Deep Q-Network (DQN) Reinforcement Learning using PyTorch and Unity ML-Agents

A simple example of how to implement vector based DQN using PyTorch and an ML-Agents environment.

The example includes the following DQN related python files:

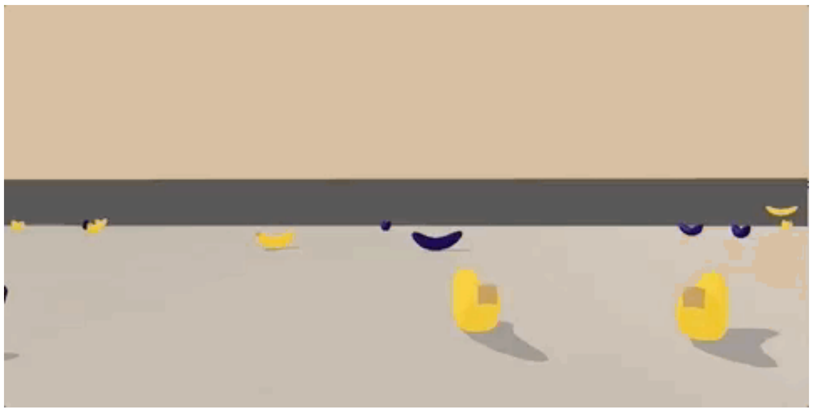
- **dqn_agent.py:** the implementation of a DQN-Agent
- **replay_memory.py:** the implementation of a DQN-Agent 's replay buffer (memory)
- **model.py:** example implementation of neural network for vector based DQN learning using PyTorch
- **train.py:** example of how to initialize and implement the DQN training processes.
- **test.py**: example of how to run (test) DQN-agent using a trained model

The repository also includes Mac/Linux/Windows versions of a simple Unity environment (**Banana**) for testing. This Unity application and testing environment was developed using ML-Agents Beta v0.4. The version of the Banana environment employed for this project was developed for the Udacity Deep Reinforcement Nanodegree course. For more information about this course visit: https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

The files in the …/**python/** directory are the ML-Agents toolkit files and dependencies required to run the Banana environment. For more information about the Unity ML-Agents Toolkit visit: https://github.com/Unity-Technologies/ml-agents

Example Unity Environment - Banana's

The example uses a modified version of the Unity ML-Agents Banana Collection example environment. The environment includes a single agent, who can turn left or right and move forward or backward. The agent's task is to collect yellow bananas (reward of +1) that are scattered around a square game area, while avoiding purple bananas (reward of -1). For the version of Bananas employed here, the environment is considered solved when the average score over the last 100 episodes > 13.



Example of the Agents view of the Banana Environment
(the agents task is to collect yellow bananas and avoid purple bananas)

Michael Richardson, 2018 Project for Udacity Nanodegree in Deep Reinforcement Learning (DRL)

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

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

State Space and Rewards:
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 for collecting a yellow banana
- a reward of -1 is provided for collecting a purple banana.

DQN Reinforcement Learning (*below description adapted 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, $e_t = (s_t, a_t, r_t, s_{t+1})$, at each time step, *t*, in a data set $D_t = \{e_1, e_2, \ldots e_t\}$. 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

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,

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

where 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). Uupdating 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 $\theta$
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 $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_1), a; \theta)$
        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 $\theta$
        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.

Hyperparameters

DQN Agent Parameters

- state_size (int): dimension of each state
- action_size (int): dimension of each action
- replay_memory size (int): size of the replay memory buffer (typically 5e4 to 5e6)
- batch_size (int): size of the memory batch used for model updates (typically 32, 64 or 128)
- gamma (float): parameter for setting the discount value of future rewards (typically .95 to .995)
- learning_rate (float): specifies the rate of model learning (typically 1e-4 to 1e-3))
- target_update (float): specifies the rate at which the target network should be updated.

The banana environment is a relative simple environment and, thu,s standard DQN hyperparameters are sufficient for timely and robust learning. The recommend hyperparameter settings are as follows:

- state_size: **37** (is the non-optional state size employed by the Banana agent)
- action_size: **4** (is the non-optional action size of the Banana agent)
- replay_memory size: **1e5**
- batch_size: **64** (32 is also sufficient)
- gamma: **0.99**
- learning_rate: **1e-3**
- target_update: **2e3**

Training Parameters

- num_episodes (int): maximum number of training episodes
- epsilon (float): starting value of epsilon, for epsilon-greedy action selection
- epsilon_min (float): minimum value of epsilon
- epsilon_decay (float): multiplicative factor (per episode) for decreasing epsilon
- scores_average_window (int): the window size employed for calculating the average score (e.g. 100)
- solved_score (float): the average score over 100 episodes required for the environment to be considered solved (i.e., average score over 100 episodes > 13)

The recommend training settings are as follows:

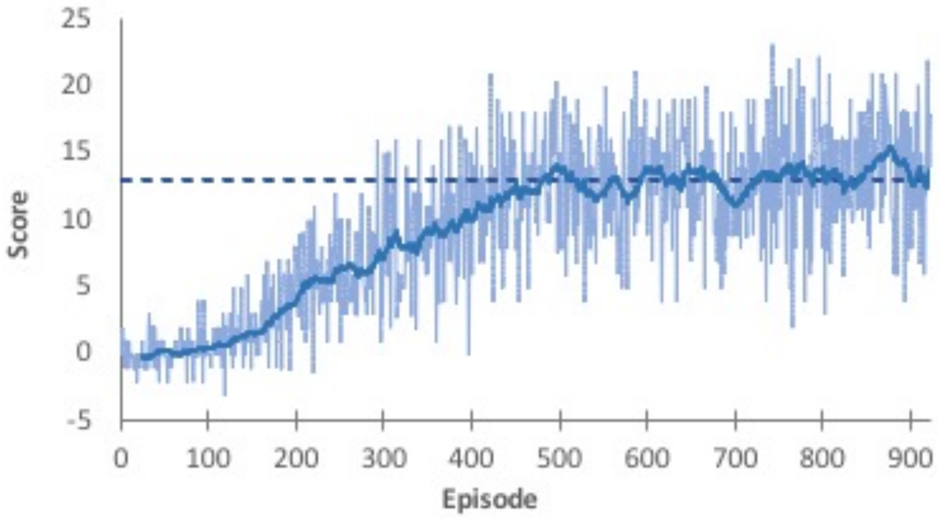
- num_episodes: **2000**
- epsilon (float): **1.0**
- epsilon_min: **0.05**
- epsilon_decay: **0.99**
- scores_average_window: **100**
- solved_score: **14** (better to set a little higher than 13 to ensure robust learning).

Neural Network (Q-network)

Because the agent learnings from vector date (not pixel data), the Q-Network (local and target network) employed here consisted of just 2 hidden, fully connected layers with 128 nodes. 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).

Training Performance

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 500 episodes. The lowest recorded number of episodes required to solve the environment over 25 training runs was 418 episodes). A prototypical example of DQN-Agent training performance is illustrated in the below figure.



Prototypical Example of DQN-Agent Training Performance for the Banana Environment (score per episode)

Future Directions

In the future I plan to implement the following DQN Extensions

- Prioritized experience replay
- Dueling DQN
- Noisy DQN
- Pixel data based training

NOTE: Double DQN has been implemented in the **dqn_agent** code. But agent learning with this implementation is very poor compared to standard (vanilla) DQN.

Reference:

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.