Report on Navigation Project

This project has been split into four modules due to its length, logical structure, and reusability. The modules are as follows:

- model.py: This file contains the regular DQN model as well as the dueling DQN model.
- dqn_agent.py: This file implements the agent that interacts with and learns from the environment. The file also has normal replay buffer and prioritized replay buffer.
- replay_buffer.py: The is the implementation of replay buffer and prioritized replay buffer.
- main.py: This is the main module with the training loop.

I will describe and explain the learning algorithm in some key implementations. I will also describe the model architectures below.

DQN Model Module

The code block below is the implementation for the Dueling DQN:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DuelingDQN(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        super(DuelingDQN, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2_adv = nn.Linear(fc1_units, fc2_units)
        self.fc3_adv = nn.Linear(fc2_units, action_size)
        self.fc2_val = nn.Linear(fc1_units, fc2_units)
        self.fc3_val = nn.Linear(fc2_units, 1)

    def forward(self, state):
        x = F.relu(self.fc1_val(state))
        x_val = F.relu(self.fc2_val(x))
        x_val = self.fc3_val(x_val)
        x_adv = F.relu(self.fc2_adv(x))
        x_adv = self.fc3_adv(x_adv)
```

```
        return x_val + x_adv - x_adv.mean()
```

Basically, this network processes the input using two independent paths: one path is responsible for state value V(s) prediction, which is just a single number, and another path predicts individual advantage values, having the same dimension as Q-values. After that, we add V(s) to every value of A(s, a) to obtain the Q(s, a), which is used and trained as normal.

Replay Buffer Module

The code block below is the implementation for the Prioritized Replay Buffer:

```
from collections import namedtuple, deque
import random
import numpy as np
import torch

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class PrioReplayBuffer:
    """Prioritized Replay Buffer."""
    def __init__(self, buffer_size, batch_size, seed):
        self.memory = []
        self.batch_size = batch_size
        self.experience = namedtuple("Experience",field_names=["state", "action", "reward",
"next_state", "done"])
        self.seed = random.seed(seed)
        self.priorities = np.zeros((buffer_size, ), dtype=np.float32)
        self.pos = 0
        self.prob_alpha = 0.6
        self.beta_start = 0.4
        self.beta_frames = 100000
        self.beta_idx = 0
        self.capacity = buffer_size
```

This class for the priority replay buffer stores samples in a circular buffer and NumPy array to keep priorities. We define the value for alpha for samples' prioritization and parameters for beta change the schedule. Our beta will be changed from 0.4 to 1.0.

```
    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

```
    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        # append priorities
        max_prio = self.priorities.max() if self.memory else 1.0
        if len(self.memory) < self.capacity:
            self.memory.append(e)
        else:
            self.memory[self.pos] = e
        self.priorities[self.pos] = max_prio
        self.pos = (self.pos + 1) % self.capacity
```

The add() method adds new experience to the memory. When our buffer hasn't reached the maximum capacity, we just need to append a new experience to the buffer. If the buffer is full, we need to overwrite the oldest experience and adjust the position accordingly.

```
    def sample(self):
        if len(self.memory) == self.capacity:
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]
        probs = prios ** self.prob_alpha
        probs /= probs.sum()
        # convert priorities to probabilities
        indices = np.random.choice(len(self.memory), self.batch_size, p=probs)
        experiences = [self.memory[idx] for idx in indices]
        # calculate weights
        total = len(self.memory)
        beta = min(1.0, self.beta_start + self.beta_idx * (1.0-self.beta_start)/self.beta_frames)
        self.beta_idx += 1
        weights = (total * probs[indices]) ** (-beta)
        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)

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

In the sample() method, we need to convert priorities to probabilities using our alpha hyperparameter. Then using these probabilities, we sample our buffer to obtain a batch of samples. After that, we calculate weights for samples in the batch, which will be returned to update priorities for sampled items later.

```
    def update_priorities(self, batch_indices, batch_priorities):
        for idx, prio in zip(batch_indices, batch_priorities):
            self.priorities[idx] = prio
```

The last function of the priority replay buffer allows us to update new priorities for the possessed batch.

DQN Agent Module

```
import numpy as np
import random
import torch
import torch.nn.functional as F
import torch.optim as optim

from model import QNetwork, DuelingDQN
from replay_buffer import ReplayBuffer, PrioReplayBuffer

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
UPDATE_EVERY = 4        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    def __init__(self, state_size, action_size, args, seed):
        self.state_size = state_size
        self.action_size = action_size
```

```
        self.ddqn = args.ddqn
        self.dueling = args.dueling
        self.prio = args.prio
        self.seed = random.seed(seed)

        if self.ddqn:
            print("Double DQN Enabled!")
        else:
            print("Double DQN Not Enabled!")
```

If double DQN is enabled, we just print it out. We will adjust the Q_targets_next later in the learn() method.

```
        if self.dueling: # if we train dueling DQN
            print("Dueling DQN Enabled!")
            self.qnetwork_local = DuelingDQN(state_size, action_size, seed).to(device)
            self.qnetwork_target = DuelingDQN(state_size, action_size, seed).to(device)
        else:
            print("Dueling DQN Not Enabled!")
            self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
            self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)
```

If dueling DQN enabled, we will initialized dueling network. Otherwise, we will initialized regular DQN network.

```
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        if self.prio:
            print("Prioritized Replay Buffer Enabled!")
            self.memory = PrioReplayBuffer(BUFFER_SIZE, BATCH_SIZE, seed)
        else:
            print("Prioritized Replay Buffer Not Enabled!")
            self.memory = ReplayBuffer(BUFFER_SIZE, BATCH_SIZE, seed)
```

If prioritized replay buffer is enabled, we will initialize the prioritized replay buffer class. Otherwise, we just initialize regular replay buffer class.

```
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

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

```
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))
```

In act() method, we return actions for given state as per current policy. The epsilon is for epsilon-greedy action selection.

```
    def step(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and learn
            if len(self.memory) > BATCH_SIZE:
                self.learn(GAMMA)
```

In each step, we add the experience into our replay buffer. We will learn every UPDATE_EVERY time steps when we have enough samples in the memory.

```
    def learn(self, gamma):
        if self.prio:
            (states, actions, rewards, next_states, dones), batch_indices, batch_weights =
self.memory.sample()
            batch_weights_v = torch.from_numpy(np.vstack(batch_weights)).float().to(device)
        else:
            experiences = self.memory.sample()
            states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        if self.ddqn:
            next_state_actions = self.qnetwork_local(next_states).max(1)[1]
            Q_targets_next = self.qnetwork_target(next_states).detach().\
                gather(1, next_state_actions.unsqueeze(-1))
        else:
            Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
```

```
        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)
```

We update value parameters using given batch of experience tuples. When double DQN is enabled, we calculate the best action to take in the next state using our local trained network, but values corresponding to this action come from the target network. This is the main difference between basic DQN and double DQN

```
        # Compute loss
        if self.prio:
            losses_v = batch_weights_v * (Q_expected - Q_targets) ** 2
            sample_prios_v = losses_v + 1e-5
            loss = losses_v.mean()
            self.memory.update_priorities(batch_indices,
sample_prios_v.data.cpu().numpy().squeeze(-1))
        else:
            loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

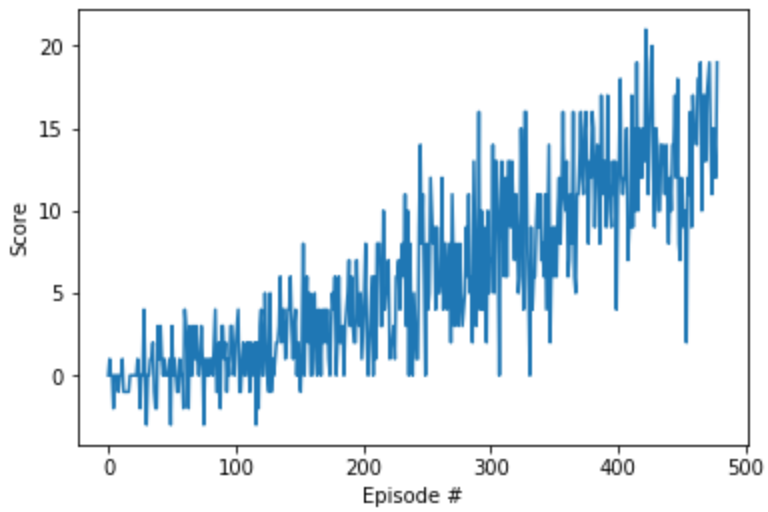
The loss calculation is customed when we use prioritized replay buffer because the MSELoss class in PyTorch doesn't support weights, we need to calculate the MSE and explicitly multiply the result on weight. This allows us to take into account weights of samples and keep individual loss values for every sample. Those values will be passed to the priority replay buffer to update priorities. Small values are added to every loss to handle the situation of zero loss value, which will lead to zero priority of entry.

```
        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target
        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)
```

Lastly we do the updating from local_model (weights will be copied from) to target model (weights will be copied to).

Plot of Rewards



Environment solved in 485 episodes with an average reward (over 100 episodes) of at least +13

Ideas for Future Work

- Implement Noust DQN model
- Implement Rainbow DQN model
- Implement Quantile Regression DQN model
- Implement Hierarchical DQN model