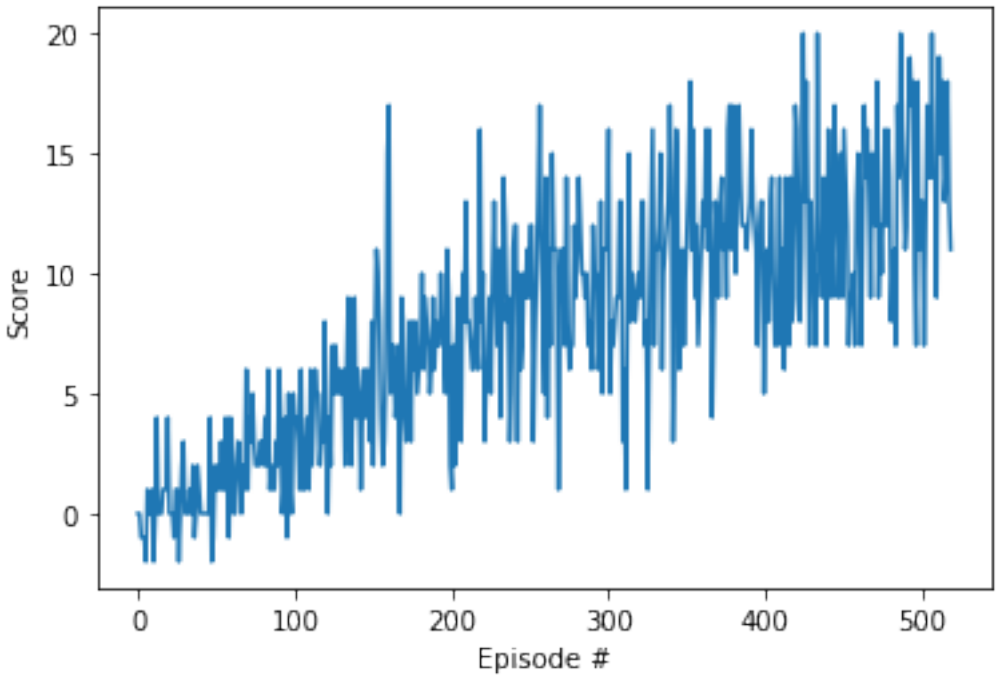
- (discount rate): .99
- (learning rate): 5e-4
- (soft update rate for target network): .001
- Batch size: 64
- Network updates every 4 steps

The hyperparameters were based on experimentation. They are based on the values from the DQN lesson code, and were not changed as the agent successfully learned in relatively few episodes. Cross validation can be used to tune more effective hyperparameters that may allow the agent to learn even faster.

1.2.2 Model architecture

The agent's model is a simple deep neural network consisting of two fully connected layers with 64 units each and relu activation. This architecture was also taken from the DQN lesson code, which was able to solve the moon landing environment successfully. Again, to improve agent performance, cross-validation can be used to compare this architecture with other ones to potentially improve learning performance.

1.3 Plot of rewards



The environment was solved in 519 episodes with an average score of 13.00.

1.4 Ideas for future work

There are many different ways the current implementation can be improved: - Prioritized experience replay can be used to assign a priority to each experience in the replay buffer based on TD error. This will allow more important experiences to be utilized more frequently. - Dueling networks can be used to learn the state values in one network and the advantage values in another, with the networks able to share some layers. This has been shown to lead to better policy evaluation when many actions have similar values, such as in this environment. - A combination of these methods can be used, also known as a rainbow method. Other extensions can also be incorporated, such as multi-step bootstrap targets, distributional DQN, and noisy DQN.