Report Udacity Navigation

Deep Q Network

Google's DeepMind set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a central goal of general artificial intelligence that has eluded previous efforts. To achieve this, they developed a novel agent, a deep Q-network (DQN)[3], which was able to combine reinforcement learning with a class of artificial neural network known as deep neural networks.

Deep Q Learning combines 2 approaches :

1. A Reinforcement Learning method called Q Learning.
2. A Deep Neural Network to learn a Q-table approximation.

Code Implementation

This project has three files – model.py, dqn_agent.py and navigation.ipynb

1. model.py

The code used here is derived from the "Lunar Lander" project a part of Udacity's Deep Reinforcement Learning Nanodegree that used PyTorch to implement a Full connected Neural Network. The network is trained to predict the best action i.e. direction for agent to move towards the yellow banana.

Model Architecture

- The input layer which size depends of the state_size parameter passed.
- There are two hidden fully connected layers with 64 nodes each.
- The output layer which size depends of the action_size parameter passed.
- Activation Function used: ReLu.

ReLU stands for rectified linear unit. **ReLU** is the most commonly used **activation function** in neural networks, especially in CNNs. The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero.

2. dqn_agent.py

This code is also derived from the Lunar Lander project. To overcome the problem of DQN there are two popular methods - Replay buffer and Fixed Q

We have implemented Replay buffer where the Q value are retained in Q and is updated every 4 steps. It has the following methods()

constructor() – to initialize the replay buffer and neural networks – local and target neural network

step() – Store the state in replay buffer and update the target neural network after every 4 steps

act() – retuans an action for a given state based on the current policy

learn() – update the neural network weights

Replay Buffer

add() – adds experience step to memory

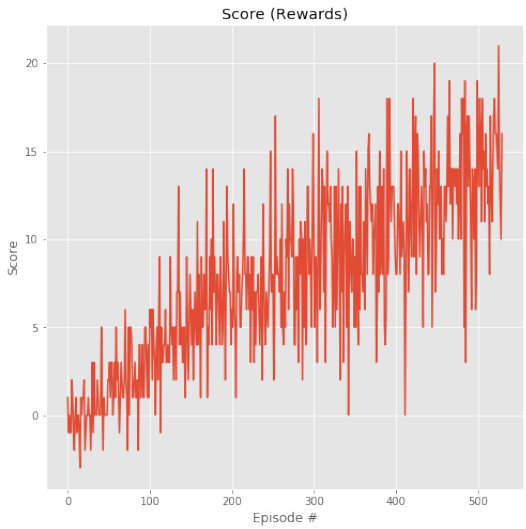
sample() – randomly sample a batch of experience for learning

Parameters

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.995           # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

3. navigation.ipynb
This code is similar to the one used in Lunar Lander but the difference is in the environment. Lunar Lander used OpenAI GYM while this banana navigator uses Unity3D. With a few changes to the navigation code it can be used for navigating the agent towards the yellow banana.
Jupyter Cell
[1] Import the necessary packages
[2] Examine the State and Action Spaces
[3] Take randam Actions in the enviroment
[4] Train the agent using DQN
[5] Plot the scores

Output: Model – checkpoint.pth



Plot – Episode Vs Score

Environment solved in 430 episodes
Average Score: 13.00
Total Training time = 11.4 min

Future Works

So far the agent is trained only using DQN which can also be implemented using Double DQN and Pritorized Experience Replay and Dueling DQN to increase their performance.
Also in this attempt the agent interacted with the environment but the agent[1] can be trained using raw pixels from the environment as input.

Reference

[1] Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., Lange, D. (2018). Unity: A General Platform for Intelligent Agents. [arXiv preprint arXiv:1809.02627.]
[2] R. S. Sutton and A. G. Barto, Introduction to Reinforcement Learning, 2nd ed. Cambridge, MA, USA: MIT Press, 2017
[3] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015.