This report describes the implementation of a Deep Reinforcement Learning algorithm (Q-learning) developed to solve the banana collection environment, a Deep RL benchmark of the Unity ML-agents project.

train.py

Run the algorithm by calling the **train** function within train.py, which has 2 main processes:

1) Sampling of the environment by performing actions
2) Learning from experience

The algorithm first creates an instance of the Agent class (in dqn_agent.py) and loads the Unity environment.
Then it loops over episodes. For each episode,

- The agent chooses an action A from a specific state S using the policy PI, which is e-greedy with respect to the estimated action-value function.
- Takes the action A, observes the reward R and the next input frame
- Stores the experience tuple SARS' in replay memory.
- Select a small bunch of tuples from memory randomly and learn from it using a gradient descent update step (at UPDATE_EVERY time steps)

dqn_agent.py

The algorithm implemented in dqn_agent.py has two classes for Agent and ReplayBuffer

- An agent is initialized with the parameters state_size, action_size, and a seed for random number generation in PyTorch.
- ReplayBuffer is initialized with parameters action_size, BUFFER_SIZE, BATCH_SIZE, and seed.
-

Two neural networks are initialized with the Agent, a target and a local network (qnetwork_local and qnetwork_target). Only when the agent learns the weights of the estimated action values are transferred between them. These networks represent the old Q function, which is used to compute the loss of every action during training. We use two networks because if we were updating at every time step, the Q-value estimate would diverge.
Two functions are defined for selecting and performing an action:

step(self, state, action, reward, next_state, done):

this function performs two key operations

- saves an experience in the replay memory buffer
- starts the process of learning from experience at every UPDATE_EVERY time steps (by calling the learn function, see below for details).

act (self, state, eps):

- this function selects an action for a given state following an e-greedy policy. The estimated action that maximizes reward is taken from the NN which is used as a function approximator. The architecture of this network is defined in the model.py file (see below). Basically, this model calculates the loss of every action during training.

learn(self, experiences, gamma):

- This function will update value parameters using a given batch of experience tuples.
- It will first get the max predicted Q values (for next states) from the target model and compute Q targets for current states
- Then, it will get the expected Q values from the local model, compute the loss and minimize the loss using gradient descent
- Finally, it will update the target network with the new weights

Replay Buffer

The replay buffer implemented as a class retains the end most recent experience tuples. If not enough experience is available to the agent (i.e., if self.memory < BATCH_SIZE), no learning takes place.
Note that we do not clear out the memory after each episode, which enables to recall and build batches of experience from across episodes.
The buffer is implemented with a Python deque. Given that maxlen is specified to BATCH_SIZE, our buffer is bounded. Once full, when new items are added, a corresponding number of items are discarded from the opposite end.

Neural Network Architecture

The Neural network used as a function approximator was built using the PyTorch nn package. The network is defined by subclassing the torch.nn.Module class. There are two steps. We first specify the parameters of the model and then outline how they are applied to the inputs. In the constructor def___init___ function of the model.py script, we instantiate three nn.Linear modules and assign them as member variables. The architecture includes an input layer of the size of the state size, two fully connected hidden layers and an output layer.
RELU activation (Regularized linear units) is applied in the forward function. This function accepts a Variable of input data (in this case, a particular state of 37 dimensions, and return predicted action values as output data. Note that on the output side, unlike a traditional RL setup where only one Q value is produced at a time, we are generating an estimated Q value for every possible action in a single forward pass (the output layer maps all neurons in the second hidden layer with all four possible actions).

Thus, our function approximator is a 3 layer fully connected neural network with an input layer of 37 dimensions, a first hidden layer of 64 dimensions, a second hidden

layer of 64 dimensions, and a final output layer of four dimensions for each of the 4 different actions. The model is trained using a variant of the Stochastic-Gradient-Descent algorithm (specifically the Adam optimizer) to update the weights.

