Deep Q-Network (DQN) Reinforcement Learning using PyTorch and Unity ML-Agents over the Banana Hunter environment.

Summary:

This report will debrief around the implementation of the Deep Q-Network RL algorithm, to solve the Unity ML-Agents Banana environment to fulfill the requirements of the Airbus Artificial Intelligence Nanodegree in the optional RL part of this Nanodegree.

The project and associated files have been stored in Github at:

https://github.com/MarioRasconMerinoML/Machine_Learning/upload/main/Reinforcement_Learning/DQN%20Using%20Pytorch%20and%20Unity%20ML%20Agents

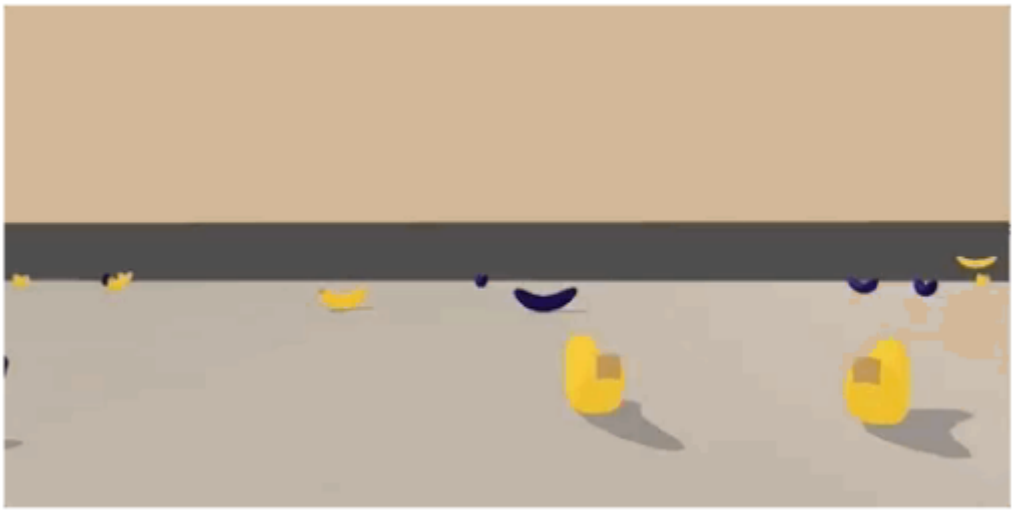
The files included in the submission are:

- dqn_agent.py: the implementation of a DQN-Agent
- model.py: example implementation of neural network for vector based DQN learning using PyTorch
- Navigation.ipynb: example of how to initialize and implement the DQN training processes

The dqn_agent.py and model.py are based on previous exercise around the implementation of a DQN Algorithm on the OpenAI Gym environment. The Navigation file has been adapted in order to account the change in the environment.

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)

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.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

0 - move forward.
1 - move backward.
2 - turn left.
3 - turn right.

Neural Network (Q-network) (model.py) and DQN Agent (dqn_agent.py)

Because the agent learns from vector data (not pixel data), the Q-Network (local and target network) employed here consisted of just 2 hidden, fully connected layers with 128 and 64 nodes respectively.

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 the dimension of the action size (i.e., 4).

In the original DQN Paper (Mhin et al) (Ref.4):

The NN proposed was a Convolutional Neural Network which acted by selecting actions according to the states generated on the different frames coming from the ATARI environment. By preprocessing 84x84x4 frames and analyzing the results of the actions according to the DQN algorithm using a DQN network, experience replay and updated network after C episodes.

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

$$q(s\{t\}, a\{t\}) = R\{t\} + \gamma * R\{t+1\} + \gamma^2 * R\{t+2\} + \gamma^3 * R\{t+3\} + \ldots\ldots$$

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

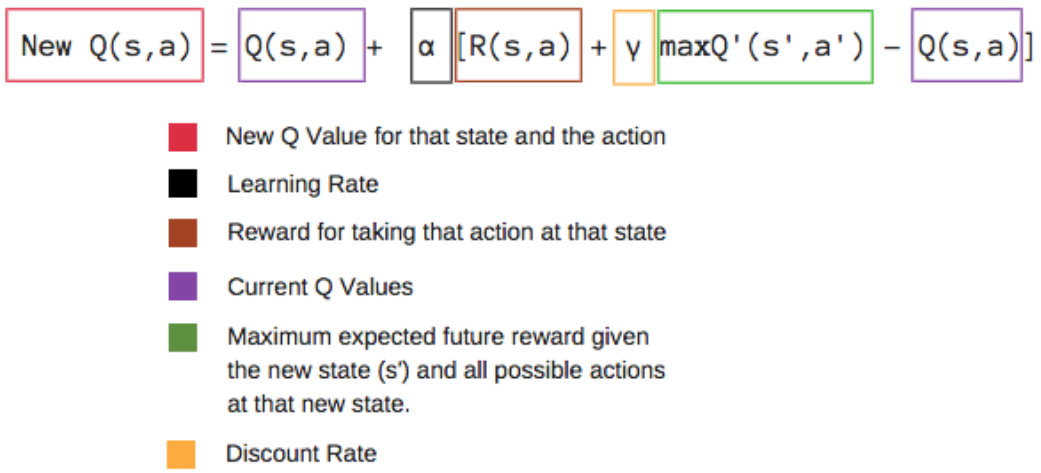
When using a nonlinear function approximator, like a neural network, reinforcement learning tends to be unstable.

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 efficiency.

Note, for practical the code in the DQN Agent obeys to the steps 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. (In the implementation this is achieved by using the "deque" class

The use of a second target network for generating the Q-learning targets employed for Q-network updates. This network has the same architecture as the first network.

This target network is only updated periodically after C steps, 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:



Source: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

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 learning oscillations.

The code taken from Udacity in the previous exercise (Lunar Lander with Open AI), contained as well a modification of the weights update known as "soft target network updates" and was introduced in Lillicrap et al., 2016

Instead of updating weights after a certain number of steps, we will incrementally update the target network after every run using the following formula:

$\theta_{target} = \tau*\theta_{local} + (1 - \tau)*\theta_{target}$ (where $0 < \tau < 1$)

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 $Q^- = Q$

    Update the target networks:

$$\theta' \leftarrow \tau\,\theta + (1 - \tau)\theta' \text{ with } \tau < 1.$$

  End For
End For

Modified DQN Algorithm with Soft-Update

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, thus 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: 5e-4
- target_update: 1e3

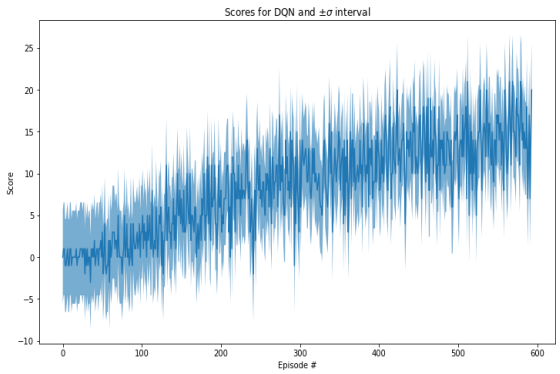
Training Parameters
- num_episodes (int): maximum number of training episodes
- epsilon (float): starting value of epsilon, for epsilon-greedy action selection
- epsilon_min (float): minimum value of epsilon
- epsilon_decay (float): multiplicative factor (per episode) for decreasing epsilon
- scores_average_window (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)

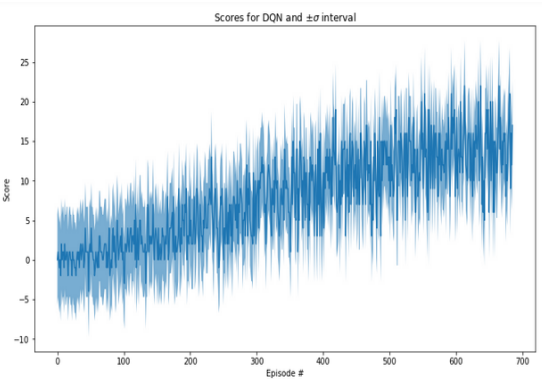
The recommend training settings are as follows:
- num_episodes: 2000
- epsilon (float): 1.0
- epsilon_min: 0.05
- epsilon_decay: 0.99
- scores_average_window: 100
- solved_score: 14 (better to set a little higher than 13 to ensure robust learning

Training Performance

Several combinations of hyperparameters have been attempted mainly batch_size, tau_update, learning_rate. All of them solve the environment after 494 to 600 episodes, only using CPU (no need to use GPU as no CNN architecture is considered)

The plots showing the performance can be seen here below:





Future Directions

In the future It is planned to implement the following DQN Extensions

- Prioritized experience replay
- Dueling DQN
- Noisy DQN
- Pixel data based training (by means of CNN)

References:

1) https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/
2) Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., … & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.
3) https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
4) 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.

https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf