DQN, Double DQN, Duelling DQN and Duelling DDQN in PyTorch for Project 1: Navigation:

In this project, I have developed four variations of Q learning techniques based on the methodology presented in Ref. [1], [2] and [3]. The techniques include Deep Q Learning (DQN), Double Deep Q Learning, Duelling DQN and Duelling Double Deep Q Learning. A brief background theory is given in Readme document in GitHup repository. The submission documents are located in DQN_variations folder (https://github.com/msfallah58/Navigation-project/tree/main/DQN_variations). I have written the codes using PyCharme IDE and have saved them in three modules namely: Navigation.py, agent_dqn.py and model.py. The Navigation module receives the input parameters and runs episode iterations for training the agent. It also saves the models and plot the results. agent_dqn module include two classes Agent () and ReplyBuffer(). Model.py module designs the network architecture for Q learning techniques. It includes Network() and Dueling_Network classes. The description of each model is given below:

Navigation.py:

First, we import the necessary libraries and modules as shown below:

```
import matplotlib.pyplot as plt
import numpy as np
import torch
from unityagents import UnityEnvironment

from agent_dqn import Agent
```

Here, the user needs to define which Q learning technique s/he wants to use. The available options are: DQN, DDQN, Dueling_DQN and Dueling_DDQN.

```
# TODO: Define the Q technique here
model_choice = "Dueling_DDQN"
```

Then we define the environment and agent functions as below:

```
agent_global = Agent(state_size=37, action_size=4,
model_choice=model_choice)

# TODO: update the path according to your local folder
environment = UnityEnvironment(
    file_name="/home/saber/deep-reinforcement-
learning/p1_navigation/Banana_Linux/Banana.x86_64", worker_id=0)
brain_name = environment.brain_names[0]
brain = environment.brains[brain_name]
```

Then, it is train_agent() function which its input variables are listed below:
env: the defined environment
agent: the Agent() class of agent_dqn module
n_episodes: the number of episodes to be run. The default value is 1000
eps_start: the initial epsilon value for greedy action selection. The default value is 1.0
eps_decay: the decay factor for decreasing epsilon. The default value is 0.995
eps_min: the minimum epsilon value. The default value is 0.01.

The function takes average of the rewards every 100 steps and print it on the screen. If the average reward is equal to 13 or more, the loop stops and then saves the model.

```
# Train the Agent with chosen Deep Q_learning Technique
def train_agent(env, agent, n_episodes=1000, eps_start=1.0,
eps_decay=0.995, eps_min=0.01):
    """The function trains an agent using DQN, Double DQN (DDQN), Dueling
DQN (Dueling DQN) and
    Dueling DDQN (Dueling_DDQN) techniques. The technique can be chosen
when the Agent class is called.

    Params
    =========
        n_episodes (int): maximum number of training episodes
        eps_start (float): starting value of epsilon for epsilon-greedy
        eps_min (float): minimum value of epsilon
        eps_decay (float): the decay factor for decreasing epsilon

    """
    eps = eps_start
    scores = []

    for i_episode in range(1, n_episodes + 1):
        state =
env.reset(train_mode=True)[brain_name].vector_observations[0]  # Get
initial state
        score = 0  # Initialise the score

        while True:
            action = agent.select_action(state, eps)  # Select action
            env_info = env.step(action)[brain_name]  # Send the action to
the environment
            next_state = env_info.vector_observations[0]  # Get the next
state
            reward = env_info.rewards[0]  # Get the reward
            done = env_info.local_done[0]  # See if episode has finished
            agent.step(state, action, reward, next_state, done)

            score += reward  # Update the score
            state = next_state  # Roll over the state to next time step
            if done:
                break  # If episodes end, stop
        scores.append(score)  # Add the score of the episode to scores
        eps = max(eps_min, eps_decay * eps)  # Apply epsilon decay
        average_size = 100
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
np.mean(scores[-average_size:])), end="")
        if i_episode % average_size == 0:
            print(
                '\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode -
average_size, np.mean(scores[-average_size:])))

        if np.mean(scores[-average_size:]) >= 13.0:
            if model_choice == "DQN":
                torch.save(agent.network_local.state_dict(),
'saved_network_DQN.pth')
                break
            elif model_choice == "DDQN":
                torch.save(agent.network_local.state_dict(),
'saved_network_DDQN.pth')
                break
            elif model_choice == "Dueling_DQN":
                torch.save(agent.network_local.state_dict(),
'saved_network_Dueling_DQN.pth')
                    break
                elif model_choice == "Dueling_DDQN":
                    torch.save(agent.network_local.state_dict(),
'saved_network_Dueling_DDQN.pth')
                    break
    return scores


scores = train_agent(environment, agent_global)
environment.close()
```

