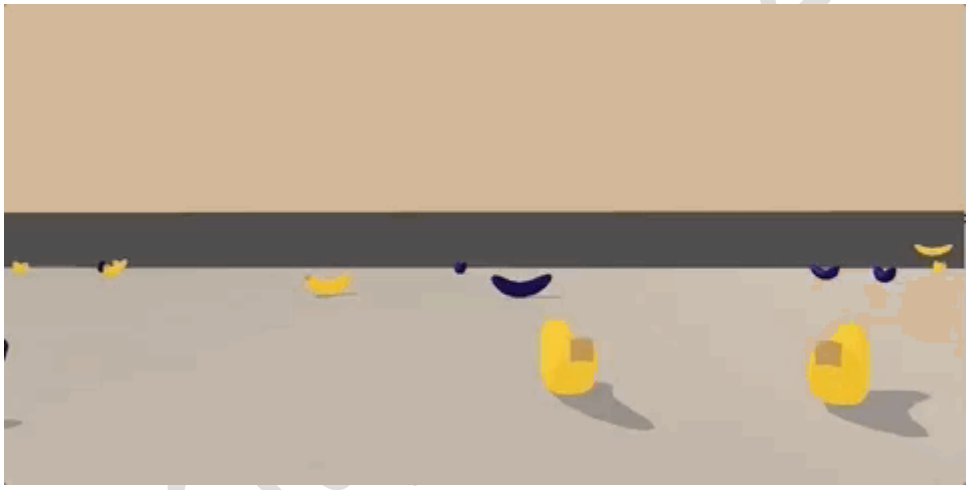
A project to implement vector based DQN using PyTorch and Unity ML-Agent's environment

Understanding 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.



Example view of the agent of the Banana Environment.
(The task is to collect yellow bananas and avoid purple bananas.)

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.

The problem with traditional Q table approach

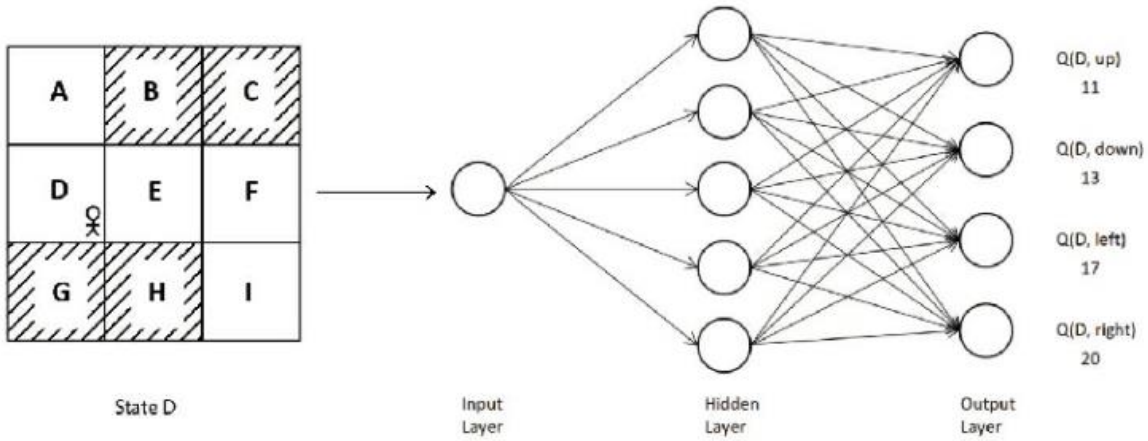
$$Q^*(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

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.

Intuition of DQN Reinforcement Learning

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 $\boldsymbol{\theta}$ 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 $\boldsymbol{Q_\theta}$**(s, a)** where the parameter θ in subscript indicate that our Q function is parameterized by $\boldsymbol{\theta}$**.** We initialize the network parameter $\boldsymbol{\theta}$ with random values and approximate the Q function (Q-values), but since we initialized $\boldsymbol{\theta}$ 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 $\boldsymbol{\theta}$. Once we find the optimal $\boldsymbol{\theta}$,we have the optimal Q function. Then we can extract the optimal policy from the optimal Q function.



