Solving Navigation “BananaBrain” Environment using DQN

The algorithm I use is largely based on the DQN algorithm provided by the Udacity code example for solving the **OpenAI Gym's LunarLander.** The code is very much readable and provides a clear framework for further fine-tuning to solve other problems using DQN. The link to the code as below:

https://github.com/udacity/deep-reinforcement-learning/blob/master/dqn/solution

In this report, I will have a brief review on the DQN paper (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf), discuss the model architecture of the implementation and the process of the training and reaching the final solution.

Deep Q-learning (DQN)

In the DQN paper, the author presented a single model-free and off-policy algorithm using deep function approximators that would be able to develop a wide range of competencies on a varied angle of challenging tasks.

The authors consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function, which is the maximum sum of rewards r_t , discounted by γ at each timestep t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a)

Key elements in the DQN algorithm:

Uses function approximator to estimate the action-value function, $Q(s,a;\theta) = Q^*(s,a)$

The target value method, using parameters θ_i^- from some previous iteration, effectively weights θ will only be updated every N step, by resetting $\theta_i^- = \theta_{i-1}$

A model-free method, without estimating the reward and transition dynamics P(r,s’|s,a).

An off-policy method, as it learns about the greedy policy $a = \text{argmax}_{a'} Q(s,a'; \theta)$, while following a behavior distribution that ensures adequate exploration of the state space, for which the ε–greedy policy is often used.

Replay buffer: a finite sized cache storing *(st, at, rt, st+1).* When the replay buffer was full the oldest samples were discarded

Soft target updates: using a separate network for generating the targets y_j in the Q-learning update. The weights of these target networks are then updated by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau << 1$

Error-clipping: clip the error term from the updated $r + \max_{a'} Q(s',a'; \theta_i^-) - Q(s,a; \theta_i)$ to be between -1 and 1

Below is the pseudo code for the DQN algorithm:

Algorithm 1: 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 $\hat{Q}$ 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 ε 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 $\left(y_j - Q(\phi_j,a_j;\theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Model Architecture

I have followed the suggested benchmark implementation to adjust the model architecture. The original model architecture was the same to the DQN algorithm in the Udacity repo. The network is a neural network with 2 full connected layers with size of 64 and with Relu.

Hyperparameters

Below is the final hyperparameters I used to for getting the solution:

BUFFER_SIZE = int(1e5) # replay buffer size

BATCH_SIZE = 64 # minibatch size

GAMMA = 0.99 # discount factor

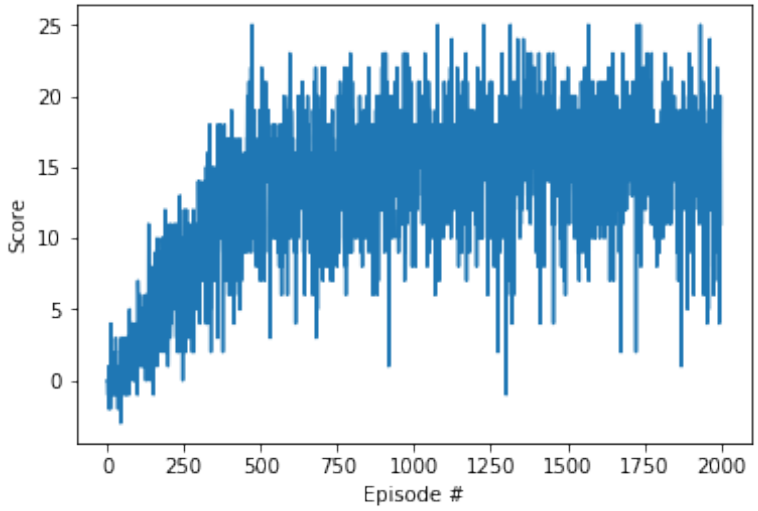
TAU = 1e-3 # for soft update of target parameters

LR = 5e-4 # learning rate of the actor

UPDATE_EVERY = 4 # update the target network every N timesteps

Model training

The model training was actually pretty straightforward as the model architecture also makes sense for solving the Navigation problem. It was solved after about 600 episodes (see below chart) and the reward remains quite stable after that. If I remembered correctly, the whole training took less than an hour.



Future Improvement

I didn't use error clipping in my solution. I think that's something worth trying to see if it needs fewer episodes to solve the task.

Also as suggested by the project instruction, the current agent learned from information such as its velocity, along with ray-based perception of objects around its forward direction. A more challenging task would be to learn directly from pixels. This environment is *almost* identical to the project environment, where the only difference is that the state is an 84 x 84 RGB image, corresponding to the agent's first-person view of the environment.

Reference:

Human-level control through deep reinforcement learning
https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Udacity repo for DQN https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn/solution