Udacity Deep Reinforcement Learning

Project 1 - Navigation - Report

Learning Algorithm

This project uses a DQN network, as first described by this DQN paper. The idea is basically to modify the standard Q-Learning approach but use a deep neural network to-do the approximation of the optimal action-value function. However reinforcement learning has been shown to be unstable when using neural networks. The authors overcame these limitations by using experience replay (to de-sequence the observations) , and using an iterative update.

Taking their algorithm and adapting for my needs, the pseudo like code for this is :

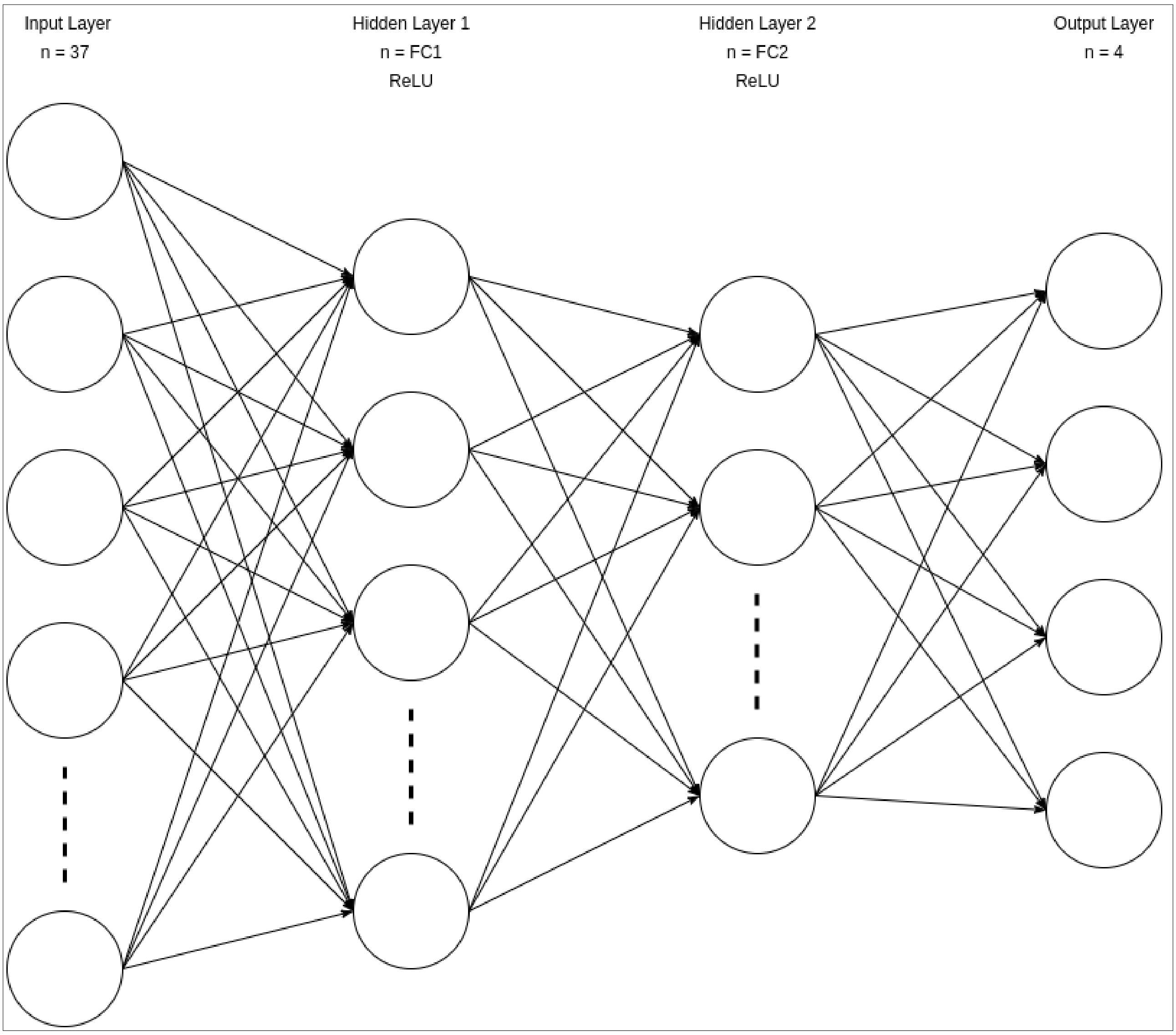
```
Initialize a replay memory replay_buffer to capacity REPLAY_BUFFER_SIZE
Initialize action-value network Q1 with random weights.
Initialize target action-value network Q2 with Q1 weights

SET ε = EPS_START

FOR episode in [1 , max_episodes DO

    reset environment to initial conditions and get starting state
    score = 0
    WHILE True

        IF ~U(0,1) > ε or score > 1 THEN
            action = argmax( Q1( state ))
        ELSE
            action = pick a random action

        perform action and get observation next_state , reward
        score += reward
        store observation in replay_buffer

        AFTER EVERY UPDATE_EVERY observations
            get MINIBATCH_SIZES samples randomly from replay_buffer
            FOR ALL j
                y_j = reward_j+GAMMA*max(Q2(next_state_j))*(1–done_j)
                e_j = Q1( state_j )[action_j]
            back propagate Q1 with mse error ( e_j - y_j )
            use adam optimizer with a learning rate = ALPHA
            transfer TAU proportion of Q1 weights to Q2 weights
        IF Done
            BREAK

        state = next_state
        ε = max( EPS_END , ε * EPS_DECAY )
```

