Navigation – Banana Collector

The Environment

The environment used for this project is the Banana Collector environment created by Unity. The environment is a square world with four walls containing yellow and blue bananas scattered around. The agent receives a reward of +1 for collecting a yellow banana and -1 for collecting a blue banana. The objective is to navigate around the environment collecting yellow bananas and avoiding blue ones. The environment consists of episodic tasks and is considered solved when a reward of 13 is achieved.

The states in the environment have 37 dimensions which contains information about the agent's position, velocity and what's in front of the agent. From each state, one of 4 discrete actions can be taken. These actions are move forward, move backwards, turn left, and turn right.

Deep Q-Learning

Theory and Implementation

In reinforcement learning the action-value function q_* is represented by a mapping between state-action pairs and the q value, is represented in a table, Q table. To find the optimal policy for the agent the optimal action-value function needs to be found. In q learning the process to find the optimal action-value function is to first initialize the Q table with zeros. Then choose an action based on the policy, get the returned reward and next state from the environment then update the Q table using the Bellman equation.

In Deep Q learning we replace the action-value function, Q table, with a neural network. A deep neural network uses non-linear function approximators to calculate the action values based on the input states. The neural network is then trained to find the optimal weights which will produce the optimal action-value function and therefore the best policy.

Fixed Q-Targets

There are two main features that need to be implemented to make a more stable Deep Q Network, the first feature is to use Fixed Q-Targets. Fixed Q-Targets improves the learning by making use of 2 neural networks, a local and target network. The reason for using 2 neural networks is the weights of the model constantly change so we need to have a second model that has fixed weights so the loss can be computed. The update rule to update the weights in our network to find the optimal action-value function is given by,

$$\Delta w = \alpha \cdot (R + \gamma \max \hat{q}(S', a, w^-) - \hat{q}(S', a, w)) \nabla_w \hat{q}(S', a, w) \quad (1)$$

where (S, a, R, S') is our experience, w is our weights on the local network and w^- are the weights on the target network that remain fixed for a pre-defined number of time steps. The first term inside the brackets,

$$R + \gamma \max \hat{q}(S', a, w^-) \quad (2)$$

is the TD target which is our target network and the second term,

$$\hat{q}(S', a, w) \quad (3)$$

is the local network. Combined they are the TD error.

In the implementation the networks used have the same architecture, and the target network is updated every five time steps. The architecture for the DQN can be seen below,

```
class model(nn.Module):
    def __init__(self, state_size, action_size, seed):
        super(model, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))

        return self.fc3(x)
```

A simple neural network is used made up of an input layer, output layer and 1 hidden layer containing 64 neurons. More complex neural networks produce better results in the long run but take more episodes to train and solve the environment. When updating the network, we have two options either the weights can be copied from one network to another or we can slowly adjust them. See below for the update functions,

```
def soft_update(self, local_model, target_model, tau):

    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)

def update_target(self, local_model, target_model):

    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(local_param.data)
```

in the implementation for the vanilla DQN the update target function solved the environment in 402 episodes and with the soft update function the DQN solved the environment in 769 episodes. For the Double DQN when using the update target function the Double DQN wasn't able to solve the environment but the Double DQN when using the soft update function solved the environment in 342 episodes.

Experience Replay

The second feature is called Experience Replay. Instead of training the model every time step experiences are saved to memory/replay buffer. The episodes in the memory are sampled randomly in batches based on a uniform distribution. In the DQN implementation the batch size was set to 64. This method improves the DQN as it stores experiences that may occur infrequently and so by storing it to a memory and then uniformly sampling the memory the DQN has a chance to learn from these rare occurring experiences. Another reason why Experience Replay improves the DQN is the algorithm is randomly sampling experiences so any correlation between experiences is lost.

Implementation

To implement a DQN there is first a sample step and then a learning step. In the sample we first choose an action based on the policy. In the implementation the epsilon-greedy policy has been chosen,

```
def select_action(self, state, eps):
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    action_values = self.model_local(state)

    if random.random() > eps:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.action_size))
```

Epsilon-Greedy has been chosen as it is one of the best solutions for the exploration exploitation tradeoff. The value for epsilon starts at 1 and has a min value of 0.01. The value of epsilon is decayed with each episode. A random number is selected and if this number is greater than the value for epsilon then an action with the highest Q value is taken otherwise a random action is taken. As the value for Epsilon decays random actions are less likely to be chosen and the DQN agent will start exploiting what it has already learnt.