Finally, it plots the average reward per episode achieved by the agent.

```
fig_1 = plt.figure(1)
ax = fig_1.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)

if model_choice == "DQN":
    plt.title('DQN agent')
elif model_choice == "DDQN":
    plt.title('DDQN agent')
elif model_choice == "Dueling_DQN":
    plt.title('Dueling_DQN agent')
elif model_choice == "Dueling_DDQN":
    plt.title('Dueling_DDQN agent')

plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

model.py():

The imported libraries and methods of this module are listed below:

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

The module defines two classes for the neural network architecture. The first class named Network() designs the network for DQN and DDQN techniques while the second one named Dueling_Network designs the network for Duelling DQN and Duelling DDQN. Below is the description of each class:

Class Network(nn.Module):
The input to the class are the number of states (state_size) with default value of 37, the number of actions (action_size) with default value of 4 and the number of neurons of the first hidden layer (fc1_units) with the default value of 64 and the number of neurons of the second hidden layer (fc2_units) with the default value of 64. The layers are connected to each other and the activation functions are defined in forward() function of the class.

```
class Network(nn.Module):
    """Actor (Policy) model without advantage function"""

    def __init__(self, state_size=37, action_size=4, fc1_units=64,
fc2_units=64):
        """Initialise parameters and build model

        Params
        ======
            state_size (int): Dimension of state space
            action_size (int): Dimension of action space
            fc1_unit (int): Number of nodes in first hidden layer, default
is 128
            fc2_unit (int): Number of nodes in second hidden layer
        """
        super().__init__()
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state --> action values"""
        x = F.relu(F.dropout(self.fc1(state)))
        x = F.relu(F.dropout(self.fc2(x)))
        return self.fc3(x)
```

Class Dueling_Netwrk(nn.Modules):
The input to the class are the number of states (state_size) with default value of 37, the number of actions (action_size) with default value of 4 and the number of neurons of the first hidden layer (fc1_units) with the default value of 64 and the number of neurons of the second hidden layer (fc2_units) with the default value of 64, and the number of output value functions (output_feature) with the default value of 1. The layers are connected to each other and the activation functions are defined in forward() function of the class. Advantage layer and Value layers are defined as Adv and Val variables, respectively.

```
class Dueling_Network(nn.Module):
    """Actor (Policy) model without advantage function"""

    def __init__(self, state_size=37, action_size=4, fc1_units=64,
fc2_units=64, output_feature=1):
        """Initialise parameters and build model

        Params
        ======
            state_size (int): Dimension of state space
            action_size (int): Dimension of action space
            fc1_unit (int): Number of nodes in first hidden layer, default
is 128
            fc2_unit (int): Number of nodes in second hidden layer
        """

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

    def forward(self, state):
        """Build a network that maps state --> action values"""
        x = F.relu(F.dropout(self.fc1(state)))
        adv = F.relu(F.dropout(self.fc2_adv(x)))
        val = F.relu(F.dropout(self.fc2_val(x)))
        adv = self.fc3_adv(adv)
        val = self.fc3_val(val).exapnd(x.size(0), self.num_actions)

        x = val + adv - adv.mean(1).unsqueeze(1)
        return self.fc3(x)
```

<u>**agent_dqn()**</u>:

This module trains the agent through different functions that will be explained in the following. The imported libraries, modules and methods to the module are as shown below:

```
import random
from collections import namedtuple, deque

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

from model import Network, Dueling_Network
```

Then the hyperparameters are defined. The main hyper parameters are replay buffer size (BUFFER_SIZE), mini batch size (BTCH_SIZE), discount factor (GAMMA), soft update factor for updating the target network (TAU), learning rate (LR), and the frequency of updating the target network (UPDATE_EVERY). The selected opimiser for training is Adam. The values of the parameters are shown below:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64  # mini batch size
GAMMA = 0.99  # discount factor
TAU = 1e-3  # for soft update of target parameters
LR = 0.001
UPDATE_EVERY = 5
```

Class Agent():

The class defined the associated functions for training of the agent. The functions of the class are step(), select_action(), learn(), and soft_update(). The inputs to the class are the number of state (state_size), the number of actions (acion_size) and the selected Q learning technique (model_choice). The parameters of the class are listed below:

```
"""Interacts with and learns from environment"""

def __init__(self, state_size, action_size, model_choice):
    """
    :param model_choice: DQN or DDQN (string)
    :param state_size: size of state space (int)
    :param action_size: size of action space (int)
    """
    self.state_size = state_size
    self.action_size = action_size

    # Q-Network
    self.network_local = Network(state_size, action_size).to(device) if
```

```
model_choice == "DQN" or "DDQN" else \
        Dueling_Network(state_size, action_size).to(device)
    self.network_target = Network(state_size, action_size).to(device) if
model_choice == "DQN" or "DDQN" else \
        Dueling_Network(state_size, action_size).to(device)
    self.optimiser = optim.Adam(self.network_local.parameters(), lr=LR)
    self.model_choice = model_choice
    # Replay memory
    self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE)
    # Initialise time step (for updating every UPDATE_EVERY steps)
    self.t_step = 0
```

The first function of agent class is step() function which receives state, action, reward, next states, and done information. The function adds the inputs to the memory list, collects samples from memory and train the network.

```
def step(self, state, action, reward, next_state, done):
    # Save experience in replay memory
    self.memory.add(state, action, reward, next_state, done)

    # Learn every UPDATE_EVERY time steps
    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:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

The second function is select_action which receives current state of the environment and returns the selected action according to the greedy policy.

```
def select_action(self, state, eps):
    """
    Returns actions for given state as per policy
    :param state: current state (array_like)
    :param eps: epsilon (float), for epsilon-greedy action selection
    :return: selected action (int)
    """

    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    self.network_local.eval()
    with torch.no_grad():
        action_values = self.network_local(state)
    self.network_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))
```

The third function is learn() function which receives sampled trajectories (experiences) and the discount factor (gamma) to train the agent. The function first calculates values of the target network and expected Q function are calculated according to chosen Q learning technique. Then the function defines the loss value using Mean Square Error (MSE) function. Then the backpropagation and optimisation will be performed.

```
def learn(self, experiences, gamma):
    """
    Update value parameters using given batch of experience tuples

    :param experiences: (Tuple[torch.Variable]): tuple of (s, a, r, s',
done)
    :param gamma: discount factor (float)
    :return:
    """

    global q_targets
    states, actions, rewards, next_states, dones = experiences

    # Get max predicted Q values (for next states) from target model
    if self.model_choice == "DQN" or "Dueling_DQN":
        q_targets_next =
self.network_target(next_states).detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        q_targets = rewards + (gamma * q_targets_next * (1 - dones))

    else:
        _, maximising_actions =
torch.max(self.network_local(next_states).detach(), dim=1)
        maximising_actions = torch.unsqueeze(maximising_actions, 1)
        q_targets = self.network_target(next_states).gather(1,
maximising_actions)
        q_targets = rewards + (gamma * q_targets) * (1 - dones)

    # Get expected Q values from local model
    q_expected = self.network_local(states).gather(1, actions)
    # Compute loss
    self.optimiser.zero_grad()
    loss = F.mse_loss(q_targets, q_expected)

    # Minimise the loss
    loss.backward()
    self.optimiser.step()

    # update target network
```

The last function of Agent() class is soft_update() that receives network of Q function (local_model) and the target function (target_model) and update factor (tau) and updates the target network. The soft update was proposed in [4]

```
def soft_update(self, local_model, target_model, tau):
    """
    soft update model parameters
    theta_target = tau*theta_local + (1-tau)*theta_target

    :param local_model: weights will be copied from (Pytorch model)
    :param target_model: weights will be copied to (Pytorch model)
    :param tau: interpolation parameter (float)
    :return:
    """

    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)
```

The second class of the agent_dqn module is ReplayBuffer(). The input of the class are the number of actions (action_size), the size of buffer (buffer_size) and the mini batch size (batch_size). The parameters of the class are listed below:

```
def __init__(self, action_size, buffer_size, batch_size):
    """
    Initialise a ReplayBuffer object

    :param action_size: dimension of action space (int)
    :param buffer_size: maximum size of buffer (int)
    :param batch_size: size of each training batch (int)
    """

    self.action_size = action_size
    self.memory = deque(maxlen=buffer_size)
    self.batch_size = batch_size
    self.experience = namedtuple("Experience", field_names=["state",
"action", "reward", "next_state", "done"])
```

The first function of the ReplayBuffer class is add() function which add current state, action, reward, next state and done information to the memory list.

```
def add(self, state, action, reward, next_state, done):
    """Add a new experience to memory"""
    e = self.experience(state, action, reward, next_state, done)
    self.memory.append(e)
```

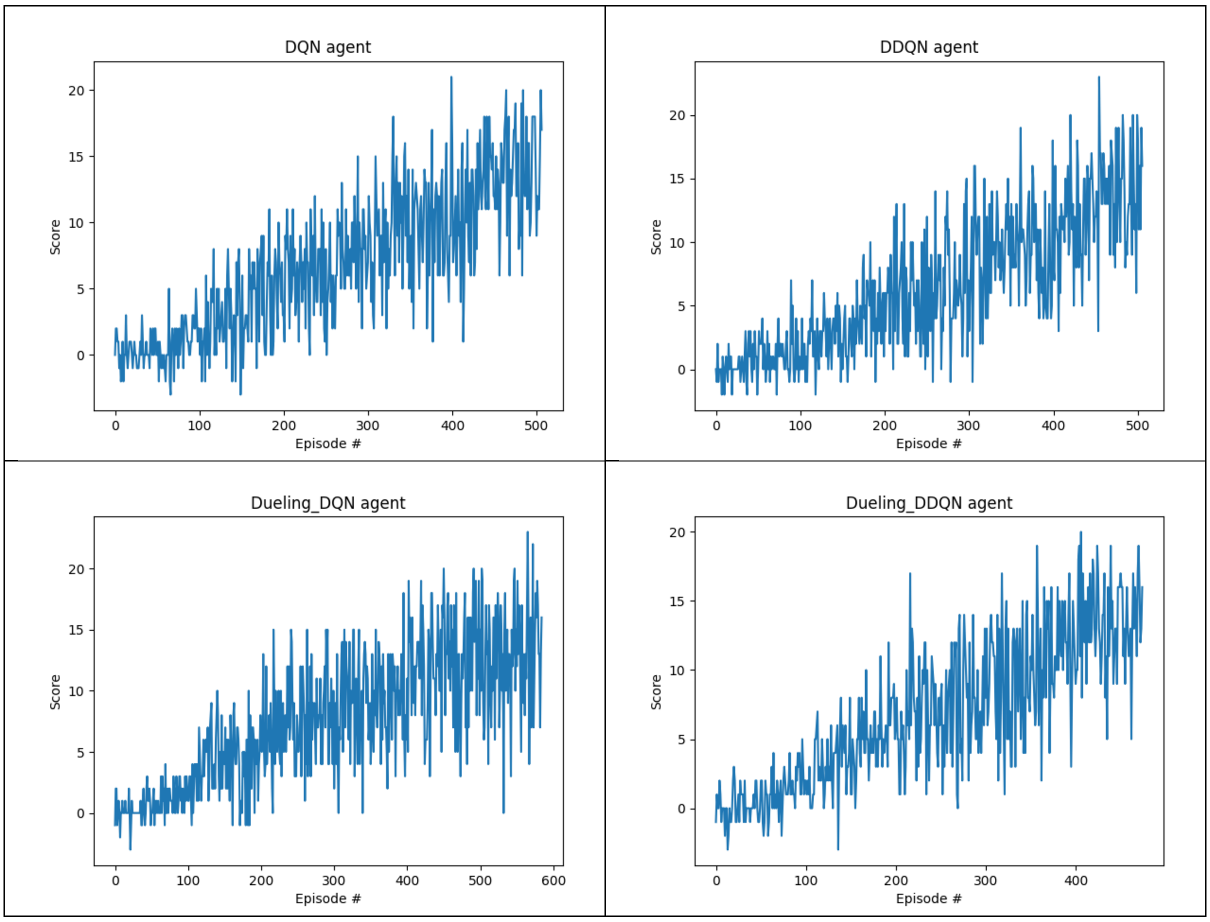
The second function is sample() which samples data and convert them to tensor format.

```
def sample(self):
    experiences = random.sample(self.memory, k=self.batch_size)

    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
```

The last function is __len__ which calculates the size of memory list.

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

Results:
The performance of four Q learning techniques are investigated through running the code for 1000 episodes. The results show that DQN reach to average reward of 13 after 514, DDQN after 513, Duelling DQN after 536 and Duelling DDQN after 475. It is observed Duelling DDQN has the best performance. Although Deulling DQN takes longer to converge but it reach to higher average reward at earlier episodes compared to DQN and DDQN techniques which means the algorithm start learning at faster speed at early training process but slower later. The results are shown below:



Future Ideas:
Since Duelling Double DQN has shown the best performance, it is wise to extend this technique using the other important extension of DQN. The Noisy Nets and Prioritise Reply are suggested as the further extension of Duelling Double DQN due to their contribution to the improvement of exploration and learning process of Q-learning, respectively.

Noisy Nets:
Classical DQN achieves exploration by choosing random actions with a specially defined hyperparameter epsilon, which is slowly decreased over time from 1.0 (fully random actions) to some small ratio of 0.1 or less. This process works well for simple environments with short episodes but even in such simple cases, it requires tuning to make the training processes efficient. In the Noisy Networks paper [5], the authors proposed a quite simple solution that, nevertheless, works well. They add noise to the

weights of fully connected layers of the network and adjust the parameters of this noise during training using backpropagation.

Prioritise Reply:
The DQN, Double DQN, Duelling Double DQN that were developed in this project use the replay buffer to break the correlation between immediate transitions in episodes. However, when the environment is "smooth" and doesn't change much according to actions the experiences collected from episodes will be highly correlated. However, the stochastic gradient descent (SGD) method assumes that the data used for training has an independent and identically distributed property. One solution to this problem is to use a large buffer of transitions, randomly sampled to get the next training batch. The authors of the paper [6] questioned this uniform random sample policy and proved that by assigning priorities to buffer samples, according to training loss and sampling the buffer proportional to those priorities, we can significantly improve convergence and the policy quality of the DQN.

References:
[1] https://www.nature.com/articles/nature14236
[2] https://arxiv.org/abs/1509.06461
[3] https://arxiv.org/abs/1511.06581
[4] https://arxiv.org/abs/1509.02971
[5] https://arxiv.org/abs/1706.10295
[6] https://arxiv.org/abs/1511.05952