Report of Coursework2

Name: **Ziyue Tong**

CID: **02077487**

Department: **Bioengineering**

Course: **Human and Biological Robotics**

Contents

Question 1: Implementing a functional DQN

Question 1.1 Three key technical elements

Three vital technical elements are achieved in the code shown in appendix 1: replay buffer, target network, and multiple frames input.

Question 1.2 Brief description

Referred to the code in appendix 1, to implement replay buffer (line 74 to 87), a class from related code provided in the template of this coursework was used after cross-comparison with the tutorial from PyTorch and multiple tests. The target network was also used based on the template provided (line 93 to 108). For multiple frame input (line 199), a library called 'gym.wrappers.frame_stack' was imported to complete inputting stack of frames to network.

Question 1.3 Design decision and hyper parameters of NN

The main design for the DQN program is to solve the Cartpole problem, which is classic in control engineering. Therefore, the rationality of designing is to provide the neural network with sufficient information to control the cart pole. In this way, the network should be trained to fit the performance of a controller. For example, a PID controller can be built by handling the information from several previous samples when being implemented. The neural network applied for fitting the performance of a controller would be presented below.

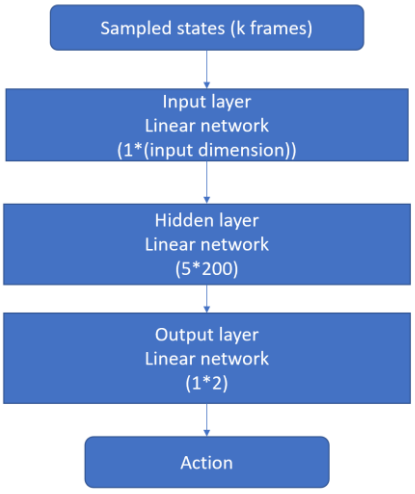

Figure 1. The structure of neural network implemented, note that the number of input dimension is adjustable (in appendix 1, the value is 4).

A stack was designed to provide sufficient input (several frames), enabling the network to evaluate the motion trend of the cart pole for better performance. Initially, a function was designed as a stack for storing frames that includes the newest state and several past states. However, this method encountered several problems due to data types and sizes, so finally, this function was implemented based on a function called FrameStack. With the imported

function, several functions can be recorded and be sent to a neural network for training purposes.

The settings of hyperparameters were also based on the logic that the network is fitting the transfer function of the controller. A hyperparameter called ‘INTEGRATED’ was designed to define the number of states sampled from the environment in one input. In the code in appendix 1, the value was set to 4, meaning four states would be sampled and sent to the network. The input under this condition was set with 16 input and 200 output. The hidden layer was set to 200 neurons and with a depth of 5 layers. The reason for setting each layer in that way was to allow the network with sufficient redundancy to weigh the importance of each input. With a depth of 5 layers, high order patterns of the system can be possibly fitted and controlled since the core of the cart pole problem is implementing linear control to a non-linear system.

Question 1.4 learning curves

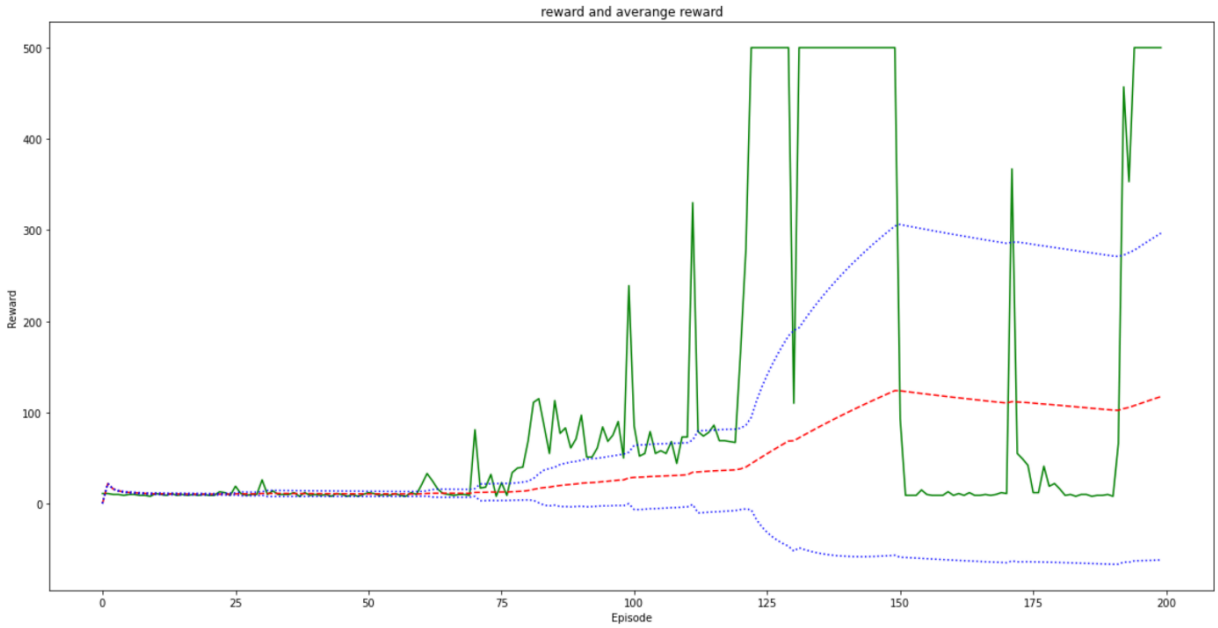

Figure 2. The learning curve of the model, green line for the reward in each episode, red dash line for the mean of the reward, and blue dot line for the mean ± standard deviation of reward.

There are 200 episodes iterated for training purposes. This is because that the neural network can be better trained with a large episode number since each neuron within the network can be effectively updated. Another reason is that the training was conducted with a high-performance GPU so that duration of training can be relatively acceptable. Moreover, with a large amount of training, some commonly seen phenomena can be observed. For example, ‘catastrophic forgetting’ can occur after many episodes. This phenomenon can be probabilistically optimized by changing the number of input frames, target network update frequency, and network structure.

During the episodes from 125 to 150 and last 5 episodes, the average total return of the DQN achieved roughly 90% of the final performance (the highest reward observed through many tests is 500, reaching the reward or duration the environment would stop and reset).

Question 2: Hyperparameters of the DQN

Question 2.1 Investigation about exploration parameter epsilon ϵ

Compared with what was presented for the model with a constant ϵ, the model with dynamic ϵ is relatively more stabilized (although there are still oscillations occurred).

