Deep Reinforcement Learning Nanodegree

Report - Project 1: Navigation

1 Introduction

The project demonstrates the ability of value-based methods, specifically, Deep Q-learning and its variants, to learn a suitable policy in a model-free Reinforcement Learning setting using a Unity environment, which consists of a continuous state space of 37 dimensions, with the goal to navigate around and collect yellow bananas (reward: +1) while avoiding blue bananas (reward: -1). There are 4 actions to choose from: move left, move right, move forward and move backward.

The project is composed by four files:

Navigation_DQN.ipynb - a Jupyter Notebook file containing the main code to initialize dependencies, environment and Agent.

dqn_agent.py - a code that contains the characteristics of the agent and how it behaves through this projects exigencies.

model.py - a model of the deep neural network(DNN) used by the agent to learn a policy and solve the environment .

dqn.pth, ddqn.pth, duel_dqn.th - a file with the DNN's weights, that solved the environment, saved.

The problem trying to be solved is modeled as a Markov Decision Process, involving mappings from states to actions, in such way that this actions will maximize the total cumulative reward signal of the agent. States are any information that the environment provides the agent, excluding the reward signal. Actions are ways that an agent can interact with the environment, and rewards are signals that the agent receive after interacting with the environment.

The solution to the problem, on this project, is a mapping called policy, ϵ(epsilon)-greedy policy specifically, that is guided towards the maximum cumulative rewards by the action-value functions (q(s,a)). This function gives the agent a sense of how good or bad is to be in a given state. The actions are selected more probably for the ones with maximum q(s,a) in the state, while choosing less probably the other actions. After interacting many times with the environment, it's possible to find an optimal policy coming from an optimal action-value function q* (s,a). The sense of optimal here is a mapping that can accomplish a given goal, with certain restraints.

To approximate this function Deep Q-Networks are used. These are value based methods of Reinforcement Learning that make use of DNN's as non-linear function approximators. With a state as input, the network outputs q(s,a) for each possible action in that state. After a defined number of time steps, DNN's parameters are modified towards minimizing loss between a target action value function, and the currently one being used.

2 Learning Algorithm :

A Deep Q-Network agent is imported from dqn_agent code and initialized with proper parameters: The state and action space sizes. The dqn function is defined with its parameters: The number of training episodes of agent x environment interaction, number of time steps for each episode, starting, final and decay values for the epsilon-greedy policy.

In each episode the environment is reset and the agent is put in an initial state, while the number of timesteps is less than a maximum defined, at the dqn function parameters, the following procedures are done:

The agent takes an action by evaluating the state it belongs, the action chosen is either the greedy action, the one that maximizes the action-value function, with probability equals to ϵ or, with 1-ϵ probability, a random action (including the greedy one). After selecting an action, and sending it to the environment, the agent receives its next state, reward signal and an information that says if episode is over or not. With these information the agent takes a step, these means that the following will happen: It will store the information from the environment, called as one experience, action, reward and next state in a memory for future learning, this technique is called Experience Replay because it can provide multiple learning steps from a single experience. This procedure is made until memory's size is greater than batch size hyper parameter. When this happens the agent calls it's learn method, and a number of experiences is randomly sampled from the memory .

As previously explained the objective is to approximate a target action value function. This is done by first, as the agent is initialized, creating two different Q-Networks, target and local, this approach is named Fixed Q-Targets, standing that one will remain unchanged while the other will move towards it. The learning takes place in a numerical optimization using stochastic gradient descent, with some tweaks, called Adam. The optimization objective is to find DNN's weights that minimize the mean squared error between target's Q-Network predicted action value for the next state and the local's network predicted action value for the current state. Each optimizer step is taken in the direction of minimizing this loss with respect to the weights. At least a soft update is made in the target network parameters, this update is controlled by a parameter TAU, that defines a proportion of the local networks weights on the new target network.

Another implementation used is Double Q-Learning, instead of evaluating the target Q-network's values using its own best actions predictions, that would be highly biased, the model uses the local network's best actions for predicting the target next state values. After this agent step the state becomes the next state, giving continuation to the episode, reward is added to scores variable and environment checks if the episode is over, otherwise, another time step is executed and the agent selects an action.

