Udacity Deep Reinforcement Learning p1-navigation using PyTorch and Unity ML-Agents

Huihui Yang

1. Introduction of the project

For this project, I trained an agent to navigate (and collect bananas!) in a large, square world.

The agent is trained from vector input data (not pixel input data). The state space has 37 dimensions and contains:

- the agent's velocity
- ray-based perception of objects around agent's forward direction.

The agent receives the following rewards:

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

Given this information, the agent has to learn how to best select actions. Four discrete actions are available.

At each time step, the agent can perform four possible actions:

- `0` - walk forward
- `1` - walk backward
- `2` - turn left
- `3` - turn right

2. Algorithms: Deep Q-Network (DQN) Reinforcement Learning

Note that, all the bellowing information is from Minh et al., 2015.

DQN reinforcement learning (RL) is an extension of Q Learning and is an RL algorithm intended for tasks in which an agent learns an optimal (or near optimal) behavioral policy by interacting with an environment. Via sequences of state (s) observations, actions (a) and rewards (r) it employs a (deep) neural network to learn to approximate the state-action function (also known as a Q-function) that maximizes the agent's future (expected) cumulative reward. More specifically, it uses a neural network to approximate the optimal action value function

$$Q^*(s,a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma r_{t+2} + \ldots | s = s, a = a, \pi],$$

where $Q*$ is the maximum sum of expected rewards; r_t discounted at each time step, t, by factor γ, based on taking action, a, given state observation, s, and the behavioural policy $\pi = P(a|s)$.

When using a nonlinear function approximator, like a neural network, reinforcement learning tends to be unstable. The DQN algorithm adds two key features to the Q-learning process to overcomes this issues.

1). The inclusion of a replay memory that stores experiences, et = (st, at, rt, st+1), at each time step, t, in a data set Dt = {e1, e2, … et}. During learning, minibatches of experiences, U(D), are randomly sampled in order to update (i.e., train) the Q-network. The random sampling of the pooled set of experiences removes correlations between observed experiences, thereby smoothing over changes in the distribution of experiences in order to avoid the agent getting stuck in local minimum or oscillating or diverging from the optimal policy. The use of a replay memory also means that experiences have the potential to be used in many weight updates, allowing for greater data efficacy. Note, for practical Michael Richardson, 2018 Project for Udacity Nanodegree in Deep Reinforcement Learning (DRL) purposes the replay memory is fixed in size and only stores the N experiences tuples. That is, it does not store the entire history of experience, only the last N experiences.

2). The use of a second target network for generating the Q-learning targets employed for Q-network updates. This target network is only updated periodically, in contrast to the action-value Q-network that is updated at each time step. More specifically, the Q-learning update at each iteration i uses the following loss function,

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a';\ \theta_i^-) - Q(s, a;\ \theta_i)\right)^2\right]$$

θ_i and θ_i^- are the parameters of the Q-network and the Q-target-network at iteration i, respectively, with the target network parameters θ_i^- updated with the Q-network parameters θ_i every C steps (Min et al., 2015). Updating the target network periodically essentially adds a delay between the time an update to Q is made and the time the update affects the Q targets employed for network training, thereby further stabilizing learning by reducing the possibility of learing oscillations. Note that in the implementation detailed here, the parameters of the target network θ_i^- are updated to equal the Q-network parameters θ_i incrementally, over C steps, rather than all at once after C steps. This subtle difference, however, has no operational effect on the performance of DQN.

The pseudo-code for the DQN algorithm is provided below. Note that the code provided in this repository includes extensive comments detailing the DQN learning/training algorithm and how it is implemented using python and PyTorch:

Algorithm for Deep Q-learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q^- with weights $\theta^- = \theta$
For episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and pre-processed sequence $\phi_1 = \phi(s_1)$
    For t = 1, T **do**
        With probability ε select a random action a_t
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_1), a; \theta)$
        Execute action a_t and observe reward r_t and state x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_{t+1})$
        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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} Q^-(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

        Perform a gradient descent step on $\left(y_j - Q(\phi_j, a; \theta)\right)^2$ with respect to the network parameters θ
        Every C steps reset $Q^- = Q$
    End For
End For

Adapted from: Mnih et al., (2015). Human-level control through deep-reinforcement learning. *Nature*, 518.

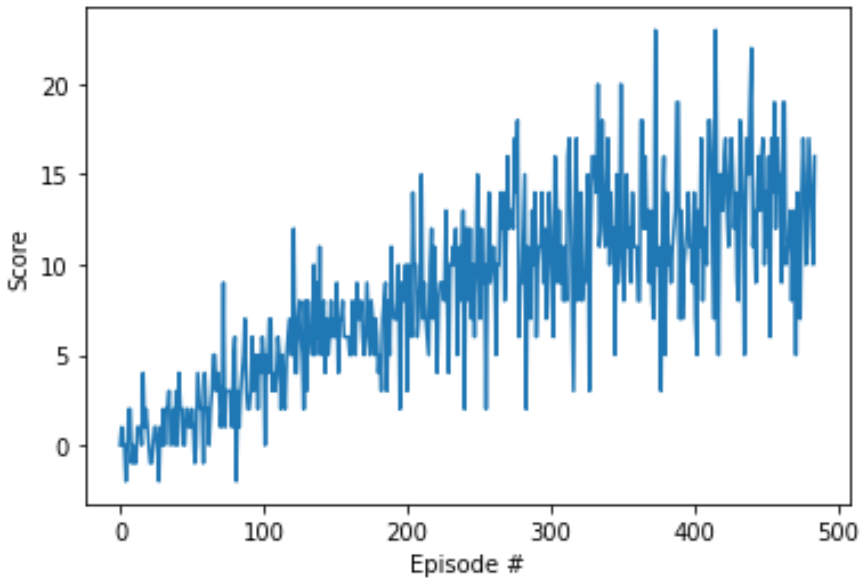
3. Building & Training

I built a DQN with 2 fully-connected (FC) layers, with experienced 64 nodes each one. The size of the input layer was equal to the dimension of the state size (i.e., 37 nodes) and the size of the output layer was equal to dimension of the action size (i.e., 4).

hyperparameter and training settings:

- n_episodes = 1000
- max_t = 2000
- eps_start = 1.0
- eps_end = 0.1
- eps_decay = 0.995

Using the recommended hyperparameter and training settings detailed above, the agent is able to "solve" the banana environment (i.e., reach of average score of >13 over 100 episodes) in less than 1000 episodes. A successful trained prototypical example of DQN-Agent training performance is illustrated in the below figure.



4. Ideas for Future Work

My current agent can solve the environment by running 385 episodes to get the average scour of 13, which is very fast.

It will be good to try other algorithms to reach the target. I see some people have already tried

- Deep Q-Network
- Double Deep Q-Network
- Dueling Q-Network
- Prioritized Experience Replay

I would like to try them too.