The states in the Grid World environment are passed as the inputs to the DQN

DQN Network

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 (parameterized by $\boldsymbol{\theta'}$) 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

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} \underbrace{Q_{\theta'}(s_i', a')}_{\text{Compute using } \theta'} - \underbrace{Q_{\theta}(s_i, a_i)}_{\text{Compute using } \theta})^2$$

The Q value of then next state-action pair in the target is computed by the target network parameterized by θ', and the predicted Q value is computed by our main network parameterized by θ. The squared distance along the batch of inputs gives loss function (to be optimized)

Pseudo-code

Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)
Initialize replay memory D
Initialize action-value function Q with random weight $\boldsymbol{\theta}$
Initialize target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for *episode = 1 to M* **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for *t = 1 to T* **do**
    Following ϵ-greedy policy, select $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \boldsymbol{\theta}) & \text{otherwise} \end{cases}$
    Execute action a_i 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
    `// experience replay`
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ w.r.t. the network parameter $\boldsymbol{\theta}$
    `// periodic update of target network`
    Every C steps reset $\hat{Q} = Q$, i.e., set $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
  end
end

Algorithm 7: Deep Q-Nework (DQN), adapted from Mnih et al. (2015)

The pseudo-code for the DQN algorithm is provided above, adapted from Mnih et al. (2015)

Code

GitHub Repository

Navigation_DQN is the GitHub repository that contains the code, environment and additional scripts that can be cloned to local repository and used to train the agent. However, one can use Navigation.ipynb to quickly experiment with DQN. The repository also includes the links to download the simply Unity Environment (Banana) for testing. This Unity application and testing environment was developed using ML-Agents Beta v0.4. The version of this Banana environment employed for this project was developed by Udacity Deep Reinforcement Learning Nanodegree Team. For more information about this course visit: https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

Installation and get started

The repository contains **/python/** directory, which contains the ML-Agents files and dependencies required to run the Banana Environment. In addition to that, requirements.txt and setup.py script can be used to install the required virtual environment to run, experiment the environment.

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).

Modules to solve the environment (in the Navigation_DQN GitHub Repository):

- ***dqn_agent.py***: Source code to train the agent.
- ***model.py***: Contains the code for QNetwork, to generate Q Values that aids agent to learn the environment. Implemented using PyTorch framework.
- ***replay_buffer.py***: contains the code for DQN Agent's replay buffer.
- ***train_agent.py***: script to train the agent for 2000 episodes.
- ***test_agent.py***: script to test the agent (that loads the trained model's weights) in the environment.
- ***Navigation.ipynb***: The Jupyter notebook that contains the code cells to experiment with the model.py and DQN Agent's performance and plot the score.

DQN Agent Parameters

- ***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

DQN Agent Hyperparameters:

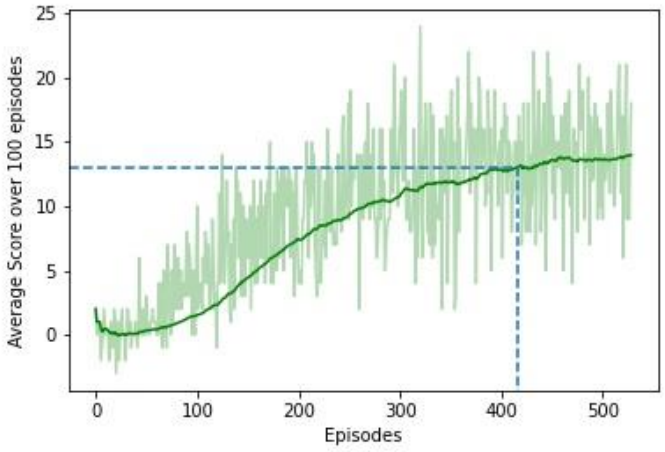
- ***num_episodes:*** maximum number of training episodes
- ***epsilon:*** starting value of epsilon, for epsilon-greedy selection.
- ***epsilon_min:*** Minimum value of the epsilon.
- ***Epsilon_decay:*** Multiplicative factor (per episode) for reducing the epsilon (to explore or exploit)
- ***score_average_window:*** 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.

Below are the values of the hyperparameters used to train 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. However, the least number of episodes required to solve the environment was 400 episodes.



DQN-Agent Training performance for solving Environment Learning (+13 rewards for over 100 episodes). The graph tells us the average score per episodes

Future Work

- Duelling DQN can be explored to if the agent's performance can be improved.
- At the same time, the existing DQN and Double-DQN can be used to train on pixel-based data (raw images as states/inputs to the deep (Convolutional) neural network).

Observation: *Double DQN has been implemented in train_dqn.py script. However, the performance is not optimal when compared to DQN*

References

- https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
- https://unnatsingh.medium.com/deep-q-network-with-pytorch-d1ca6f40bfda
- https://ai.googleblog.com/2015/02/from-pixels-to-actions-human-level.html
- Ravichandiran, Sudharsan. Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow, 2nd Edition. Packt Publishing.