Deep Reinforcement Learning Navigation Project Report

Larry Miller

Overview

This document is the report for the navigation project for Udacity Deep Reinforcement Learning class. The implementation of the trained agent and the results that were achieved are summarized.

Learning Algorithm

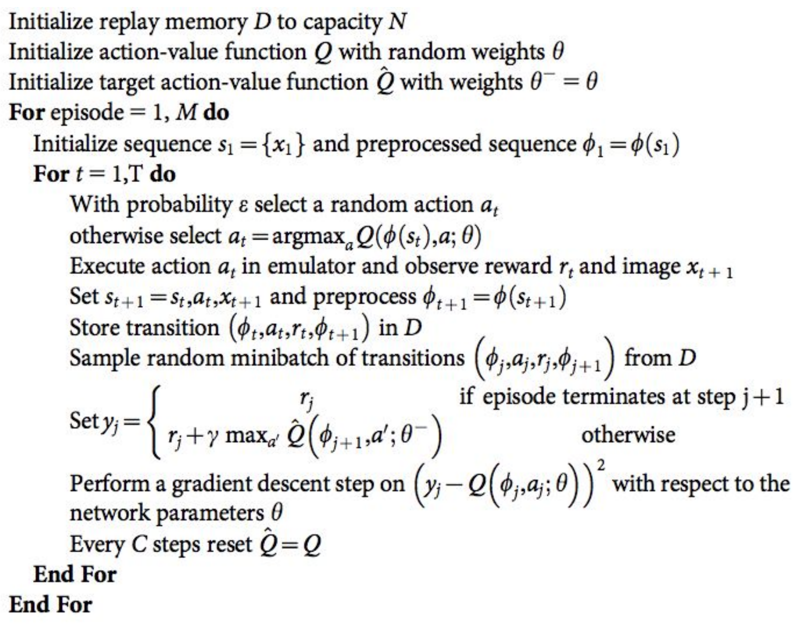
My algorithm uses a Deep Q Network learner to accomplish the task of collecting yellow bananas in the environment while avoiding blue bananas. As suggested in the project instructions, I adapted my deep Q learner from the lunar lander exercise to run in this environment. Since the number of size of the state and the number of actions are specified at the time of instantiation of the agent, there were no changes needed to run this agent in this environment.

The implementation consists of an agent which performs epsilon-greedy actions in the environment, stores its experiences in a replay buffer, and trains a Q network based on random batches of experiences after taking a designated number of actions.

The Q network is modeled by a neural network with 5 layers:

- A fully-connected input layer with 37 inputs (which is the size of the state vector) and 64 outputs
- A ReLU activation layer
- A fully-connected hidden layer with 64 inputs and 64 outputs
- Another ReLU activation layer
- A fully-connected output layer with 64 inputs and 4 outputs

The learning algorithm is based on the algorithm provided in the one provided in the DQN paper that was included in the class materials: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf. The pseudo code for the algorithm is as follows:



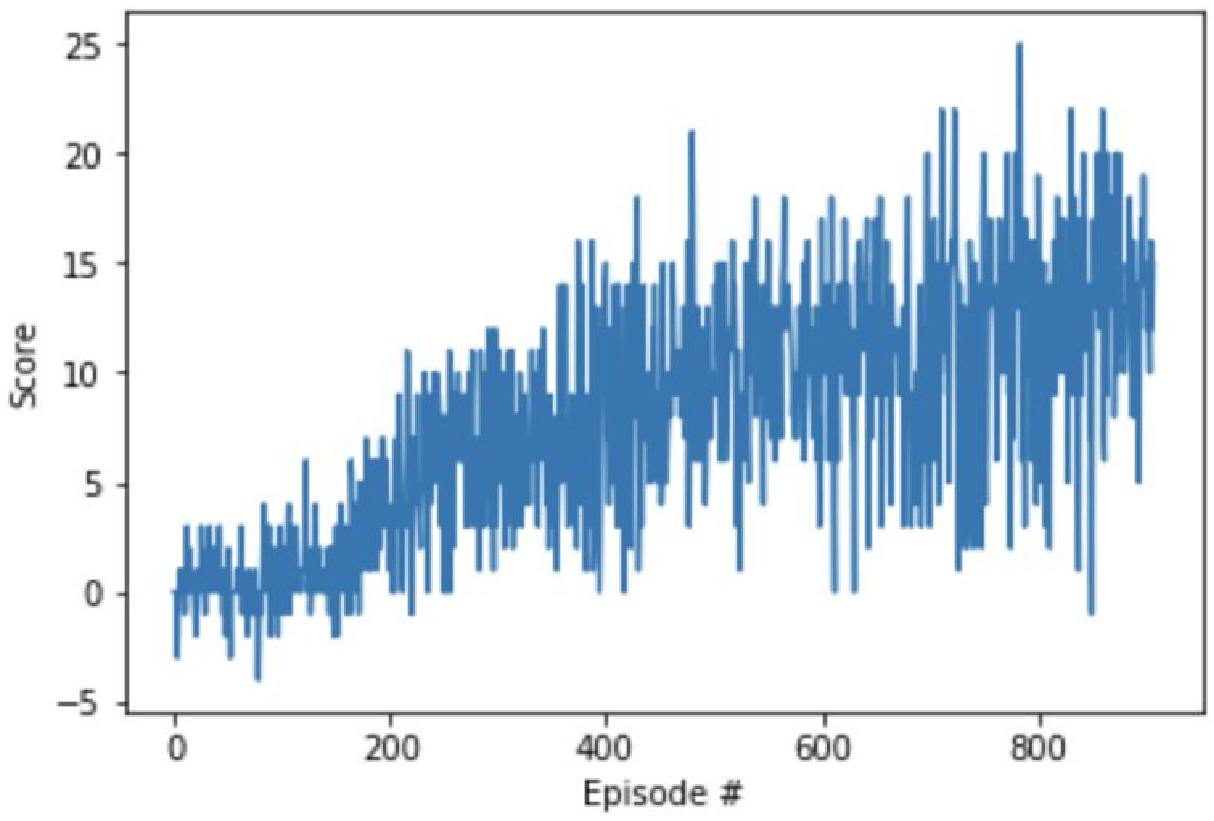
```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

There are several hyperparameters associated with my my implementation. The same values that were chosen in the lunar lander exercise worked well for this task. They are listed in the following table:

Hyperparameter Name	Value	Description
BUFFER_SIZE	10000	Replay buffer size
BATCH_SIZE	64	Experiences per batch
GAMMA	0.99	Discount factor for future rewards
TAU	0.001	Parameter update factor
LR	.0005	Learning rate for optimizer
UPDATE_EVERY	4	Frequency of neural net update
eps	1	Initial value of epsilon
eps_end	0.01	Final/minimum value of epsilon
eps_decay	0.995	Geometric decay rate for epsilon

Results

The implemented learner was able to solve the environment in 807 episodes. A plot of the reward for each episode during training is shown below.



Future Work

There are several improvements that can be made to my implementation. The first improvement would be to improve the performance of my learning algorithm by calculating the loss and optimizing the network weights for an entire batch in one step rather than one experience at a time. I suspect this would be an improvement because when I compared my learner to the solution implementation in the lunar lander exercise, my learner took more episodes to complete and took more time to train. The most significant difference between my implementation and the exercise solution is the way the gradient is calculated.

Another improvement would be to optimize the hyperparameters and Q network structure for this environment. It is likely that performance can be improved by choosing different values for the parameters listed in the previous section and it is possible that a different design for the neural network could improve the agent as well.

Finally, I could improve my deep Q learner by implementing the improvements presented in class which include double DQN, dueling DQN, and prioritized experience replay.