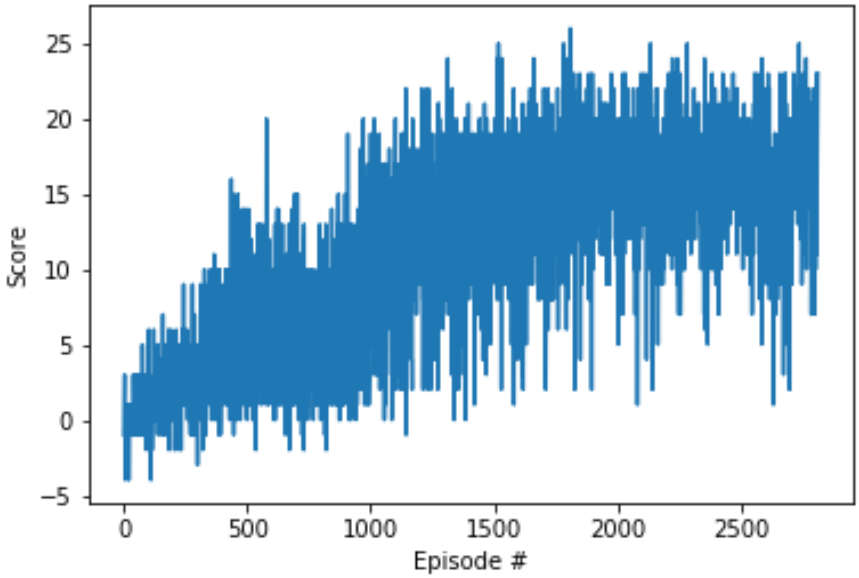
Chosen hyperparameters

The agent was trained until an average score of 17 over 100 consecutive episodes was reached. This limit was reached after 2713 episodes

Using the same parameters in a different simulation, we observed that the average score of 13 over 100 consecutive episodes was reached at episode 1278.

```
Unity brain name: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
Episode 100     Average Score: 0.443
Episode 200     Average Score: 1.55
Episode 300     Average Score: 2.39
Episode 400     Average Score: 3.72
Episode 500     Average Score: 5.75
Episode 600     Average Score: 6.56
Episode 700     Average Score: 6.20
Episode 800     Average Score: 6.31
Episode 900     Average Score: 6.46
Episode 1000    Average Score: 8.88
Episode 1100    Average Score: 8.93
Episode 1200    Average Score: 10.90
Episode 1300    Average Score: 11.85
Episode 1378    Average Score: 13.08
Environment solved in 1278 episodes!    Average Score: 13.08
```

The plot below shows that the agent learns to perform the task and receives an average reward (over 100 episodes) of at least +13 in the late phases of learning.



Note that in order to balance the amount of exploration and exploitation during learning, the values of epsilon varied in each episode, starting from 1.0 and decaying by a factor of 0.995 until a minimum value of 0.025 was reached. This was designed to allow the agent to randomly explore less rewarding actions with a small probability during the training process.

Instructions to train and test the agent's performance

To train the agent, run the train.py script from the terminal.
train.py contains the function dqn_funtion that will train the agent with the following parameters: number of episodes = 2000, and values for epsilon (start= 1, min= 0.025, decay = 0.995).

To observe a trained agent behaving, run the test.py script. This will launch the compiled unity application and show the behavior of the trained agent in 5 episodes.

Ideas for future work

This is a basic agent built from the architecture provided in the Udacity's Lunar Lander exercise.
The task is perfectly solvable with this implementation however it would be interesting to check other neural network architectures and apply the optimizations suggested to the DQN algorithm which include:
Double DQN: Helps to avoid the overestimation of action values typically observed in Q-learning.
Prioritized Experience Replay: Some experience transitions are more relevant for learning, and the more important transitions should be sampled with higher probability during experience replay.
Dueling DQN: A dueling architecture can assess the value of each state, without having to learn the effect of each action as in traditional DQN.