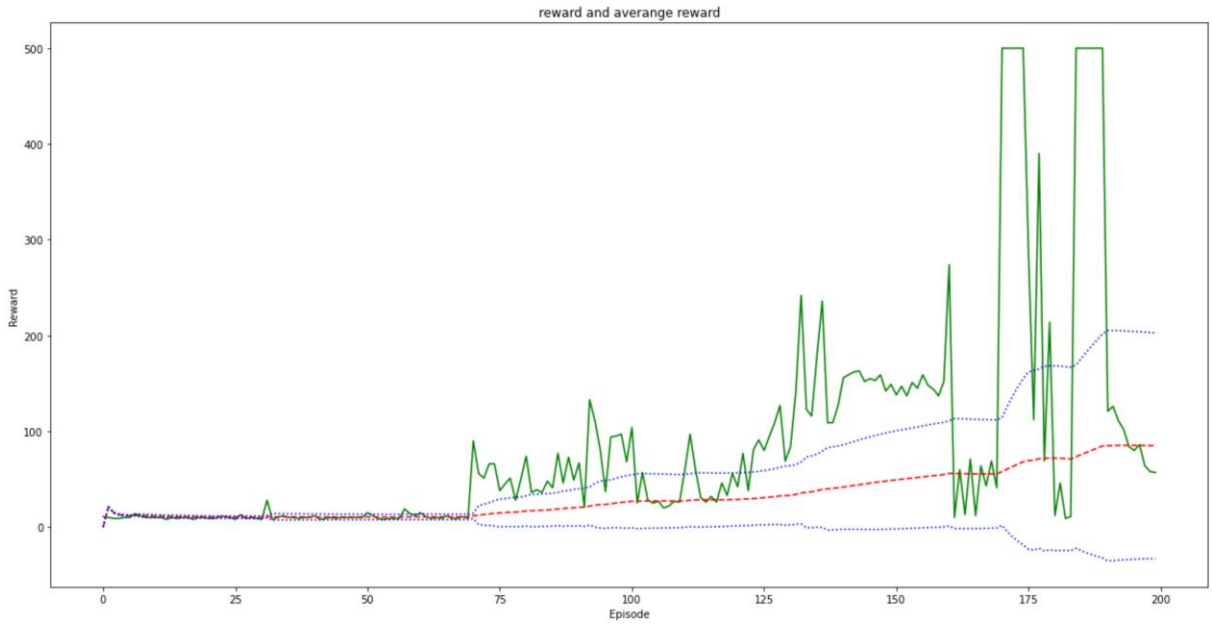

Figure 3. The learning curve of the model with dynamic ϵ, green line for the reward, red dash line for the mean of the reward, and blue dot line for the mean ± standard deviation of reward.

The exploration parameter ϵ regulates the tendency for exploration or optimization. Multiple tests discovered that situations with a small ϵ would have less oscillation and be maintained at either low reward or nearly maximal reward. Therefore, small ϵ in this scenario would change the agent more to optimize with acquired information instead of exploring. The value of ϵ can affect the result of learning significantly. With ϵ overly large or overly small would cause more severe oscillation on the learning curve and more frequent forgetting.

With a changing exploration parameter (Dynamic ϵ), the learning curve should be similar to that of large ϵ at the beginning of training, which oscillates severely and achieves large peak reward values. The pattern should be similar to that of smaller ϵ, which is relatively stable. However, it can be observed from the figure below that before the 125th episode, the model with dynamic ϵ is working as expected, but afterwards, a forgetting occurred, and the reward started oscillating. Therefore, models with dynamic ϵ can reach a high reward level and stabilize faster than that with constant ϵ. This process can be logically explained as the model can thoroughly

learn the rules of the environment and then concentrate on optimizing. In the figure below, a learning curve with different constant and dynamic exploration parameters will be presented.

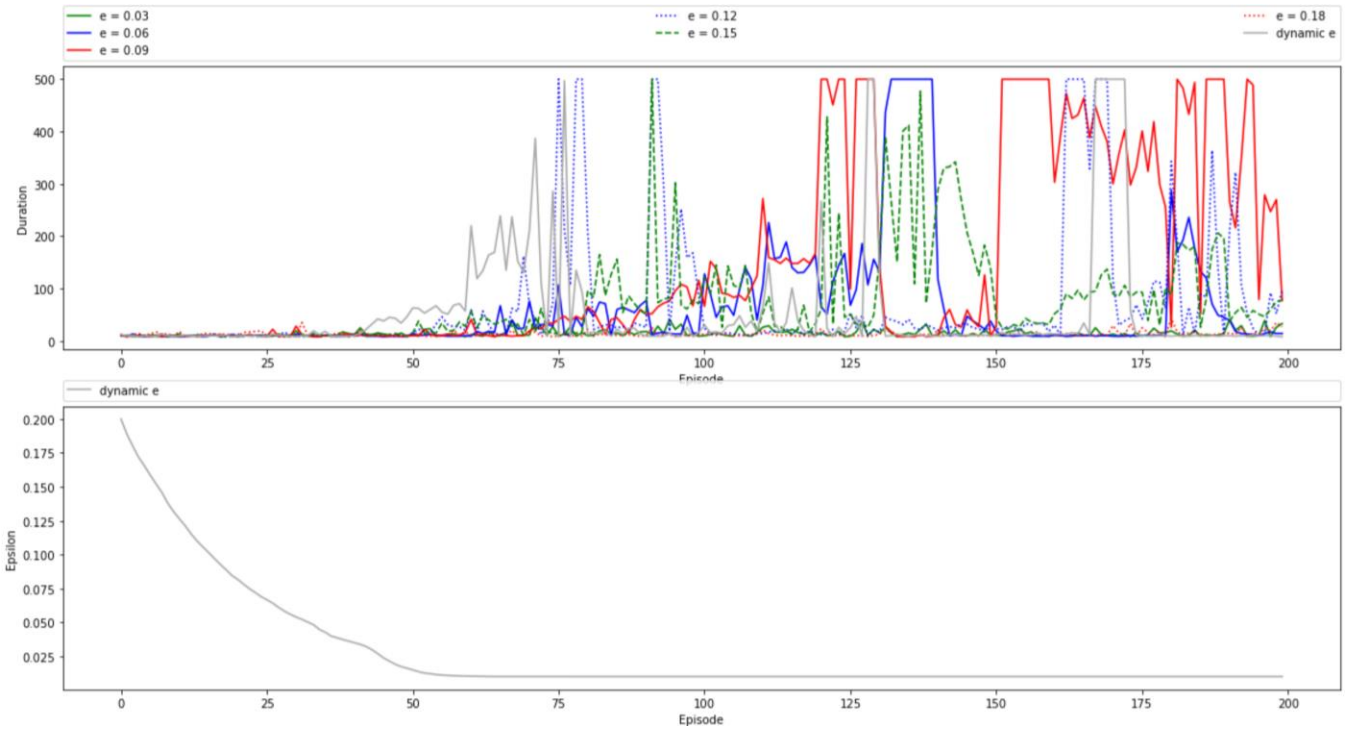

Figure 4. Learning curve of the DQN with different constant and dynamic exploration parameters (upper plot), and the changes of exploration parameter (lower plot)

Question 2.2 Investigation about the size of replay buffer

The size of the replay buffer would determine the number of experiences to be reused in training. With a large replay buffer, the random selections of batches of samples can be further decorrelated, which helps remove noises when training the network (Pytorch, 2021). The model was trained with different sizes of replay buffer. Theoretically, if the reply buffer is small, the neural network would be significantly affected by the noises, including the function used to generate a random selection of batches.

Though many tests toward different buffer sizes, it can be derived that the mean value of the reward is tending to be increasing, and the standard deviation of reward is tending to be stabilized as the sizes of buffer increase. Particularly, when the size is 300, which is slightly larger than the size of the sampling batch, the mean reward was maintained at a low level (so the standard deviation is small), which means that the model is severely affected by noises according to the derivation in the previous text. The eventual result can be caused by the limitation of episode number (200) and sampled batch size, so the space in the buffer was not fully used. The standard deviation to mean reward with varying sizes of replay buffers (300, 1300, 2300, 3300, 10000) would be presented in the figure below.