After an action has been taken the experience tuple (S, A, R, S') is returned and stored in a replay memory. In the learn step a batch of experiences is sampled from the replay memory. Then the target is set and the loss is calculated between the two action-value functions. Once the loss is calculated the neural network is trained and the weights are updated. Finally, after a certain number of time steps the weights on both models are synced.

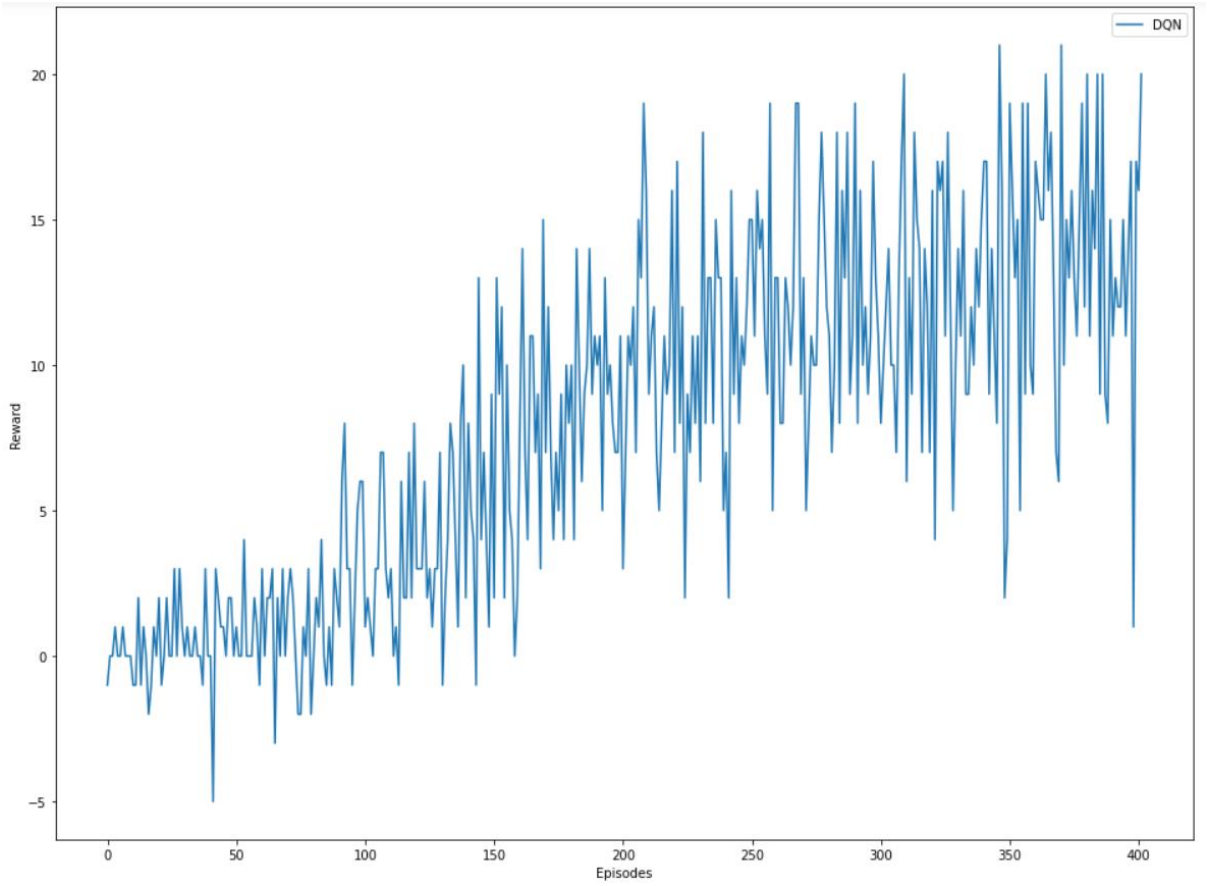
DQN Results

With the following hyperparameters,

{buffer_size: 25000,
batch_size: 64,
eps_start: 1.0,
eps_end: 0.01,
eps_decay: 0.99,
gamma: 0.9,
update_every: 5,
local network learning rate: 1e-3,
model copy weights: update_target is used}

the DQN model was able to solve the environment in 402 episodes.

```
Episode 100    Average Score: 0.93
Episode 200    Average Score: 6.01
Episode 300    Average Score: 11.33
Episode 400    Average Score: 12.90
Episode 402    Average Score: 13.08
```



