Project Navigation Report

Environment Background

The environment used was a modified version of UnityML Agents Banana environment. The agent in this environment is tasked with navigating through a world with blue or yellow bananas. The blue bananas provide a reward of -1, while yellow bananas provide a reward of +1. The goal of the agent is to maximize the number of yellow bananas collected.

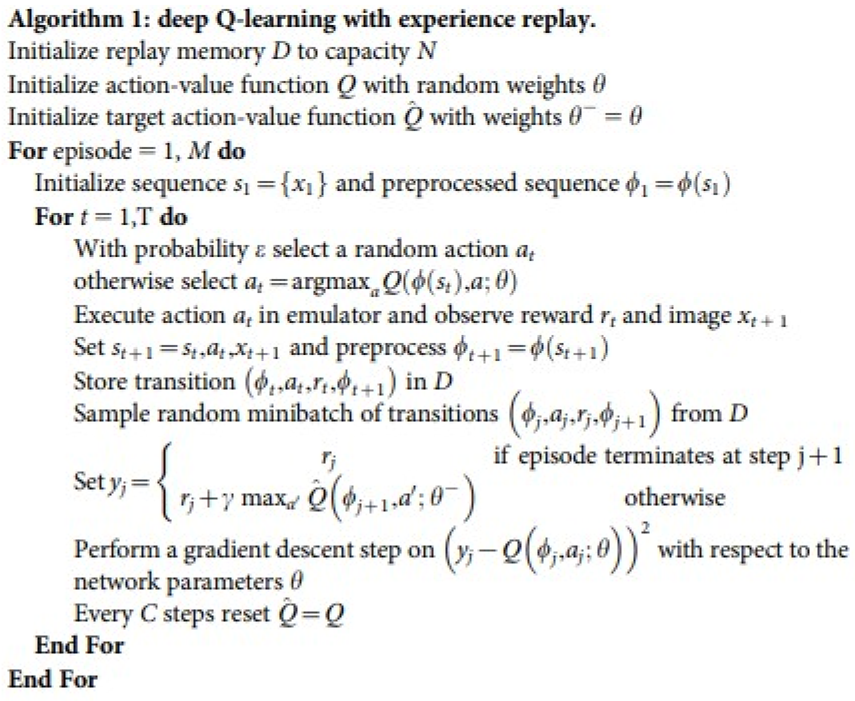
The agent is limited to moving forward or backward or turning left or right. The state space consists of the velocity of the agent and a representation of the objects in the direction the agent is facing in. The task is episodic with the environment considered solved when the agent averages a score of at least +13 over 100 consecutive episodes

DQN Learning Algorithm

Theory

The algorithm used to solve this environment is a Deep Q-Network or DQN algorithm. The DQN algorithm was first used to solve Atari games in a groundbreaking paper by AI researchers at Google's DeepMind in 2015. The algorithm involves sampling from the environment and storing the action state and reward information to later learn from and estimate the action-value function. The pseudocode for the algorithm is recreated as it was presented from the original DeepMind paper in Figure 1.

Figure 1: The DQN algorithm



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
   Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
   For $t$ = 1,T do
      With probability $\varepsilon$ select a random action $a_t$
      otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
      Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
      Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
      Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
      network parameters $\theta$
      Every $C$ steps reset $\hat{Q} = Q$
   End For
End For
```

In essence, the algorithm samples the environment to obtain a state, action, reward, next state tuple that is stored in the memory buffer, D. The memory buffer is then sampled randomly every C steps and used to update the network weights. The original algorithm included the memory buffer and fixed Q-Targets to avoid two common problems with traditional Q-learning. The memory buffer with random sampling prevents the influence of consecutive experience samples since the previous experience sample can effect the next experience sample. The fixed Q-targets prevent changing the weights of the neural network based on constantly changing parameters. The DeepMind paper includes a more expansive discussion of these additions that stabilized the algorithm.

Implementation

The DQN Algorithm solution presented in this project involves defining a QNetwork, a DQN agent, a memory buffer, and the DQN algorithm used to train the DQN agent. Each component of the implementation is described below.

QNetwork: The action value function network was defined in the class QNetwork. The QNetwork defines a neural network with 3 layers. The first layer has an input size equal to the stateSize variable, a representation of the environment's state size, and an output size of 64. The second layer has an input size of 64 and output size of 64. The final layer has an input size of 64 and an output size equal to the actionSize variable, a representation of the environment and agent's action size. The QNetwork has three methods: __init__, reset, and forward

- The __init__ method initializes the neural network with the architecture described above
- The reset method resets the values of the neural network
- The forward method forward propagates the input through the network

DQN Agent: The DQN agent was defined in the NavAgent class. The memory buffer was defined in the ReplayBuffer class. The agent and the memory buffer involve some key hyperparameters:

- BUFFER_SIZE is the size of the memory buffer
- BATCH_SIZE is the number of experience tuples pulled randomly from the memory buffer
- GAMMA is the discount factor used for reward calculation
- TAU is the interpolation parameter used to soft update the target network
- LEARNING_RATE is the learning rate for the optimizer
- UPDATE_EVERY is the number of every steps that the networks are updated

The NavAgent class has four methods: __init__, step, act, learn, and soft_update.

- The NavAgent init method creates a target and local QNetwork and a replay memory buffer. To allow for updating the weights every few steps and not after each step, the init method also creates a timestep counter, t_step. The parameters for the init method are the stateSize, actionSize, and a random seed number.
- The step method adds an experience tuple to the memory buffer and updates the network weights every UPDATE_EVERY number of steps.
- The act method involves the agent taking an action in an epsilon greedy manner
- The learn method involves the agent learning from an experience by updating the model parameters through sampling a batch of experiences.
- The soft update method updates the target network.

Replay Buffer: The memory buffer class is defined in the ReplayBuffer class. ReplayBuffer has the following methods: __init__, add, sample, and __len__.

- The __init__ method creates a ReplayBuffer of size, BUFFER_SIZE, with a namedtuple with field names for the experience tuples
- The add method adds an experience tuple of state, action, reward, next state, dones to the memory buffer
- The sample method samples a set of experience tuples with a size equal to the BATCH_SIZE from the memory buffer.
- A __len__ method returns the length of the memory buffer

DQN Algorithm: The DQN algorithm is used to train the NavAgent in the dqn function. The function takes the following parameters: n_episodes, max_t, eps_start, eps_end, eps_decay.

- n_episodes: the maximum number of episodes the agent trains for
- max_t: the maximum timestep allowed for each episode
- eps_start: the starting value of epsilon used for epsilon-greedy selection of an action
- eps_end: the ending/minimum value of epsilon used for epsilon-greedy selection of an action
- eps_decay: the factor used to decrease epsilon per episode

The algorithm creates a score buffer with length of 100 to store 100 episodes of reward scores. For each episode, the agent acts, receives reward information, and updates the memory buffer with the new experience tuple from timesteps t=0 to t=max_t. The score per episode is the sum of the reward per timestep. The score per episode is appended to the score memory buffer. If the episode number is a multiple of 100, the average of the 100 scores in the buffer. Since the environment was considered solved when the average score over a hundred episodes was greater than 13, the target was to test, manipulate, and find the hyperparameter values that enable the agent to achieve a score of 13 or greater over 100 episodes.

Hyperparameter Values in the Successful Solution

After testing and experimenting with multiple different hyperparameter values, the following values produced a successful solution to the environment.

DQN Agent and Buffer Hyperparameters

BUFFER_SIZE = 1e5
BATCH_SIZE = 64
GAMMA = 0.99
TAU = 1e-3
LEARNING_UPDATE = 5e-4
UPDATE_EVERY = 4

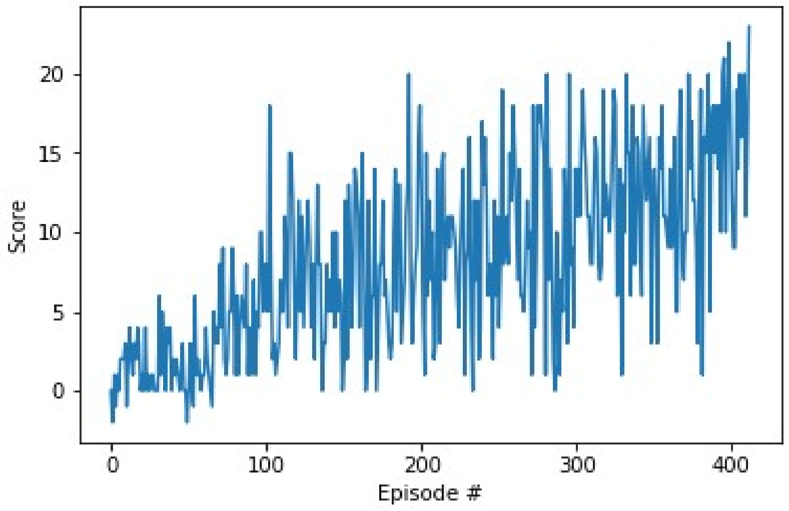
DQN Algorithm Function Hyperparameters

n_episodes = 4000

max_t = 1000
eps_start =1.0
eps_end = 0.01
eps_decay = 0.8

The implementation described above with the above hyperparameters solved the environment in 312 episodes with an average score of 13.04. A plot of the rewards is posted below.

Figure 2 Score Plot



Future Directions

For future improvements, the directions that could be taken are additions to the DQN algorithm. Specifically, the addition such as the Double DQN, Dueling DQN's, or prioritized experience replay. These additions would improve the learning potential of the DQN algorithm and may help the agent learn even faster.

References

Original DQN Paper: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf