REPORT

PROJECT 1: NAVIGATION

AUTHOR NAME:VIGNESH.B.YAADAV

In this project, we'll be using a value-based method which is a sub-portion of Reinforcement Learning. Here the Deep Q-Learning algorithm represents the optimal action-value function q* as a neural network.
In DQL we use a neural network to represent the action values.
Using reinforcement learning is quite unstable and we address the issues in DQL by using
-> Experience Replay
-> Fixed Q-Targets

Why use Experience Replay?
By using the Replay pool the behavior distribution is averaged over many of its previous states.

Why use Fixed Q-Targets?
here we use a target network to represent the old Q function we use to calculate the loss of every action during training.

The environment of the game state consists of a continuous state space of 37 dimensions, intending 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 aims to demonstrate the ability of the Deep Q-learning Algorithm to solve the "Banana Collecting Game" here the agent has to learn to gather yellow bananas and avoid blue bananas from the collections. To understand the basics idea of Q-learning is to learn the action-value function often denoted as Q(s, a), where (s)represents the current state and (a) represents the action. Q-learning is a derivative of Temporal-Difference learning where unlike Monte-Carlo methods, we can learn from each step rather than waiting for an episode to complete. The basic idea is that once we take action and update the experience into a new state, we use the current Q-value of that state as the estimate for future rewards. There's one niche problem here. Since our space is continuous, we can't use a tabular/scalar representation. Hence, we use a function approximately.

The core idea behind a function approximator is to introduce a new parameter that helps us to obtain an approximate of the Q(s, a). So, this tends to become a supervised-learning problem where the approximate represents the expected value and becomes the target. To compensate this use mean-square error as (loss function) and update the weights accordingly using gradient-descent. Now all that's left with us is the function approximator. That's Deep Learning right! (hope so) We use a neural network as a function approximator here (certainly it is !!). More specifically, we choose a 2-hidden layer network with both the layers having 64 hidden input units with a relu activation function applied after each fully-connected layer. Adam optimizer was used as the optimizer for finding the optimal weights of the network.

Here the agent interacts with the game environment through a sequence of observations as humans and interacts with its action and we penalize for not doing good and reward it for rights actions taken. The agent's goal is to select the action in a fashion that maximizes cumulative future reward. We use a CNN to approx the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s,\ a_t = a,\ \pi\right]$$

rt -> sum of rewards
ɣ -> discount
t -> time step t
π -> Policy P(a|s) a ->
action,s->state

-> Experience Replay
When an agent is interacting with the environment, the sequence of experience tuples is correlated.
The replay buffer mainly contains the experience tuples (S, A, R, S').
Here we sample a small batch of tuples from the replay buffer to learn from it we call it experience replay. Experience replay is based on the idea that we can learn better if we do multiple passes over the same experience. It is used to generate uncorrelated experience while training.

-> Fixed Q-targets
Her goal is to reduce the difference between TD Target and the current predicted Q-value

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

DQL Algorithm
There are two basic
steps here
-> Sample the batches
-> Learn from the sample experience

Initialize target action-value weights w- ~ w
How does the algorithm work?
For each episode and each time step t within that episode, the raw screen state 84x84 pixel state is observed and it is converted into a greyscale 24 x 24.
The model is built upon 3 fully connected layers number of neurons in the first 2 layers is 64 and the last layer is equal to the number of output sizes.

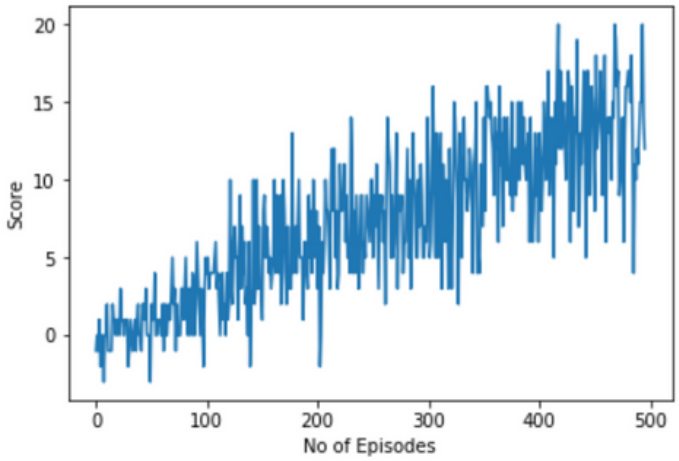
4.3 Hyperparameters

- BUFFER_SIZE = int(1e5) # replay buffer size
- BATCH_SIZE = 64 # minibatch size
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR = 5e-4 # learning rate
- n_episodes = 2000 # maximum number of training episodes
- max_t = 1000 # maximum number of time steps per episode
- eps_start = 1.0 # starting value of epsilon, for epsilon-greedy action selection
- eps_end = 0.01 # minimum value of epsilon
- eps_decay = 0.995 # multiplicative factor (per episode) for decreasing epsilon

While training the agent on 2000 epochs the agent solved the environment in 496 episodes with an average score of 13.01

```
In [10]: scores = dqn()
```

```
Episode 100     Average Score: 0.95
Episode 200     Average Score: 4.79
Episode 300     Average Score: 7.36
Episode 400     Average Score: 10.17
Episode 496     Average Score: 13.01
Environment solved in 496 episodes!      Average score: 13.01
```



when the trained agent is loaded and allowed to play for 3 iterations its scored an average of 15.3 in all 3 games.

```
In [12]: #Load a trained agent
agent.qnetwork_local.load_state_dict(torch.load("checkpoint.pth"))

for i in range(3):
    env_info = env.reset(train_mode=False)[brain_name]
    state = env_info.vector_observations[0]
    score = 0
    for j in range(1000):
        action = agent.act(state)
        env_info = env.step(action)[brain_name]
        next_state = env_info.vector_observations[0]
        reward = env_info.rewards[0]
        done = env_info.local_done[0]
        agent.step(state, action, reward, next_state, done)
        state = next_state
        score += reward
        if done:
            break

    print("\rEpisode {}\t Score: {:.2f}".format(i+1, score))
```

```
Episode 1        Score: 14.00
Episode 2        Score: 14.00
Episode 3        Score: 18.00
```