Double DQN

One problem with DQNs is the vanilla DQN over estimates action values as the DQN always picks the maximum value for the next state so introduces a bias in learning. To combat this double DQN use two estimators independent of each other. One of the estimators chooses the action and the other evaluates the action to make sure it's the best action, this solves the problem of overestimating action values. In the implementation the local network chooses the action and the target network evaluates the action. This works as we fix the weights of the target network. See below for the implementation,

```
def action(self, state, dqn):
    _, actions = dqn(state).max(dim=1, keepdim=True)
    return actions

def eval_action(self, states, actions, rewards, next_states, dones, gamma, dqn):

    Q_expected = dqn(next_states).gather(1, actions)
    q_values = rewards + (gamma * Q_expected * (1 - dones))

    return q_values

def double_q_update(self, states, rewards, next_states, dones, gamma):

    actions = self.action(next_states, self.model_local)
    q_values = self.eval_action(states, actions, rewards, next_states, dones, gamma, self.model_target)

    return q_values
```

Firstly, in the action function the best action is selected by the local model based on a greedy policy. The actions are then evaluated by the target model and the action values are returned in the double_q_update function.

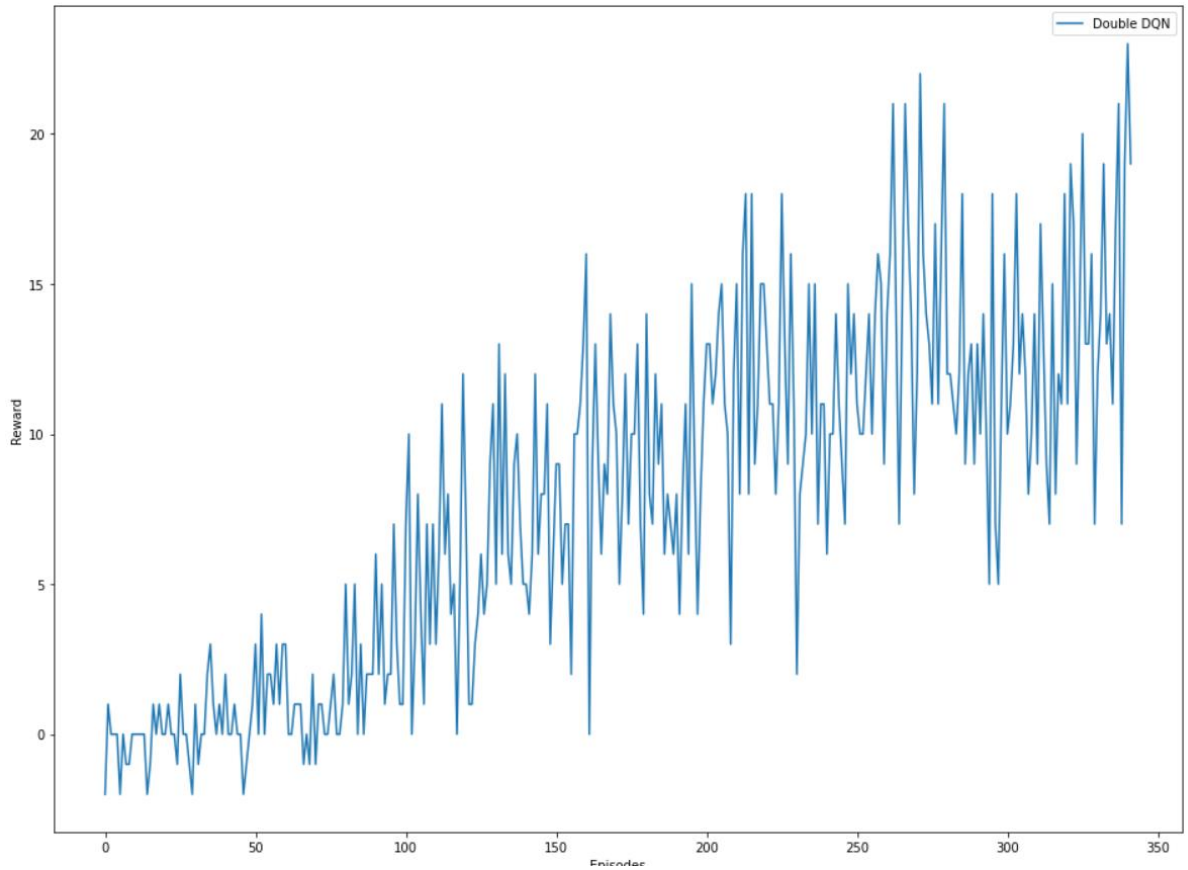
Double DQN results

With the following hyperparameters,

{buffer_size: 20000,
batch_size: 64,
eps_start: 1.0,
eps_end: 0.01,
eps_decay: 0.99,
gamma: 0.9,
update_every: 5,
local network learning rate: 1e-3,
model copy weights: soft_update
tau: 1e-3}

the DQN model was able to solve the environment in 342 episodes.

```
Episode 100    Average Score: 0.83
Episode 200    Average Score: 7.44
Episode 300    Average Score: 12.21
Episode 342    Average Score: 13.08
```



Prioritized Experience Replay

In Experience Replay experiences are sampled uniformly however some experiences maybe more important than others as we can learn more from some experiences than others. The method used in this implementation of Prioritized Experience Replay is the use the TD error,

$$\delta_t = R_{t+1} + \gamma \max_a \hat{q}(S', a, w^-) - \hat{q}(S', a, w) \tag{4}$$

to assign importance's to experiences. This error is the difference between the q values produced by the target network and the q values for the local network. The greater the error the more we can expect to learn from that experience. The absolute value of this error is called the priority,

$$p_t = |\delta_t| + e \tag{5}$$

the value e is added to the absolute value so that the priority will never equal zero. Adding this value e will mean at the sample of training all experiences will have a chance of being selected. This priority value is then used to create a sampling probability defined as,

$$P(i) = \frac{{p_i}^a}{\Sigma_k {p_k}^a} \tag{6}$$