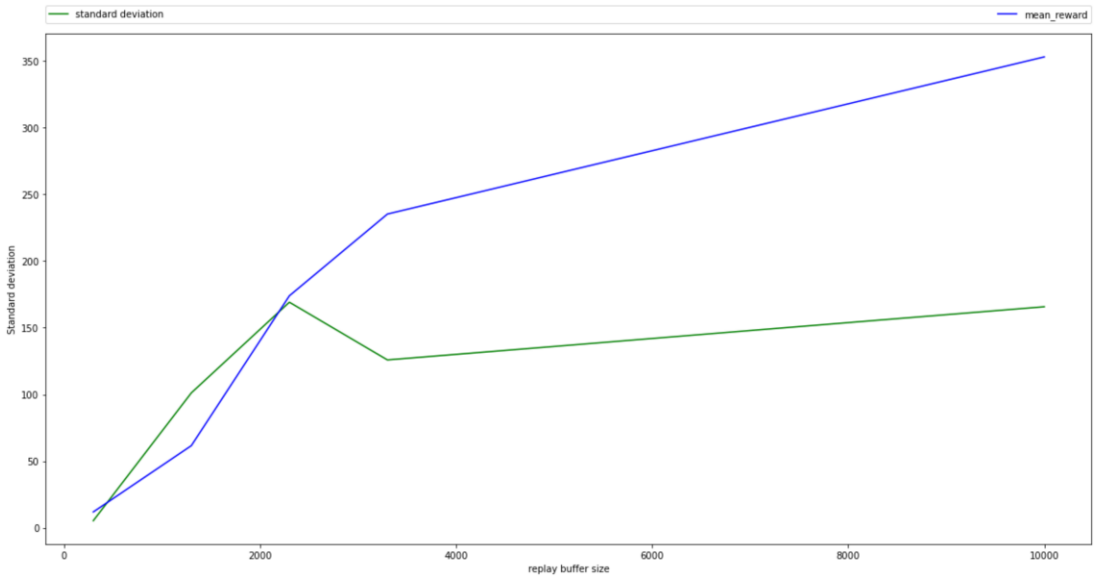

Figure 5. The standard deviation and mean reward of the model with different buffer size

Question 2.3 Investigation about the frames input

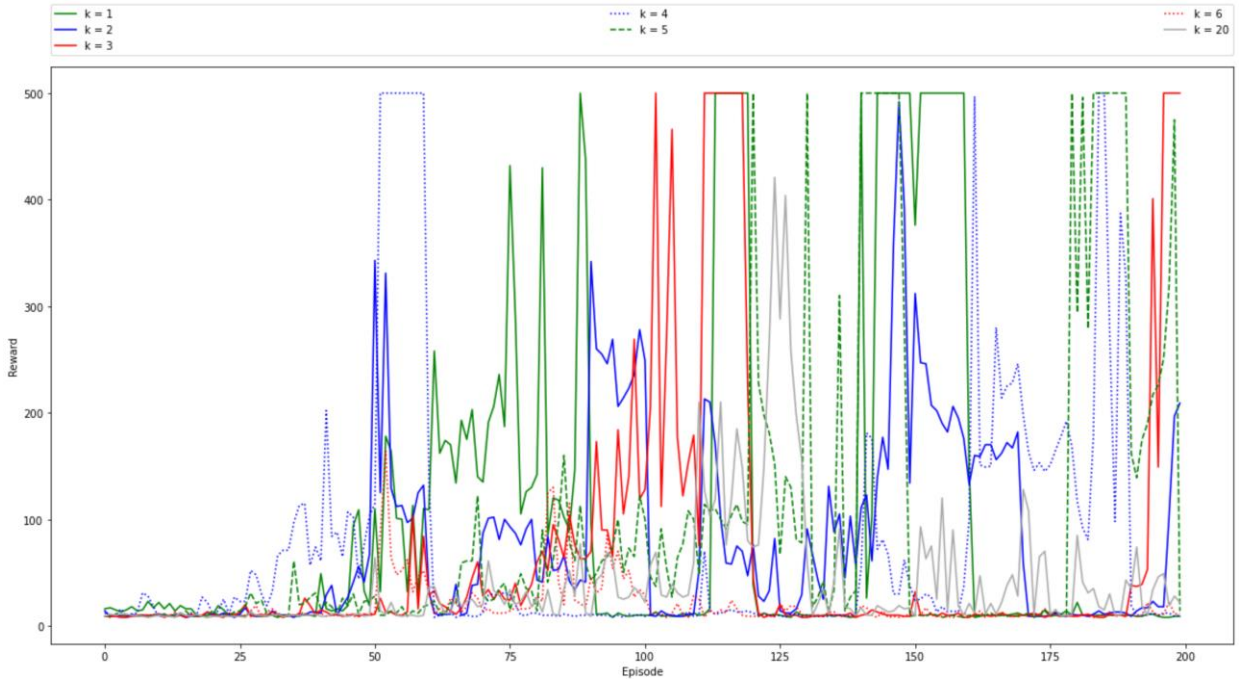

Figure 6. The learning curves of the model with different frames input

It can be observed from above that as the number of frames increases, the reward level decreases, and it may not reach the maximum reward with many frame inputs. The result was acquired by only changing the number of input frames, which means that the current neural network may not be sufficient to handle the data input. Additionally, in the original code at line 256, the frames were acquired by repeat receiving environment states several times, which

would cause a lag between the changes in environment and actions selected. The lag would be increasingly significant as the receiving frames number becomes larger. Since controlling the pole is a relatively simple scenario, it can be well controlled with only one data frame.

Question 3: Ablation/Augmentation experiments

Question 3.1 DDQN implementation

The double DQN was implemented based on a logic that uses both policy net and target net to calculate estimated the Q value of the next state (van Hasselt et al., 2016). A new network called 'D_policy_net' was created to increase the accuracy of estimation probabilistically. In the code, the new network was used to estimate the next action, which would return a tensor indicating the index of the following action. Afterwards, the current state was sent to the target network to estimate the values of actions. By applying the tensor returned by the new network, the Q value (action-value function) can be filtered and returned for computing loss to update the policy network (eventually target network). The implementation of DDQN is presented in appendix 3, mainly line 154 to line 163.

Question 3.2 Investigation about target network, replay buffer, and DDQN

The DQN model without a target network would become much unstable, and the average reward is less than that with the target network. Since the policy net is frequently updated, the loss function can be significantly disrupted by noises. The interference is becoming severely as the grown of episode number. Hence the bootstrapping estimations of Q values become highly distorted. Thus, the actions taken would not be the same as expected.

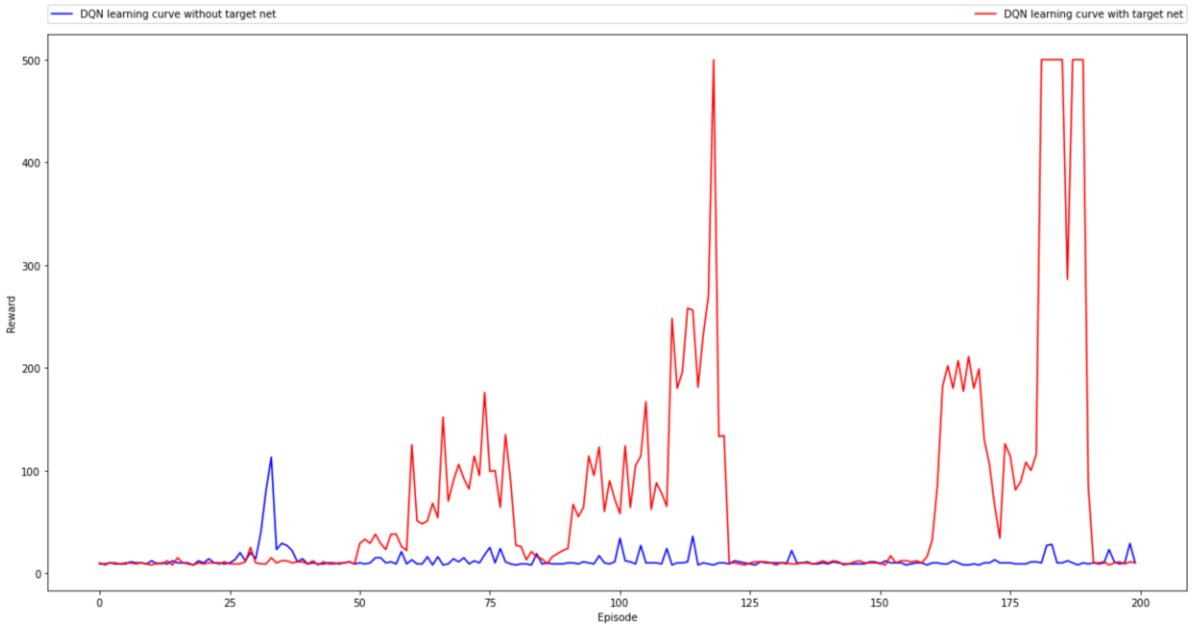

Figure 7. The learning curve of DQN without and with target network

The replay buffer is the precondition for training the DQN model with batches of randomly sampled experiences. Therefore, without a replay buffer, the model cannot decorrelate the input

data, which would lead to severe interference caused by noises. The resulted contrast learning curve would be presented below. Thus, the model cannot be trained as desired.

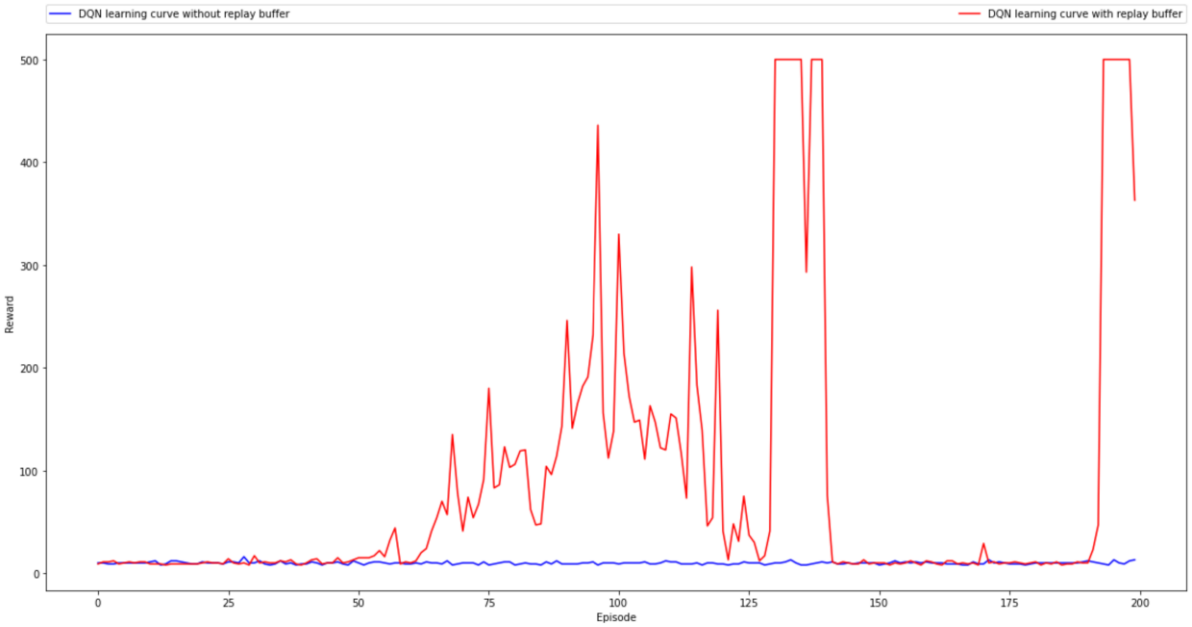

Figure. 8 The learning curve of DQN with and without replay buffer

Theoretically, the DQN can be enhanced by using the DDQN model since the estimation of the Q value of the following state has been optimized. Compared with figure 2 shown in section 1.4, the learning curve of DDQN reached maximum reward faster. However, since there is another network involved in training, the episodes number required to optimize the model also increased, and the phenomenon of forgetting had become more severe since many neural networks were being updated.

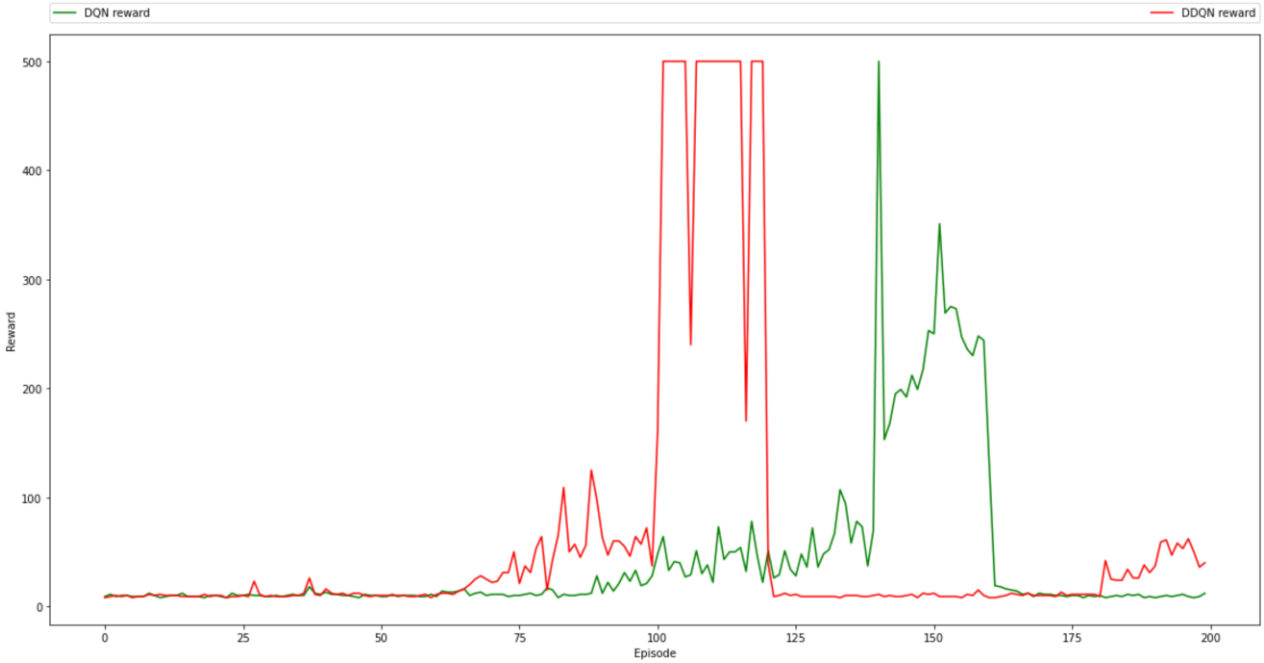

Figure. 9 The learning curve of DQN and DDQN

Reference

Pytorch. (2021). *REINFORCEMENT LEARNING (DQN) TUTORIAL*. Https://Pytorch.Org/Tutorials/Intermediate/Reinforcement_q_learning.Html.

van Hasselt, H., Guez, A., & Silver, D. (2016). *Deep Reinforcement Learning with Double Q-Learning*. www.aaai.org

Appendix

Appendix 1. DQN code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

# This is the coursework 2 for the Reinforcement Leaning course 2021 taught at Imperial College London (https://www.imperial.ac.uk/computing/current-students/courses/70028/)
# The code is based on the OpenAI Gym original (https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html) and modified by Filippo Valdettaro and Prof. Aldo Faisal for the purposes of the course.
# There may be differences to the reference implementation in OpenAI gym and other solutions floating on the internet, but this is the defeinitive implementation for the course.

# Instaling in Google Colab the libraries used for the coursework
# You do NOT need to understand it to work on this coursework

# !pip install gym

from IPython.display import clear_output
clear_output()

# In[2]:

# Importing the libraries

import gym
from gym.wrappers.monitoring.video_recorder import VideoRecorder    #records videos of episodes
import numpy as np
import matplotlib.pyplot as plt # Graphical library

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as T
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # get RTX3080 Ready

from collections import namedtuple, deque
from itertools import count
import math
import random
from gym.wrappers.frame_stack import FrameStack

clear_output()

# In[3]:

# Test cell: check ai gym  environment + recording working as intended

env = gym.make("CartPole-v1")
file_path = './video_test.mp4'
recorder = VideoRecorder(env, file_path)

observation = env.reset()
terminal = False
while not terminal:
  recorder.capture_frame()
  action = int(observation[2]>0)
  observation, reward, terminal, info = env.step(action)
  # Observation is position, velocity, angle, angular velocity

recorder.close()
env.close()

# In[4]:

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

# defining replay buffer
class ReplayBuffer(object):
```

```
    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

# In[5]:

class DQN(nn.Module):

    def __init__(self, inputs, outputs, num_hidden, hidden_size):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(inputs, hidden_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size) for _ in range(num_hidden-1)])
        self.output_layer = nn.Linear(hidden_size, outputs)

    def forward(self, x):
        x.to(device)

        x = F.relu(self.input_layer(x))
        for layer in self.hidden_layers:
            x = F.relu(layer(x))

        return self.output_layer(x)

# In[6]:


def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    # setting the imput of mini-batches
    transitions = memory.sample(BATCH_SIZE)

    batch = Transition(*zip(*transitions)) # seperate state action reward

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool) # transform the array of states into tensor so that GPU
can handle


    # Can safely omit the condition below to check that not all states in the
    # sampled batch are terminal whenever the batch size is reasonable and
    # there is virtually no chance that all states in the sampled batch are
    # terminal
    if sum(non_final_mask) > 0:
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
    else:
        non_final_next_states = torch.empty(0,state_dim).to(device)

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)

    with torch.no_grad(): #there will be a backward() called later
        # Once again can omit the conditional if batch size is large enough
        if sum(non_final_mask) > 0:
            next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
        else:
            next_state_values = torch.zeros_like(next_state_values)

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute loss
    loss = ((state_action_values - expected_state_action_values.unsqueeze(1))**2).sum()

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Limit magnitude of gradient for update step
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
```

```
    optimizer.step()

# In[7]:

# Training
#setting parameters
NUM_EPISODES = 200
BATCH_SIZE = 256
GAMMA = 0.99
MEMORY_CAPACITY = 5000
INTEGRATED = 4 # number of previous states
# threshold_duration = 1000 this is not needed, the duration never exceed 500

epsilon = 0.09 # for epsilon greedy policy
num_hidden_layers = 5
size_hidden_layers = 200
target_update_frequency = 10

# Get number of states and actions from gym action space
env = gym.make("CartPole-v1")
env = FrameStack(env, INTEGRATED)
env.reset()
state_dim = len(env.state) * INTEGRATED   #states are: x, x_dot, theta, theta_dot

n_actions = env.action_space.n
env.close()

policy_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
target_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())

memory = ReplayBuffer(MEMORY_CAPACITY)

def select_action(state, current_eps=0):

    sample = random.random()
    eps_threshold = current_eps
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.

            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)

# In[8]:


file_path_2 = './video_train.mp4'
recorder = VideoRecorder(env, file_path_2)

observation = env.reset()
done = False

state = torch.tensor(env.state).float().unsqueeze(0)

duration_record = np.zeros(NUM_EPISODES)
reward_record = np.zeros(NUM_EPISODES)

#training
for i_episode in range(NUM_EPISODES):
    duration = 0 #reset duration
    acc_r = 0
    if i_episode % target_update_frequency == 0:
        print("episode ", i_episode, "/", NUM_EPISODES)

    # Initialize the environment and state
    env.reset()
    state = torch.tensor(env.state).float().unsqueeze(0).to(device)
    state = state.repeat(1, INTEGRATED)

    for t in count():
        recorder.capture_frame()
        # Select and perform an action
        action = select_action(state, epsilon)

        states, reward, done, _ = env.step(action.item())
        frames_1d = np.array(states).flatten()

        reward = torch.tensor([reward], device=device)
        duration += 1
        r = 1
        acc_r = acc_r + r # to gain total reward

        # Observe new state
        if not done:
            next_state = torch.tensor(frames_1d).float().unsqueeze(0).to(device)
        else:
            next_state = None

        # Store the transition in memory
```

```
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        # update target net
        if i_episode % target_update_frequency == 0:
           target_net.load_state_dict(policy_net.state_dict()) #transfer the data in policy net to target net
        if done:
            duration_record[i_episode] = duration
            reward_record[i_episode] = acc_r
            break
    recorder.close()
    env.close()
print("Episode duration: ", duration)


print('Complete')

# duration analysis
# print("duration recorder shows: ", duration_record)
avg_duration = np.zeros(NUM_EPISODES)
acc_duration = 0
for i in range(NUM_EPISODES):
    acc_duration += duration_record[i]
    if i == 0:
        avg_duration[i] = 0
    else:
        avg_duration[i] = acc_duration / i
avg_reward = np.zeros(NUM_EPISODES)
std_reward = np.zeros(NUM_EPISODES)
mix_n = np.zeros(NUM_EPISODES)
mix_p = np.zeros(NUM_EPISODES)
acc_reward = 0
for i in range(NUM_EPISODES):
    acc_reward += reward_record[i]
    std_reward[i] = np.std(reward_record[0:i])
    if i == 0:
        avg_reward[i] = 0
    else:
        avg_reward[i] = acc_reward / i
        mix_p[i] = avg_reward[i] + std_reward[i]
        mix_n[i] = avg_reward[i] - std_reward[i]

# print("duration recorder shows: ", avg_duration)
plt.figure(1, figsize=(20,10))
plt.title('Durations and averange durations')
plt.xlabel('Episode')
plt.ylabel('Duration')
plt.plot(duration_record,color='green', linestyle='-')
plt.plot(avg_duration,color='red', linestyle=':')

plt.figure(2, figsize=(20,10))
plt.title('reward and averange reward')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.plot(reward_record,color='green', linestyle='-')
plt.plot(avg_reward,color='red', linestyle='--')
plt.plot(mix_p,color='blue', linestyle=':')
plt.plot(mix_n,color='blue', linestyle=':')

recorder.close()
env.close()
print("Episode duration: ", duration)
```

Appendix 2. Code for changing parameters of DQN learning

Note that the denoted part is implementing dynamic epsilon input.

```
#!/usr/bin/env python
# coding: utf-8

# In[11]:

# This is the coursework 2 for the Reinforcement Leaning course 2021 taught at Imperial College London (https://www.imperial.ac.uk/computing/current-students/courses/70028/)
# The code is based on the OpenAI Gym original (https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html) and modified by Filippo Valdettaro and Prof. Aldo Faisal for the purposes of the course.
# There may be differences to the reference implementation in OpenAI gym and other solutions floating on the internet, but this is the defeinitive implementation for the course.

# Instaling in Google Colab the libraries used for the coursework
# You do NOT need to understand it to work on this coursework

# get_ipython().system('pip install gym')

from IPython.display import clear_output
clear_output()

# In[12]:
```

```
# Importing the libraries

import gym
from gym.wrappers.monitoring.video_recorder import VideoRecorder    #records videos of episodes
import numpy as np
import matplotlib.pyplot as plt # Graphical library

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as T
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # get RTX3080 Ready

from collections import namedtuple, deque
from itertools import count
import math
import random
from gym.wrappers.frame_stack import FrameStack

clear_output()

# In[13]:

# Test cell: check ai gym  environment + recording working as intended

env = gym.make("CartPole-v1")
file_path = 'F:/COMP97144 Reinforcement learning/cw2/videos/video.mp4'
recorder = VideoRecorder(env, file_path)

observation = env.reset()
terminal = False
while not terminal:
  recorder.capture_frame()
  action = int(observation[2]>0)
  observation, reward, terminal, info = env.step(action)
  # Observation is position, velocity, angle, angular velocity

recorder.close()
env.close()

# In[14]:

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

# defining replay buffer
class ReplayBuffer(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

# In[15]:

class DQN(nn.Module):

    def __init__(self, inputs, outputs, num_hidden, hidden_size):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(inputs, hidden_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size) for _ in range(num_hidden-1)])
        self.output_layer = nn.Linear(hidden_size, outputs)

    def forward(self, x):
        x.to(device)

        x = F.relu(self.input_layer(x))
        for layer in self.hidden_layers:
            x = F.relu(layer(x))

        return self.output_layer(x)

# In[16]:


def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    # setting the imput of mini-batches
    transitions = memory.sample(BATCH_SIZE)

    batch = Transition(*zip(*transitions)) # seperate state action reward

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
```

```
                                          batch.next_state)), device=device, dtype=torch.bool) # transform the array of states into tensor so that GPU
can handle


    # Can safely omit the condition below to check that not all states in the
    # sampled batch are terminal whenever the batch size is reasonable and
    # there is virtually no chance that all states in the sampled batch are
    # terminal
    if sum(non_final_mask) > 0:
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
    else:
        non_final_next_states = torch.empty(0,state_dim).to(device)

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)

    with torch.no_grad(): #there will be a backward() called later
        # Once again can omit the conditional if batch size is large enough
        if sum(non_final_mask) > 0:
            next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
        else:
            next_state_values = torch.zeros_like(next_state_values)

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute loss
    loss = ((state_action_values - expected_state_action_values.unsqueeze(1))**2).sum()

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Limit magnitude of gradient for update step
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()

# In[17]:

# Training

# seed to regulate the change in different epsilon
NUM_EPISODES = 200
central_record = np.zeros([len(range(1,6)), NUM_EPISODES])
size_record = np.zeros([len(range(1,6))])

for seed in range(1,6):
    #setting parameters
    # NUM_EPISODES = 100
    BATCH_SIZE = 256
    GAMMA = 0.99
    MEMORY_CAPACITY = 300 + (seed-1) * 1000
    INTEGRATED = seed # number of previous states
    if seed == 5:
        MEMORY_CAPACITY = 10000
    # threshold_duration = 1000 this is not needed, the duration never exceed 500
    size_record[seed - 1] = MEMORY_CAPACITY

    epsilon = 0.09 # for epsilon greedy policy
    num_hidden_layers = 5
    size_hidden_layers = 200
    target_update_frequency = 10

    # Get number of states and actions from gym action space
    env = gym.make("CartPole-v1")
    env = FrameStack(env, INTEGRATED)
    env.reset()
    state_dim = len(env.state) * INTEGRATED   #states are: x, x_dot, theta, theta_dot

    n_actions = env.action_space.n
    env.close()

    policy_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
    target_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()

    optimizer = optim.RMSprop(policy_net.parameters())

    memory = ReplayBuffer(MEMORY_CAPACITY)

    def select_action(state, current_eps=0): #, current_eps=0):
```

```
        global steps_done
        sample = random.random()
        eps_threshold = current_eps

        if sample > eps_threshold:
            with torch.no_grad():
                # t.max(1) will return largest column value of each row.
                # second column on max result is index of where max element was
                # found, so we pick action with the larger expected reward.

                return policy_net(state).max(1)[1].view(1, 1)
        else:
            return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)

    file_path_2 = 'F:/COMP97144 Reinforcement learning/cw2/videos/video_1.mp4'
    recorder = VideoRecorder(env, file_path_2)

    observation = env.reset()
    done = False

    state = torch.tensor(env.state).float().unsqueeze(0)

    duration_record = np.zeros(NUM_EPISODES)
    reward_record = np.zeros(NUM_EPISODES)

    #training
    for i_episode in range(NUM_EPISODES):
        duration = 0 #reset duration
        acc_r = 0
        if i_episode % target_update_frequency == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.state).float().unsqueeze(0).to(device)
        state = state.repeat(1, INTEGRATED)

        for t in count():
            recorder.capture_frame()
            # Select and perform an action
            action = select_action(state, epsilon) #, epsilon)

            states, reward, done, _ = env.step(action.item())
            frames_1d = np.array(states).flatten()

            reward = torch.tensor([reward], device=device)
            duration += 1
            r = 1
            acc_r = acc_r + r # to gain total reward

            # Observe new state
            if not done:
                next_state = torch.tensor(frames_1d).float().unsqueeze(0).to(device)
            else:
                next_state = None

            # Store the transition in memory
            memory.push(state, action, next_state, reward)

            # Move to the next state
            state = next_state

            # Perform one step of the optimization (on the policy network)
            optimize_model()
            # update target net
            if i_episode % target_update_frequency == 0:
                target_net.load_state_dict(policy_net.state_dict()) #transfer the data in policy net to target net
            if done:
                duration_record[i_episode] = duration
                reward_record[i_episode] = acc_r
                break


    central_record[seed-1,:] = duration_record
    print('Complete')
    recorder.close()
    env.close()
    print("Episode duration: ", duration)

