Navigation Report

The project implements a simple agent trained using Reinforcement Learning to navigate large environment in continuous space. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The goal of our agent is to collect as many yellow bananas as possible while avoiding blue bananas. In order to solve the environment, our agent must achieve an average score of +13 over 100 consecutive episodes.

Implementation

The implementation is split into the following modules:

- **model.py**: contains Neural Network implementation in [PyTorch]() which is used for local and target Q-network.

- **dqn_agent.py** : contains the agent implementations .

- **Navigation.ipynb**: imports all the required modules and allows to explore the environment and traind the agent

Environment

A reinforcement learning task is about training an agent which interacts with its environment. The agent arrives at different scenarios known as states by performing actions. Actions lead to rewards which could be positive and negative.

The agent has only one purpose here – to maximize its total reward across an episode. This episode is anything and everything that happens between the first state and the last or terminal state within the environment. We reinforce the agent to learn to perform the best actions by experience. This is the strategy or policy.

The environment is already saved in the Workspace and was accessed through the file path provided. The Workspace provided was a jupyter notebook which contain all the needed files. The project uses an environment built in Unity.Below are the characteristics of the environment.

- Unity Academy name: Academy
- Number of Brains: 1
- Number of External Brains : 1
- Lesson number : 0

- 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

The Algorithm

Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform. But what if this cheatsheet is too long? Imagine an environment with 10,000 states and 1,000 actions per state. This would create a table of 10 million cells. Things will quickly get out of control!

It is pretty clear that we cannot infer the Q-value of new states from already explored states. This presents two problems:

- First, the amount of memory required to save and update that table would increase as the number of states increases

- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic

Here's a thought – what if we approximate these Q-values with machine learning models such as a neural network? Well, this was the idea behind DeepMind's algorithm.

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.

Below are the steps involved in reinforcement learning using deep Q-learning networks (DQNs)

1. All the past experience is stored by the user in memory

2. The next action is determined by the maximum output of the Q-network.

3. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The steps involved in a deep Q-network (DQN) are listed below:

1. Preprocess and feed the game screen (state s) to our DQN, which will return the Q values of all possible actions in the state

2. Select an action using the epsilon-greedy policy. With the probability epsilon, we select a random action a and with probability 1-epsilon, we select an action that has a maximum Q-value, such as a = argmax(Q(s,a,w))

3. Perform this action in a state s and move to a new state s' to receive a reward. This state s' is the preprocessed image of the next game screen. We store this transition in our replay buffer as <s,a,r,s'>

4. Next, sample some random batches of transitions from the replay buffer and calculate the loss

5. It is known that: which is just the squared difference between target Q and predicted Q

6. Perform gradient descent with respect to our actual network parameters in order to minimize this loss

7. After every C iterations, copy our actual network weights to the target network weights

8. Repeat these steps for M number of episodes

Results

Once the agent achieves the score over 13, the problem is considered solved. After tuning of the parameters, I was able to solve the problem in 273 episodes. The plot below shows the rewards per episode and moving average over last 100 episodes.

Further improvements

1. Parametric noise can be added to the weights to induce stochasticity to the agent's policy, producing greater efficient exploration.

2. prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error can minimize the total training time