where a is value between 0 and 1 and is used to stop overfitting due to experiences being constantly prioritized. It does this by introducing an element of uniform random sampling as some experiences are prioritized over others. When using prioritized experience replay the algorithm is not sampling uniformly so will a bias will be added in therefore a change to the update rule is made. The importance sampling weight,

$$w = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b \tag{7}$$

is added to the update rule where N is the size of the replay memory and b is a value between 0 and 1 that adds importance to the weight. For the implementation of prioritized experience replay the replay memory, called self.buffer, is a sliding window list of namedtuples with a set length defined by the buffer_size. The priorities for each experience are also stored in a sliding window list however this list is initialized with zeros.

```
class ExperienceReplay():
    def __init__(self, buffer_size, batch_size, a, prioritized):

        self.prioritized = prioritized
        self.buffer = deque(maxlen=buffer_size)
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.priorities = deque(np.zeros(buffer_size, np.float32), maxlen=buffer_size)
        self.index = 0
        self.batch_size = batch_size
        self.buffer_size = buffer_size
        self.a = a
```

Every time step an experience is added to the replay memory. If the argument prioritized is set to True then the DQN will use prioritized experience replay,

```
    def add(self, state, action, reward, next_state, done):

        if self.prioritized == True:
            priorityCopy = self.priorities.copy()
            priorityAddArray = list(priorityCopy)

            priority = max(priorityAddArray) if self.buffer else 1.0

            self.buffer.append(self.experience(state, action, reward, next_state, done))

            self.priorities.append(priority)
            self.index = min((self.index+1), self.buffer_size)

            del priorityAddArray
            del priorityCopy

        else:
            self.buffer.append(self.experience(state, action, reward, next_state, done))
```

When the time step is greater than the batch_size the DQN will start sampling experiences and learning from them as the DQN has enough experiences in memory to start learning.

```
def sample(self, beta):

    if self.prioritized == True:
        priorityCopy = self.priorities.copy()
        prioritySampleArray = np.asarray(list(priorityCopy), dtype = float)

        if len(self.buffer) == self.buffer_size:
            samplePriorities = prioritySampleArray
        else:
            samplePriorities = prioritySampleArray[-self.index:]

        sampleProbs = (samplePriorities)**self.a / (samplePriorities**self.a).sum()

        idx = np.random.choice(len(self.buffer), self.batch_size, p = sampleProbs, replace = True)

        experiences = [self.buffer[i] for i in idx]

        weights = (len(self.buffer) * sampleProbs[idx])**(-beta)
        weights = weights / weights.max()

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        del prioritySampleArray
        del priorityCopy

        return (states, actions, rewards, next_states, dones), idx, weights
```