####################   Dynamic E
# #setting parameters
# BATCH_SIZE = 256
# GAMMA = 0.99
# MEMORY_CAPACITY = 5000
# INTEGRATED = 4 # number of previous state

# num_hidden_layers = 5
# size_hidden_layers = 200
# target_update_frequency = 10

# EPS_START = 0.2
# EPS_END = 0.01
# EPS_DECAY = NUM_EPISODES

# # Get number of states and actions from gym action space
# env = gym.make("CartPole-v1")
```

```
# env = FrameStack(env, INTEGRATED)
# env.reset()
# state_dim = len(env.state) * INTEGRATED   #states are: x, x_dot, theta, theta_dot

# n_actions = env.action_space.n
# env.close()

# policy_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
# target_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
# target_net.load_state_dict(policy_net.state_dict())
# target_net.eval()

# optimizer = optim.RMSprop(policy_net.parameters())

# memory = ReplayBuffer(MEMORY_CAPACITY)
# eps_record = np.zeros(NUM_EPISODES)
# steps_done = 0
# def select_action(state): #, current_eps=0):
#     global steps_done
#     sample = random.random()
#     eps_threshold = EPS_END + (EPS_START - EPS_END) * \
#         math.exp(-1. * steps_done / EPS_DECAY)
#     # if steps_done < NUM_EPISODES:
#     #     eps_record[steps_done] = eps_threshold
#     steps_done += 1
#     if sample > eps_threshold:
#         with torch.no_grad():
#             # t.max(1) will return largest column value of each row.
#             # second column on max result is index of where max element was
#             # found, so we pick action with the larger expected reward.

#             return policy_net(state).max(1)[1].view(1, 1)
#     else:
#         return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)

# file_path_2 = 'F:/COMP97144 Reinforcement learning/cw2/videos/video_1.mp4'
# recorder = VideoRecorder(env, file_path_2)

# observation = env.reset()
# done = False

# state = torch.tensor(env.state).float().unsqueeze(0)

# duration_record = np.zeros(NUM_EPISODES)
# reward_record = np.zeros(NUM_EPISODES)

# #training
# for i_episode in range(NUM_EPISODES):
#     duration = 0 #reset duration
#     acc_r = 0
#     if i_episode % target_update_frequency == 0:
#         print("episode ", i_episode, "/", NUM_EPISODES)

#     # Initialize the environment and state
#     env.reset()
#     state = torch.tensor(env.state).float().unsqueeze(0).to(device)
#     state = state.repeat(1, INTEGRATED)

#     eps_record[i_episode] = EPS_END + (EPS_START - EPS_END) * \
#         math.exp(-1. * steps_done / EPS_DECAY)

#     for t in count():
#         recorder.capture_frame()
#         # Select and perform an action
#         action = select_action(state) #, epsilon)

#         states, reward, done, _ = env.step(action.item())
#         frames_1d = np.array(states).flatten()

#         reward = torch.tensor([reward], device=device)
#         duration += 1
#         r = 1
#         acc_r = acc_r + r # to gain total reward

#         # Observe new state
#         if not done:
#             next_state = torch.tensor(frames_1d).float().unsqueeze(0).to(device)
#         else:
#             next_state = None

#         # Store the transition in memory
#         memory.push(state, action, next_state, reward)

#         # Move to the next state
#         state = next_state

#         # Perform one step of the optimization (on the policy network)
#         optimize_model()
#         # update target net
#         if i_episode % target_update_frequency == 0:
#             target_net.load_state_dict(policy_net.state_dict()) #transfer the data in policy net to target net
#         if done:
#             duration_record[i_episode] = duration
#             reward_record[i_episode] = acc_r
#             break
```

```
# central_record[6,:] = duration_record
# print('Complete')
# recorder.close()
# env.close()

# duration analysis
# print("duration recorder shows: ", duration_record)

################################################
avg_duration = np.zeros(NUM_EPISODES)
acc_duration = 0

mean_reward = np.zeros([len(range(1,6))])
for s in range(1,6):
    for i in range(NUM_EPISODES):
        acc_duration += central_record[s-1,i]
        # if i == 0:
        #     avg_duration[i] = 0
        # else:
        #     avg_duration[i] = acc_duration / i
    mean_r = acc_duration/ NUM_EPISODES
    mean_reward[s-1] = mean_r

# avg_reward = np.zeros(NUM_EPISODES)

std_reward = np.zeros([len(range(1,6))])
# mix_p = np.zeros(NUM_EPISODES)
acc_reward = 0
for i in range(1,6):
    # acc_reward += reward_record[i]
    std_reward[i-1] = np.std(central_record[i-1,:])
    # if i == 0:
    #     avg_reward[i] = 0
    # else:
    #     avg_reward[i] = acc_reward / i
    #     mix_p[i] = avg_reward[i] + std_reward[i]
    #     mix_n[i] = avg_reward[i] - std_reward[i]

plt.figure(figsize=(20,10))

# plt.subplot(2,1,1)
plt.xlabel('replay buffer size')
plt.ylabel('Standard deviation')
plt.plot([300, 1300, 2300, 3300, 10000], std_reward,color='green', linestyle='-', label="standard deviation")
plt.plot([300, 1300, 2300, 3300, 10000], mean_reward,color='blue', linestyle='-', label="mean_reward")
# plt.plot(central_record[2,:],color='red', linestyle='-', label="k = 3")
# plt.plot(central_record[3,:],color='blue', linestyle=':', label="k = 4")
# plt.plot(central_record[4,:],color='green', linestyle='--', label="k = 5")
# plt.plot(central_record[5,:],color='red', linestyle=':', label="k = 6")
# plt.plot(central_record[6,:],color='darkgray', linestyle='-', label="k = 20")
# plt.plot(central_record[7,:],color='yellow', linestyle='--', label="e = 0.24")
# plt.plot(central_record[8,:],color='green', linestyle=':', label="e = 0.27")
# plt.plot(central_record[9,:],color='blue', linestyle=':', label="e = 0.30")

# plt.plot(avg_duration,color='red', linestyle=':', label="test2")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=3, mode="expand", borderaxespad=0.)
# # plt.subplot(2,1,2)
# plt.xlabel('Episode')
# plt.ylabel('Epsilon')
# plt.plot(eps_record,color='darkgray', linestyle='-', label="dynamic e")
# plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
#            ncol=3, mode="expand", borderaxespad=0.)
```

Appendix 3. Code of DDQN

```
#!/usr/bin/env python
# coding: utf-8

# In[17]:


# This is the coursework 2 for the Reinforcement Leaning course 2021 taught at Imperial College London (https://www.imperial.ac.uk/computing/current-students/courses/70028/)
# The code is based on the OpenAI Gym original (https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html) and modified by Filippo Valdettaro and Prof. Aldo Faisal for the purposes of the course.
# There may be differences to the reference implementation in OpenAI gym and other solutions floating on the internet, but this is the defeinitive implementation for the course.

