Navigation Udacity Deep RL Project Report

Overview:

Through this report we will introduce the work done in the first project of the Udacity Deep RL course. We will present the project structure, after that the learning algorithm with the neural network and the hyper-parameters and finishing with showing the results and showing the future work that will be done.

Project Structure:

- config.yaml: a configuration file where you could change the parameters.
- main.py: the main file which is used to run the project.
- trainer.py: the trainer file which is used to train the agent.
- evaluator.py: the evaluator file which is used to evaluate the agent.
- model.py: the deep network model created with pytorch.
- agent.py: contains the class Agent.
- replay_buffer.py: contains the ReplayBuffer class used in the DQN algorithm.
- utils.py: contains the utils used in the project.
- model.pth: the saved weights of the trained model.

Learning Algorithm:

In order to train our agent we used the **Vanilla DQN** algorithm which was first proposed by DeepMind in 2015 in order to approximate the optimal action-value function in Reinforcement Learning.
To train this algorithm, we want to minimize the mean squared error between our target Q value (according to the Bellman equation) and our current Q output.
And this learning phase could be done using Gradient Descent algorithm in order to find the best weights of the deep network.

This algorithm is illustrated in this pseudo-code:

```
**Algorithm 1** Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          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 $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
      end for
  end for
```

DQN Agent Hyper Parameters:

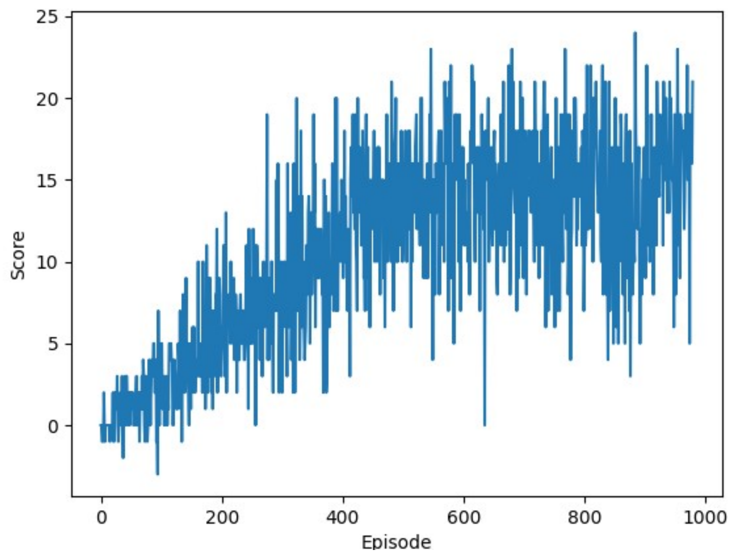
- BUFFER_SIZE (int=1e5): replay buffer size
- BATCH_SIZ (int=64): mini batch size
- GAMMA (float=0.99): discount factor
- TAU (float=1e-3): for soft update of target parameters
- LR (float=5e-4): learning rate for optimizer
- UPDATE_EVERY (int=4): how often to update the network
- n_episodes (int=1400): the maximum number of episodes
- eps_start (float=1.0): value of epsilon when running the code
- eps_end (float=0.01): min value of epsilon
- device (str=cpu): the training is don on this device could be cuda (on gpu)
- max_t (int=1000): maximium number of timestamps to be done in 1 episode in case it isn't finished (done=False)

Neural Network:

For the model we used a basic architecture consisting of **3 fully connected layers** (64, 128, 64) as number of units with **Relu** as and activation function for each one and followed by a final fully connected layer as an output with number of units equal to the number of actions.

Results:

This environment was solved after 900 episodes and you could see the plot in the figure below:



Future work:

In order to improve the results we could try other algorithms which are similar to Vanilla DQN like:
Double DQN, Dueling DQN, Dueling Double DQN etc.
Also we could make more hyper-parameter tuning by implementing some search algorithms.