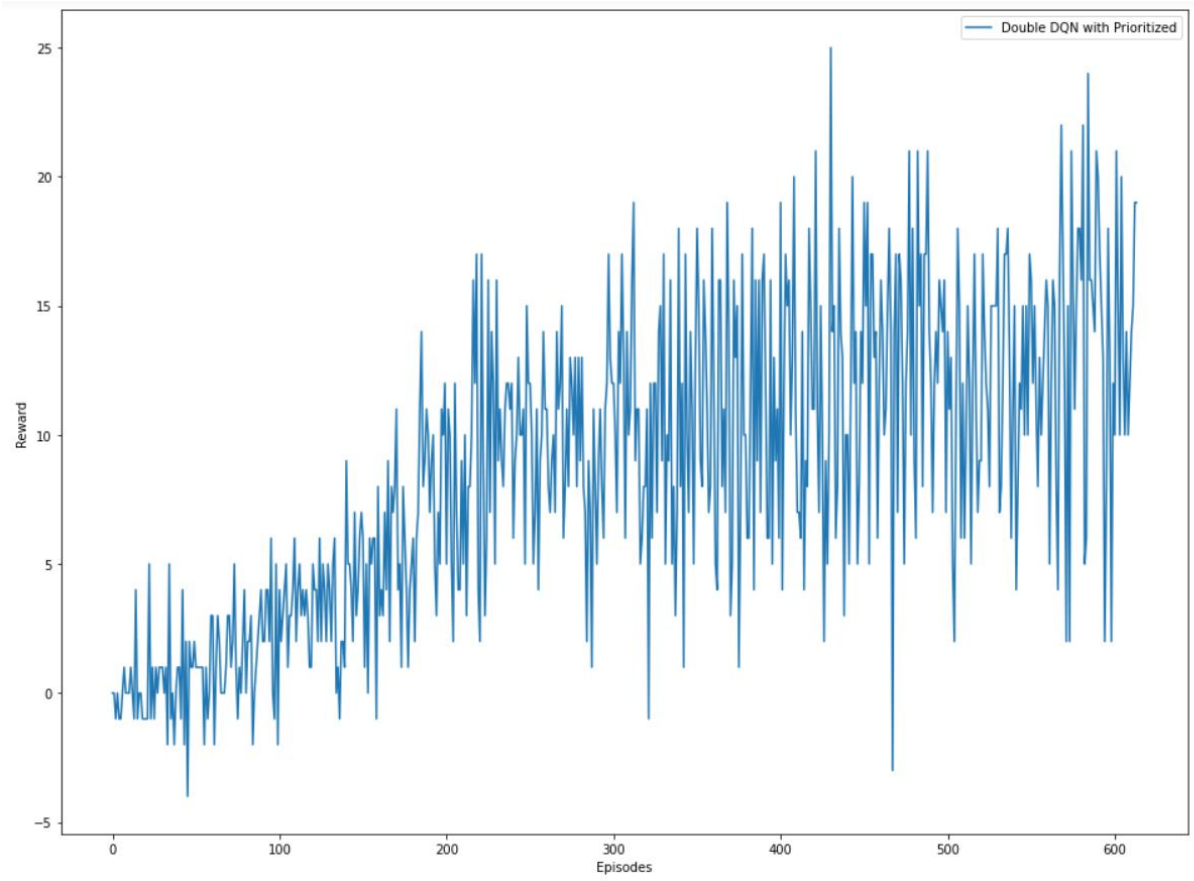
From the experiences the sampling probabilities, weights and the indices where the experiences and priorities are stored in the lists are returned.

Double DQN with prioritized experience replay

With the following hyperparameters,

{buffer_size: 20000,
batch_size: 64,
eps_start: 1.0,
eps_end: 0.01,
eps_decay: 0.99,
gamma: 0.9,
a: 0.5,
e: 1e-6,
beta: 0.4,
update_every: 5,
local network learning rate: 1e-3,
model copy weights: soft_update
tau: 1e-3}

```
Episode 100    Average Score: 0.85
Episode 200    Average Score: 4.87
Episode 300    Average Score: 9.43
Episode 400    Average Score: 10.56
Episode 500    Average Score: 12.40
Episode 600    Average Score: 12.48
Episode 614    Average Score: 13.00
```