# Instaling in Google Colab the libraries used for the coursework
# You do NOT need to understand it to work on this coursework

# get_ipython().system('pip install gym')

from IPython.display import clear_output
clear_output()

# In[18]:
```

```
# Importing the libraries

import gym
from gym.wrappers.monitoring.video_recorder import VideoRecorder    #records videos of episodes
import numpy as np
import matplotlib.pyplot as plt # Graphical library

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as T
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # get RTX3080 Ready

from collections import namedtuple, deque
from itertools import count
import math
import random
from gym.wrappers.frame_stack import FrameStack

clear_output()

# In[19]:

# Test cell: check ai gym  environment + recording working as intended

env = gym.make("CartPole-v1")
file_path = 'F:/COMP97144 Reinforcement learning/cw2/videos/video.mp4'
recorder = VideoRecorder(env, file_path)

observation = env.reset()
terminal = False
while not terminal:
  recorder.capture_frame()
  action = int(observation[2]>0)
  observation, reward, terminal, info = env.step(action)
  # Observation is position, velocity, angle, angular velocity

recorder.close()
env.close()

# In[20]:

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

# defining replay buffer
class ReplayBuffer(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

# In[21]:

class DQN(nn.Module):

    def __init__(self, inputs, outputs, num_hidden, hidden_size):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(inputs, hidden_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size) for _ in range(num_hidden-1)])
        self.output_layer = nn.Linear(hidden_size, outputs)

    def forward(self, x):
```

```
        x.to(device)

        x = F.relu(self.input_layer(x))
        for layer in self.hidden_layers:
            x = F.relu(layer(x))

        return self.output_layer(x)

# In[22]:


def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    # setting the imput of mini-batches
    transitions = memory.sample(BATCH_SIZE)

    batch = Transition(*zip(*transitions)) # seperate state action reward

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool) # transform the array of states into tensor so that GPU
can handle



    # Can safely omit the condition below to check that not all states in the
    # sampled batch are terminal whenever the batch size is reasonable and
    # there is virtually no chance that all states in the sampled batch are
    # terminal
    if sum(non_final_mask) > 0:
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    else:
        non_final_next_states = torch.empty(0,state_dim).to(device)

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch) # THIS GIVES CURRENT VALUE!
    #action batch is consisted of 1 and 0. so the above through gather will give actions values based on experience

    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device) # size:(1*batch_size)

    with torch.no_grad(): #there will be a backward() called later
        # Once again can omit the conditional if batch size is large enough
        if sum(non_final_mask) > 0:
            temp_next_action_d = D_policy_net(non_final_next_states) #gives the value of next two actions
            next_action = torch.max(temp_next_action_d,1)[1].unsqueeze(1) # here gives the next action that gives maximum value, a =
max_a(Q(s_(t+1),a)) and CHANGE DIMENSION
            temp_next_action = target_net(non_final_next_states)
            next_state_values[non_final_mask] = temp_next_action.gather(1,next_action).squeeze(1) # through the newest policy net to update the action
to take so as new a-s value
                                                                                                    #again mind Dimension
        else:
            next_state_values = torch.zeros_like(next_state_values)

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute loss
    loss = ((state_action_values - expected_state_action_values.unsqueeze(1))**2).sum()

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
```

```
    # Limit magnitude of gradient for update step
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()


# In[23]:

# Training
#setting parameters
NUM_EPISODES = 200
BATCH_SIZE = 256
GAMMA = 0.99
MEMORY_CAPACITY = 5000
INTEGRATED = 4 # number of previous states
# threshold_duration = 1000 this is not needed, the duration never exceed 500

epsilon = 0.09 # for epsilon greedy policy
num_hidden_layers = 5
size_hidden_layers = 200
target_update_frequency = 10
D_policy_net_frequency = 20

# Get number of states and actions from gym action space
env = gym.make("CartPole-v1")
env = FrameStack(env, INTEGRATED)
env.reset()
state_dim = len(env.state) * INTEGRATED   #states are: x, x_dot, theta, theta_dot

n_actions = env.action_space.n
env.close()

policy_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
D_policy_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
target_net = DQN(state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())

memory = ReplayBuffer(MEMORY_CAPACITY)

def select_action(state, current_eps=0):

    sample = random.random()
    eps_threshold = current_eps
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.

            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)

# In[24]:


file_path_2 = 'F:/COMP97144 Reinforcement learning/cw2/videos/video_1.mp4'
recorder = VideoRecorder(env, file_path_2)

observation = env.reset()
done = False

state = torch.tensor(env.state).float().unsqueeze(0)

duration_record = np.zeros(NUM_EPISODES)
reward_record = np.zeros(NUM_EPISODES)

#training
for i_episode in range(NUM_EPISODES):
    duration = 0 #reset duration
```

```
    acc_r = 0
    if i_episode % target_update_frequency == 0:
        print("episode ", i_episode, "/", NUM_EPISODES)

    # Initialize the environment and state
    env.reset()
    state = torch.tensor(env.state).float().unsqueeze(0).to(device)
    state = state.repeat(1, INTEGRATED)

    for t in count():
        recorder.capture_frame()
        # Select and perform an action
        action = select_action(state, epsilon)

        states, reward, done, _ = env.step(action.item())
        frames_1d = np.array(states).flatten()

        reward = torch.tensor([reward], device=device)
        duration += 1
        r = 1
        acc_r = acc_r + r # to gain total reward

        # Observe new state
        if not done:
            next_state = torch.tensor(frames_1d).float().unsqueeze(0).to(device)
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        # update target net
        if i_episode % target_update_frequency == 0:
            target_net.load_state_dict(policy_net.state_dict()) #transfer the data in policy net to target net
        if i_episode % D_policy_net_frequency == 0:
            D_policy_net.load_state_dict(policy_net.state_dict()) #transfer the data in policy net to target net
        if done:
            duration_record[i_episode] = duration
            reward_record[i_episode] = acc_r
            break
    recorder.close()
    env.close()
print("Episode duration: ", duration)



print('Complete')

# duration analysis
# print("duration recorder shows: ", duration_record)
avg_duration = np.zeros(NUM_EPISODES)
acc_duration = 0
for i in range(NUM_EPISODES):
    acc_duration += duration_record[i]
    if i == 0:
        avg_duration[i] = 0
    else:
        avg_duration[i] = acc_duration / i
avg_reward = np.zeros(NUM_EPISODES)
std_reward = np.zeros(NUM_EPISODES)
mix_n = np.zeros(NUM_EPISODES)
mix_p = np.zeros(NUM_EPISODES)
acc_reward = 0
for i in range(NUM_EPISODES):
    acc_reward += reward_record[i]
    std_reward[i] = np.std(reward_record[0:i])
    if i == 0:
        avg_reward[i] = 0
    else:
        avg_reward[i] = acc_reward / i
```

```
        mix_p[i] = avg_reward[i] + std_reward[i]
        mix_n[i] = avg_reward[i] - std_reward[i]

# print("duration recorder shows: ", avg_duration)
plt.figure(1, figsize=(20,10))
plt.title('Durations and averange durations')
plt.xlabel('Episode')
plt.ylabel('Duration')
plt.plot(duration_record,color='green', linestyle='-')
plt.plot(avg_duration,color='red', linestyle=':')

plt.figure(2, figsize=(20,10))
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.plot(reward_record,color='green', linestyle='-', label="reward")
plt.plot(avg_reward,color='red', linestyle='--', label="mean reward")
plt.plot(mix_p,color='blue', linestyle=':',label="standard deviation")
plt.plot(mix_n,color='blue', linestyle=':',label="standard deviation")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=3, mode="expand", borderaxespad=0.)

recorder.close()
env.close()
print("Episode duration: ", duration)
```