Q1 is the local neural network , Q2 is the target neural network . Parameters in **RED** are hyper parameters. Q(state) evaluates the neural network with given state input , and returns a vector of action values.

We have added an extra condition for choosing weather to use the Q1 action or pure random chance, We wont use the random choice if the score is currently 2 or more , this empirically seems to help speed up training a little (less episodes) . Not sure if this a known technique or not.

Hyper Parameters & Model

Working out what hyper-parameter values to use , and what model of neural network to use is always a challenge.



So we fixed our neural network model (code in ***qnet.py***) , to have only two hidden layers of adjustable size (FC1 and FC2). The outputs use **ReLU** activation , with the inputs transformation being a simple linear activation of the form $\mathbf{y = x \cdot W^t + b}$.

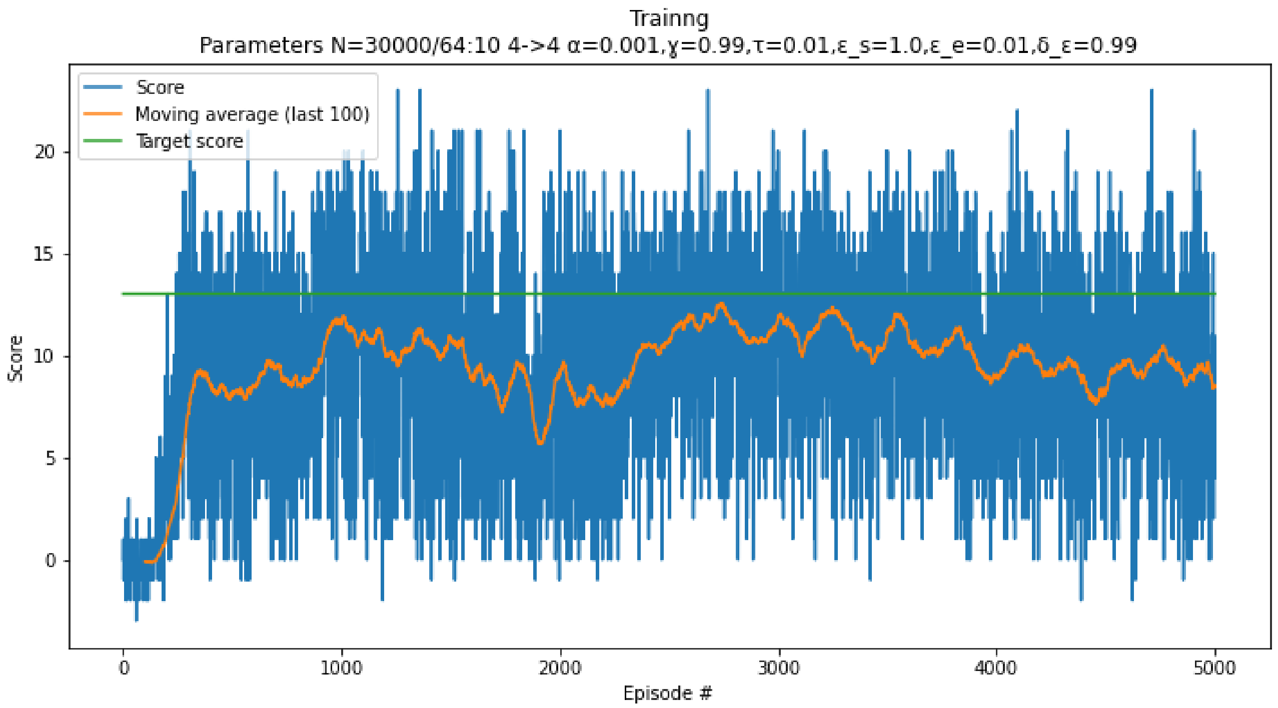
We therefore could just concentrate on the parameters, which are briefly described here

FC1 , FC2 - size of each of the hidden layers respectively
REPLAY_BUFFER_SIZE – how many samples to keep in memory at one time
MINIBATCH_SIZE – how many samples to use for training
UPDATE_EVERY – after how many samples should we train the Q1 network.
GAMMA – discount factor
ALPHA – learning rate for the Adam optimizer
TAU – Weight transfer ratio of Q1 into Q2.
EPS_START , EPS_END , EPS_DECAY – epsilon greedy start/end and decay rates

We started by setting our hyper-parameters as such.

FC1 = 4, FC2 = 4, REPLAY_BUFFER_SIZE = 30000, MINIBATCH_SIZE = 64, UPDATE_EVERY = 10, GAMMA = 0.99, ALPHA = 0.001,TAU = 0.01, EPS_START = 1.0,EPS_END = 0.01,EPS_DECAY = 0.99

This was deliberately small 4x 4 network , as it is always best to start small (consider the final product – smaller is easier / cheaper to embed into another system).



It actually did OK, it achieved a goal of around 11.5 at around 1000 , and nearly hit 12.0 after 3000 episodes, but was slowly deteriorating. So we increased the size of both hidden layers to 30. At this point we noticed it takes just as long to train, so the bottle neck is the agent simulator.

One problem that did crop up , is that although the model was trained to get an average score off 13.0 or more over the last 100 episodes. If you used that model, and run it for

100 more episodes it could consistently get a score a lot less than 13.0. This is because the early scores may have been very high , but the model has started to over fit , and actually getting worse and worse scores , but the average score was enough to be considered solved. So when this happens just tweaked the parameters and retrained.

