Udacity Nanodegree

Deep Reinforcement Learning Course

14. April 2020

Project 1: Banana Navigator

Solution Report

This report summarizes the solution for the Banana Navigator project - the first project of the Deep Reinforcement Learning Nanodegree from Udacity. The first section introduces the problem and what should be solved. The second section explains the model and the parameters used to solve this. The final section concludes the approach and proposes further steps for potential improvements.

Problem Description

The problem consists of training an agent able to pick up yellow bananas and avoid the blue bananas. The agent resides in the Unity Machine Learning environment. The state space consists of 37 dimensions and contains the agent's velocity and ray-based perception of the objects around its forward direction. The agent can take one of the following actions: 0 (move forward), 1 (move backward), 2 (turn left) and 3 (turn right). A reward of +1 is given when the agent collects a yellow banana and -1 for collecting a blue banana. The task is episodic and is considered solved when the agent scores average reward of +13 over 100 consecutive episodes.

Model

For the solution of this problem I used the Deep Q-Learning approach. The neural network model consists of two fully connected layers (size of 256 and 512 respectively), with input size of 37 (equals to the state space) and output of 4 (the number of possible actions). To avoid overfitting, I introduced a Dropout layer after each FC layer.

```
def QNet(state_size=state_size, action_size=action_size, seed=10, fc1=256, fc2=512):
    torch.manual_seed(seed)
    return nn.Sequential(OrderedDict([
        ('fc1', nn.Linear(state_size, fc1)),
        ('relu1', nn.ReLU()),
        ('dropout1', nn.Dropout(p=0.25)),
        ('fc2', nn.Linear(fc1, fc2)),
        ('relu2', nn.ReLU()),
        ('dropout2', nn.Dropout(p=0.25)),

        ('output', nn.Linear(fc2, action_size))
    ]))
```

I also tested the model with three hidden layers and the improvement was not significant. Therefore I decided to go with the simpler architecture. As an activation function I used the rectified linear unit. For the hyper parameters, the following values were used:

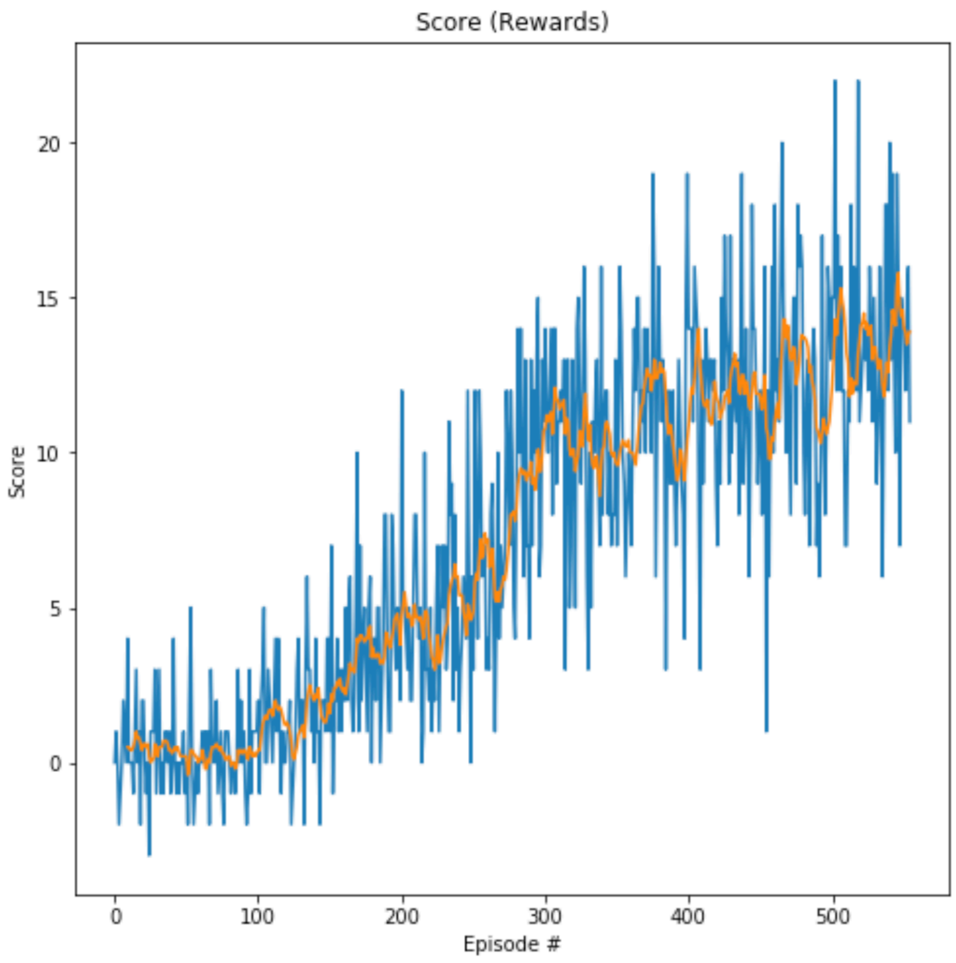
```
BUFFER_SIZE = 100000
BATCH_SIZE = 64
GAMMA = 0.995
TAU = 1e-3
LR = 0.001
UPDATE_EVERY = 4 # how often to update the network
```

The experience replay buffer size was initialized to the value of 10^5. The networks was trained using batches of size of 64 with the Adam optimization algorithm. For the discontinuation of the future rewards (Q-learning), I chose the value of 0.995. The target Q-network is being updated every 4 timesteps using the soft-update technique with tau equals to $0{,}001$. To balance between exploration and exploitation, the epsilon value is decayed from 1.0 to 0.01 with decaying factor of 0.995. The training of the DQN took less the 500 episodes:

We also plot the reward function: the blue line is the score for each episode. Since the

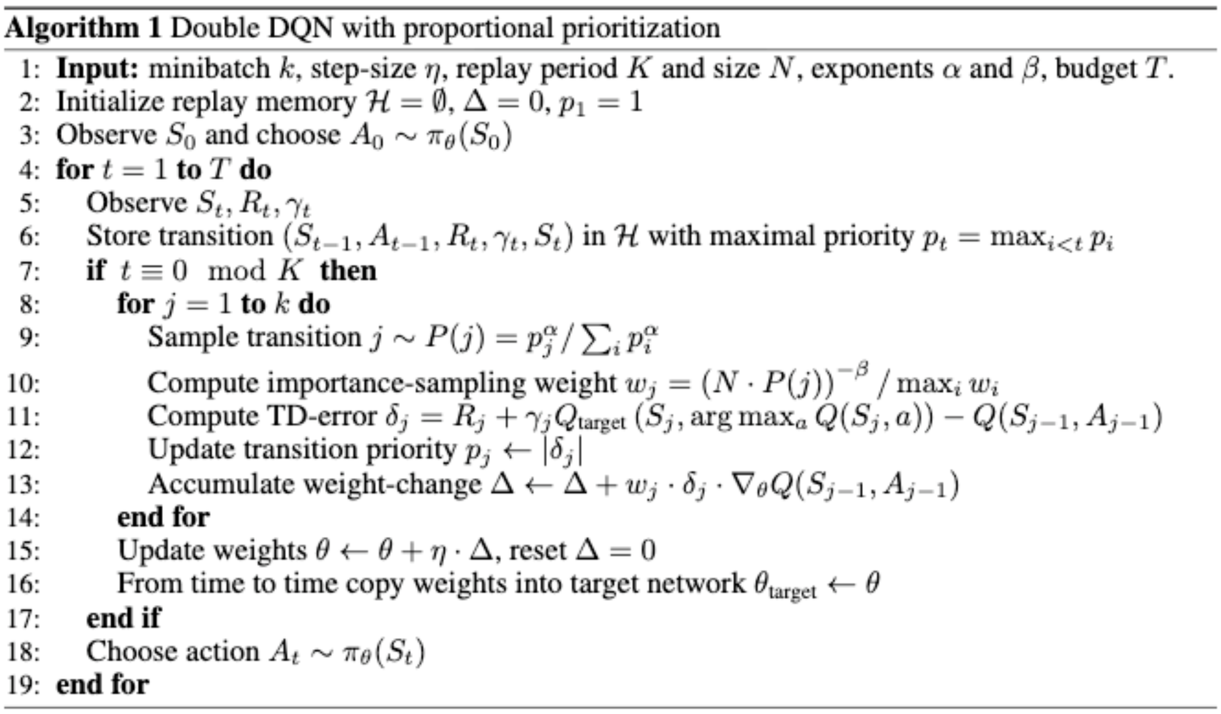
```
Episode 100        Average Score: 0.32
Episode 200        Average Score: 2.55
Episode 300        Average Score: 6.44
Episode 400        Average Score: 10.77
Episode 500        Average Score: 12.10
Episode 555        Average Score: 13.09
Environment solved in 456 episodes!        Average Score: 13.09
```

reward-episode function (blue) is really noisy, we averaged over 10 consecutive rewards and plot the mean function (orange).



Conclusion

The DQN simple architecture was able to overcome this problem in a relatively short time. The state space was small so the current implementation of the DQN seems like a good approach. One possibility that can be considered is to add prioritization to the experience replay queue. This means that also the TD-error needs to be calculated and used as a metric for priority. The implementation (obtained from this paper) is given below:



Algorithm 1 Double DQN with proportional prioritization

```
Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ to $T$ do
   Observe $S_t, R_t, \gamma_t$
   Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
   if $t \equiv 0 \mod K$ then
      for $j = 1$ to $k$ do
         Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
         Compute importance-sampling weight $w_j = \left(N \cdot P(j)\right)^{-\beta} / \max_i w_i$
         Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}\left(S_j, \arg\max_a Q(S_j, a)\right) - Q(S_{j-1}, A_{j-1})$
         Update transition priority $p_j \leftarrow |\delta_j|$
         Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
      end for
      Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
      From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
   end if
   Choose action $A_t \sim \pi_\theta(S_t)$
end for
```

For the optional assignment - training an agent on raw pixel data one needs to come up with more complex - a convolution based one.