Another implementation is Dueling Network, normally DQNs have a single output stream with the number of output nodes equal to the number of actions. But this could lead to unnecessarily estimating the value of all the actions for states for states which are clearly bad and where, choosing any action won't matter that much. So, the idea behind dueling networks is to have two output streams, with a shared feature extractor layer.

3 Hyperparameters :

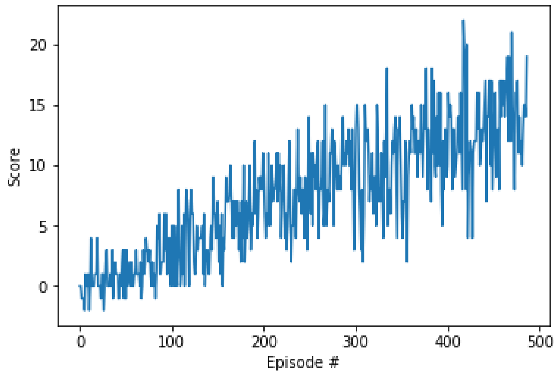
Hyperparameters used in this experiment are given below,

Hyperparameter	Value
Replay Buffer Size	1e5
Batch Size	64
Gamma (Discount factor)	0.99
TAU	1e-3
Learning Rate	5e-4
Update interval	4
Epsilon start	1
Epsilon minimum	0.025
Epsilon Decay	0.995

4 Plot of rewards :

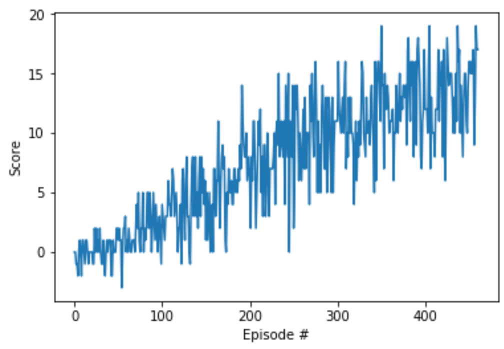
DQN

```
Episode 100     Average Score: 1.12
Episode 200     Average Score: 4.88
Episode 300     Average Score: 8.34
Episode 400     Average Score: 10.43
Episode 487     Average Score: 13.03
Environment solved in 387 episodes!     Average Score: 13.03
```



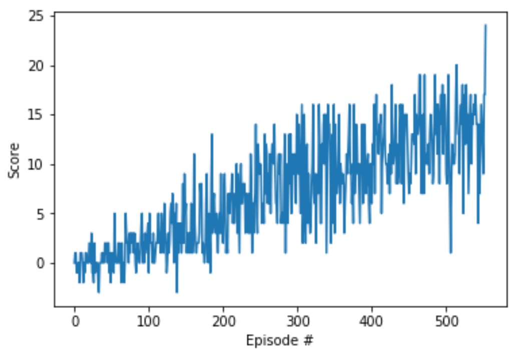
Double DQN

```
Episode 100     Average Score: 0.89
Episode 200     Average Score: 4.92
Episode 300     Average Score: 8.88
Episode 400     Average Score: 11.86
Episode 461       Average Score: 13.04
Environment solved in 361 episodes!        Average Score: 13.04
```



Dueling DQN

```
Episode 100       Average Score: 0.76
Episode 200       Average Score: 3.29
Episode 300       Average Score: 7.28
Episode 400       Average Score: 9.20
Episode 500       Average Score: 12.30
Episode 554       Average Score: 13.10
Environment solved in 454 episodes!        Average Score: 13.10
```



The best performance is achieved by Double DQN, where the environment is solved in 361 episodes compared to DQN with 387 episodes and Dueling DQN with 454 episodes.

5 Ideas for Future Work

- Other techniques like Prioritized Experience Replay, Multi-Step Learning and Distributional RL should be used for further improvement .
- It will be interesting to try learning directly from the pixels.