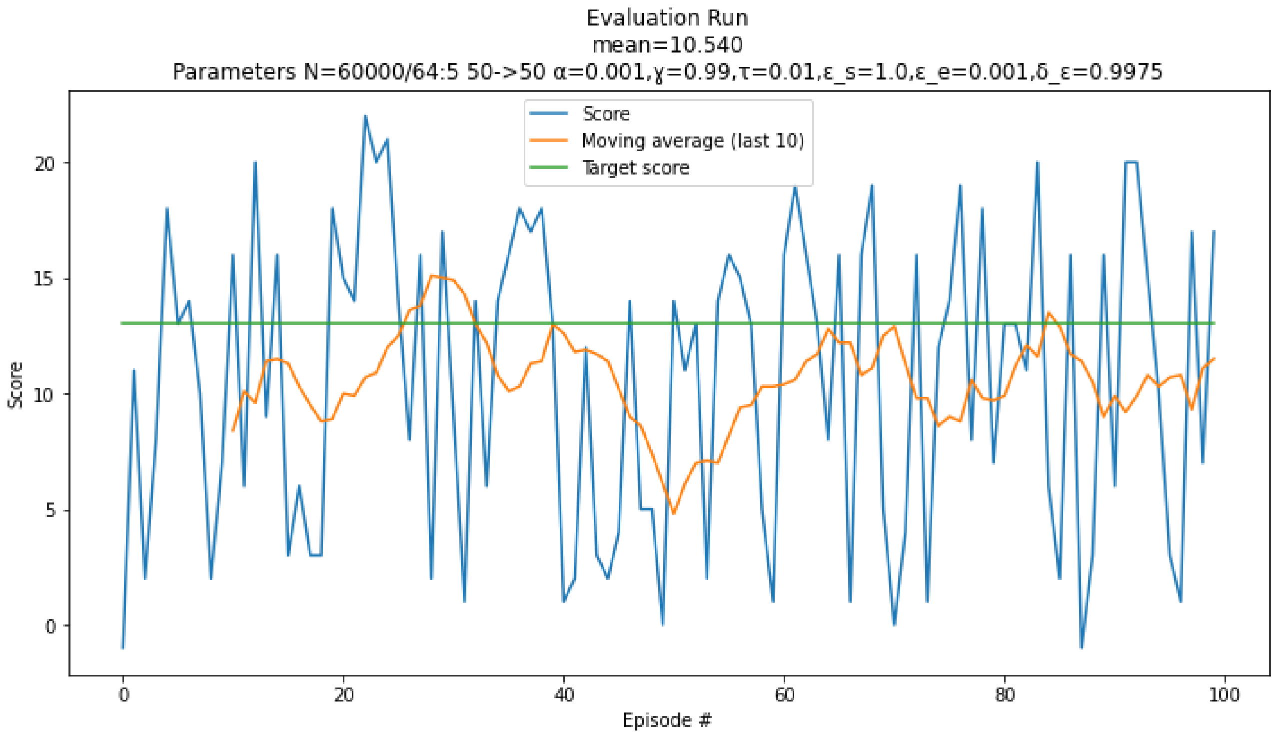


Figure 2: Actual performance is worse, average only 10.5

Plot of Rewards (Final)

Eventually settled for these parameters, can be solved in a lot less cycles , but this seems to give a more stable solution (fewer zero scores , less getting stuck when surrounded by blue bananas – even when there is an obvious way out.)

FC1 = 18
FC2 = 18
REPLAY_BUFFER_SIZE = 60000
MINIBATCH_SIZE = 64
UPDATE_EVERY = 6
GAMMA = 0.99
ALPHA = 0.001
TAU = 0.01
EPS_START = 1.0
EPS_END   = 0.01
EPS_DECAY = 0.998

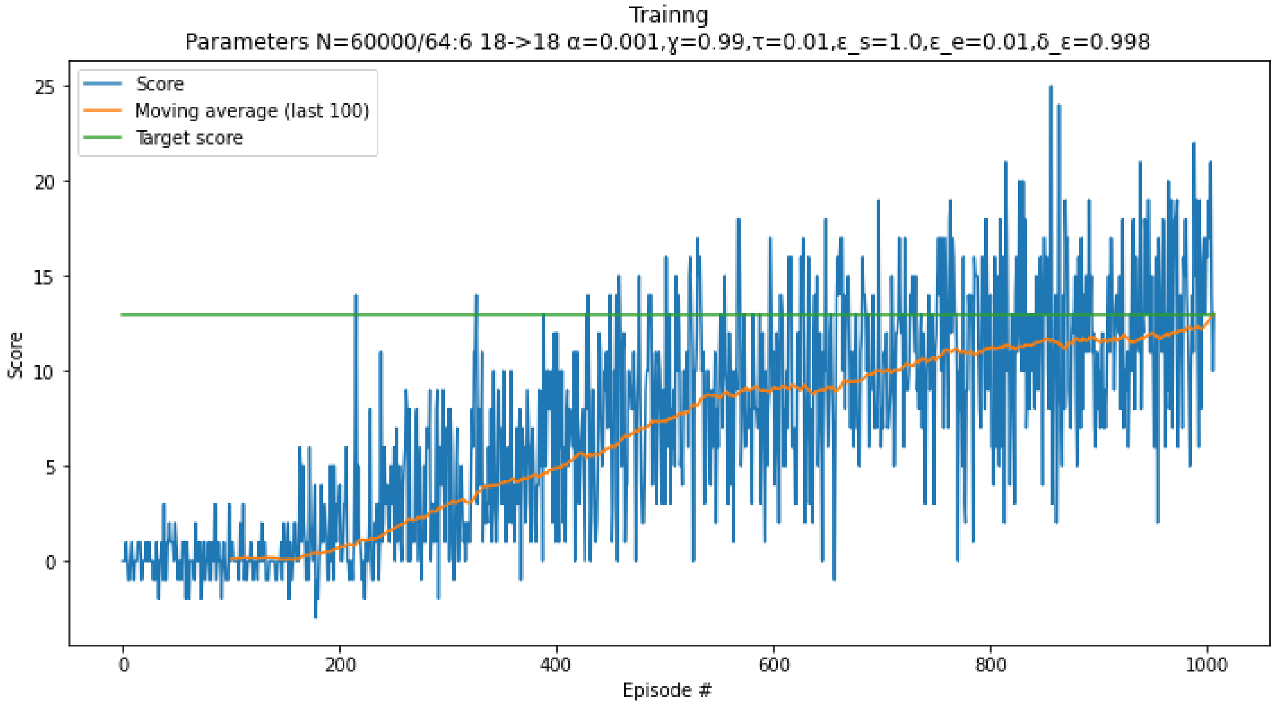


Figure 3: Solved after 907 episodes

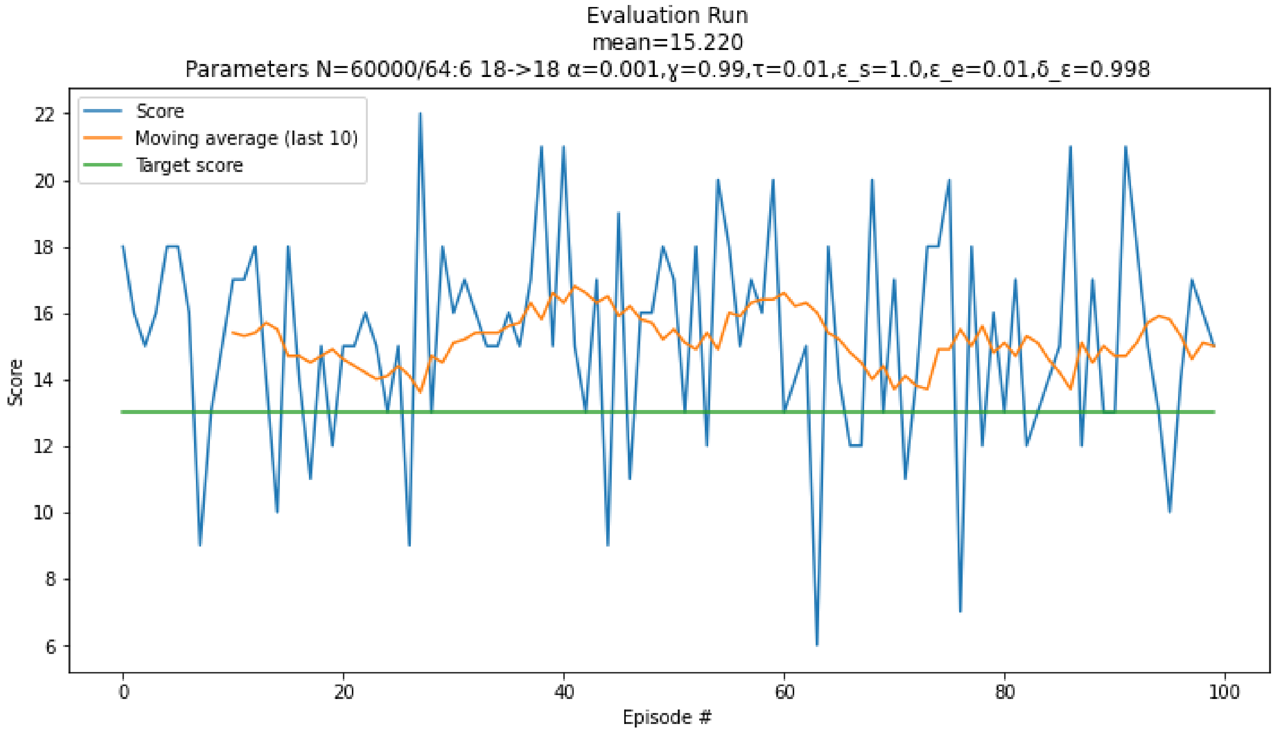


Figure 4: Validation run - showing passes average score condition , not perfect but minimum 6.

A video , of the trained agent can be found below

https://www.youtube.com/watch?v=ou-iFp0bhrs

The model weights are saved in the file “model.pt” in this repository.

Ideas for Future Work

Training performance can be increased by more tweaking, for instance increased units in each hidden layer – although having more will mean actual running performance will be worse. Improvement in the decaying of ε , seems to be a good choice , so using a more sophisticate algorithm like Boltzmann exploration may yield results. Also continuing to investigate my amendment of adding a score > 1 condition.

Other areas that could be investigated are the use of Double DQN (would require minimal code changes) and also Duelling DQN and even the Rainbow technique.