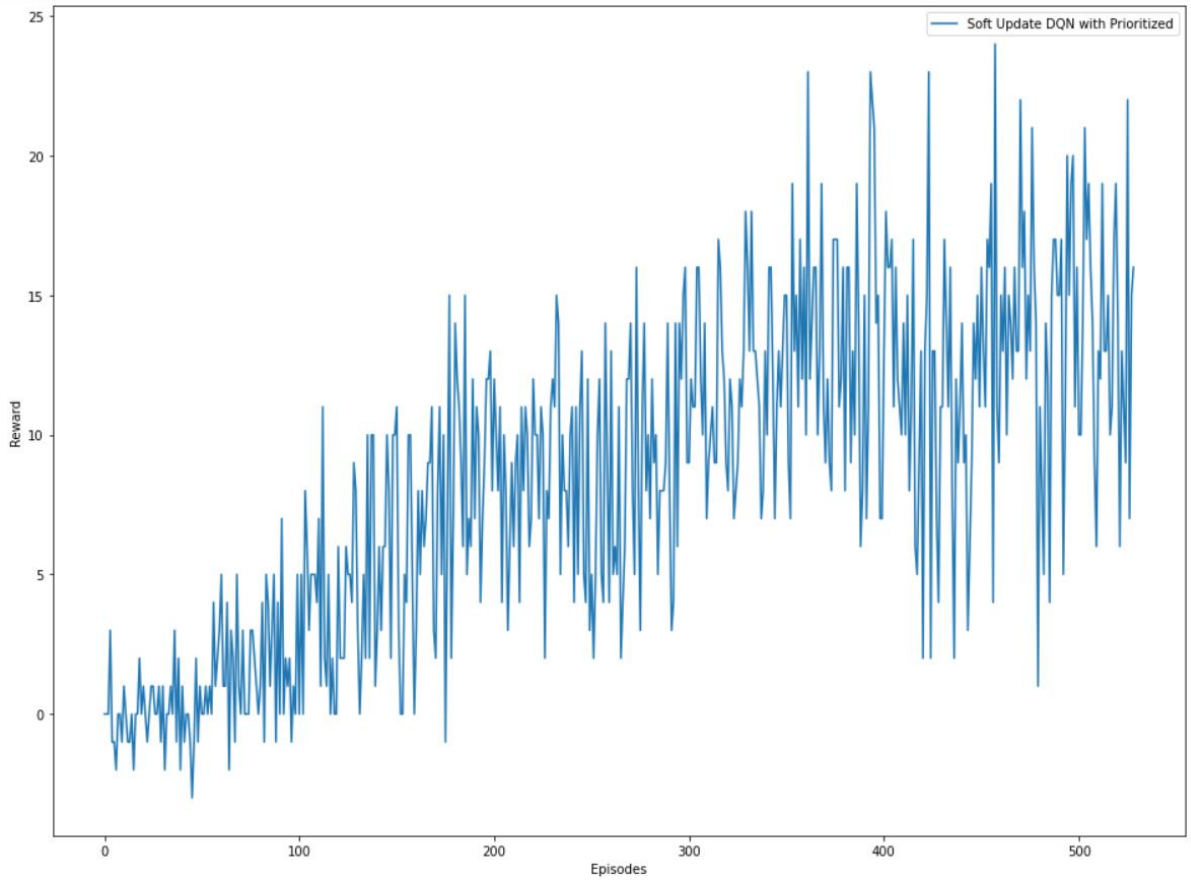


DQN with prioritized experience replay using soft update

DQN with prioritized experience replay using update target doesn't improve on the DQN without prioritized experience replay using update target but a DQN with prioritized experience replay using soft update does improve upon DQN without prioritized experience replay using soft update managing to solve the environment in 529 episodes with the following hyperparameters,

{buffer_size: 20000,
batch_size: 64,
eps_start: 1.0,
eps_end: 0.01,
eps_decay: 0.99,
gamma: 0.9,
a: 0.5,
e: 1e-6,
beta: 0.4,
update_every: 5,
local network learning rate: 1e-3,
model copy weights: soft_update,
tau: 1e-3}

```
Episode 100    Average Score: 0.83
Episode 200    Average Score: 6.01
Episode 300    Average Score: 8.63
Episode 400    Average Score: 12.57
Episode 500    Average Score: 12.57
Episode 529    Average Score: 13.08
```



Future Work