A project to implement vector based DQN using PyTorch and Unity ML-Agent’s environment.

Environment:

This project uses a modified version of 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.

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 (raw image data/pixels are not used to train the DQN model). The state space has 37 dimensions (input to DQN agent is a tensor for 37 entries). The state space has:

* The agent’s velocity.
* Ray-based perception of objects around agent’s forward direction.

The agent receives the following rewards:

* A reward of +1 if agent collects a yellow banana.
* A reward of -1 if agent collects a purple banana.

Intuition of DQN Reinforcement Learning

The problem with traditional Q table approach:

Let's suppose we have an environment where we have 1,000 states and 50 possible actions in each state. In this case, our Q table will consist of 1,000 x 50 = 50,000 rows containing the Q values of all possible state-action pairs. In cases like this, where our environment consists of a large number of states and actions, it will be very expensive to compute the Q values of all possible state-action pairs in an exhaustive fashion.

Instead of computing Q values this way, we can use any non-linear function approximator, such as a neural network. We can parameterize the Q function by a parameter (theta) and compute the Q value where the parameter is just the parameter of our neural network. So, we just feed the state of the environment to a neural network and it will return the Q value of all possible actions in that state. Once we obtain the Q values, then we can select the best action as the one that has the maximum Q value. Since we are using a deep neural network to approximate the Q value, then the deep neural network is called the deep Q network (DQN).

We can denote the Q function by Q(s, a) where the parameter (theta) in subscript indicate that our Q function is parameterized by (theta). We initialize the network parameter (theta) with random values and approximate the Q function (Q-values), but since we initialized (theta) with random values the approximated Q function will not be optimal. So, we train the neural network for several iterations by finding the optimal parameter (theta). Once we find the optimal (theta), we have the optimal Q function. Then we can extract the optimal policy from the optimal Q function.

Input: Current state vector of the agent.

Output: On the output side, unlike a traditional reinforcement learning setup where only one Q value is produced at a time, the Q network is designed to produce a Q value for every possible state-action in a single forward pass.

Handling Instability in the network.

Training such a DNN requires a lot of data, but even then, it is not guaranteed to converge on the optimal value function. In fact, there are situations where the network weights can oscillate or diverge, due to high correlation between action and states.

This can result is a very unstable and inefficient policy we can solve this by:

* Experience Replay
* Fixed Q Target.

Experience Replay:

Some states are pretty rare to come by and some action can be pretty costly, so it would be nice to recall such experiences, and for the same we use a replay buffer.

Replay Buffer

We store each experience tuple (current\_state, action, reward, next\_state, done) in this buffer as we are interacting with the environment and then sample a small batch of tuples from it in order to learn.

As a result, we are able to learn from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Fixed Q Targets:

The use of 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,

The pseudo-code for the DQN algorithm is provided below.

Code:

Deep Neural Network (Q-Network):

As mentioned, the input/vector data is used to learn Q values by the agent, the Q Network (both local and the target network networks) consists of 3 hidden layers, feed forward fully connected with 128, 64, 32 nodes respectively. The size of the input layer is equal to the dimension of the state vector data (in our case it’s 37) and the size of the output layer is equal to the dimension of the action space (in our case it’s 4).

DQN Agent Paramaters:

* state\_size: Total number of dimensions of each state. (This would be the input shape of the tensor input to DNN)
* action\_size: Total number of dimensions of each action (Output shape of the DNN)
* replay\_memory: size of the replay memory buffer for (experience replay)
* batch\_size: size of the memory batch used for model updates.
* gamma: Parameter for setting the discount value for future rewards
* learning\_rate: specifies the rate of the model learning
* target\_update: Specifies the rate at which the target network should be updated.

Below are the values of the hyperparameters used to train the DQN agent to solve the environment/collect at least 13 yellow bananas.

* state\_size: 37
* action\_size: 4
* replay\_memory: 1e5
* batch\_size: 64
* gamma: 0.99
* learning\_rate: 1e-3
* target\_update: 2e-3

Parameters used to train the DQN agent:

* num\_episodes: maximum number of training episodes
* epsilon: starting value of epsilon, for epsilon-greedy selection.
* epsilon\_min: Minimum value of the epsilon.
* Epsilon\_decay: Multiplicative factor (per episode) for reducing the epsilon (to explore or exploit)
* score\_average\_window: the window size employed for calculating the average score (as mentioned in the project description, the task is episodic and in order to solve the environment, the agent must get an average score +13 over 100 episodes.
* required\_score: the average score over 100 episodes required to solve the environment.

Hyperparameter values for training the DQN agent:

* num\_episodes: 2000
* epsilon: 1.0
* epsilon\_min: 0.05
* epsilon\_decay: 0.99
* scores\_average\_window: 100
* required\_score: 14

Training Phase:

Using the above hyperparameters setting, the agent has been able to solve the Unity ML Banana environment (to reach the average score of +13 over 100 episodes) in less than 500 episodes. The lowest recorded number of episodes required to solve the