Reinforcement Learning
Coursework 2: CartPole Environment

Charlize Yang
CID: 01666113
Department of Computing
MSc Artificial Intelligence

November 29, 2021

1 Implementing a functional DQN

1.1 Implementing three features of DQN

The experience replay buffer was implemented in the `ReplayBuffer` class from line 83 in Code Appendix. It stores the transitions that the agent observes and allows the reuse of data later in training. By sampling from it randomly, the transitions that build up a batch are decorrelated. This can significantly stabilize and improve the DQN training. I set the maximum capacity of the replay buffer (i.e. the maximum number of transitions to be stored) to be 10000 and the batch size to be 128.

The target network was implemented as follows: I started creating the target network as a copy of the policy network in the `set_up_parameters()` function (line 205). Then I used the target network to compute the expected Q value in the `optimize_model()` function (line 160) for added stability. Both functions are called in the `learning()` function (lines 251 and 287) for training the DQN model. The target network has its weights kept frozen most of the time, but is updated with the policy network's weights every 20 steps in the `learning()` function (line 292). This was specified by setting the `target_update` variable to be 20 (line 184).

For stacking k-frames as input, I modified the class `DQN` to set the input size for the neural network to be $4k$ (4 is the dimension of a state). Then in the `set_up_parameters()` function, I called the `FrameStack` class from `gym.wrappers` to automatically stacks k frames together (line 197) and fed the flattened stacked tensor with dimension $4k \times 1$ to the policy net and target net (lines 204-205). In the `learning()` function, I called the `FrameStack` class again and initialised all the states as flattened tensor with dimension $4k \times 1$ (lines 261 and 275).

1.2 Design decisions of the deep network

The deep network architecture is shown in Figure 1. It consists of fully connected linear layers with ReLU activation functions after each linear layer for non-linearity. The number of neurons in the input layer equals the dimension of flattened stacked k-frames, which is $4k$. I chose $k = 4$ to focus on relatively recent frames, which makes it 16. The output layer contains two neurons, representing the estimated Q-values of selecting action `left` or `right`. I chose two hidden layers as the task is not overly complex and does not require an excessively deep neural network. Similarly, at 50 neurons per layer, the architecture is not excessively wide.

I used the Root Mean Squared Propagation (RMSProp) to minimise the loss function with a learning rate of 0.01. This algorithm forgets early gradients and focuses on the most recently observed partial gradients seen during the search, overcoming the limitation of, e.g. AdaGrad. The Huber loss was used to optimise the network, which is less sensitive to outliers in data than the squared error loss. The training was run over 200 epochs, sufficient to achieve good learning performance and relatively fast (6 min over ten repetitions).

For the hyperparameters in learning, I set the discount rate $\gamma = 0.99$. The exploration parameter in the ϵ-greedy strategy was set to be $\epsilon = 0.05$ to ensure that the agent has a high probability of following the greedy action. Table 1 shows all the default parameters in the network and hyperparameters used for learning.

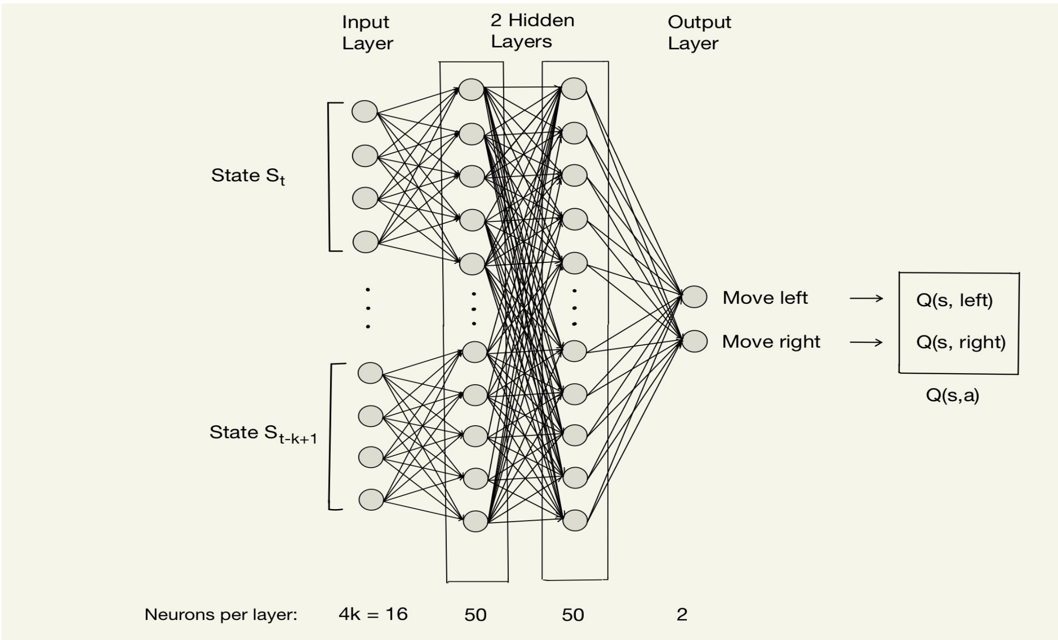


Figure 1: The architecture of the deep network.

Parameters of network		Hyperparameters for learning	
Number of hidden layers	2	Exploration parameter (ϵ)	0.05
Number of hidden units	50	Discount rate (γ)	0.99
Batch size	128	Size of replay buffer	10000
Number of episodes/epochs	200	Number of stacked frames (k)	4

Table 1: The parameters in the network and hyperparameters used in learning.

1.3 Learning curve of the DQN

The average total return and episode number at which my DQN achieves roughly 90% of the final performance value are approximately 160 and 185, respectively. This is indicated by the red point in Figure 2.

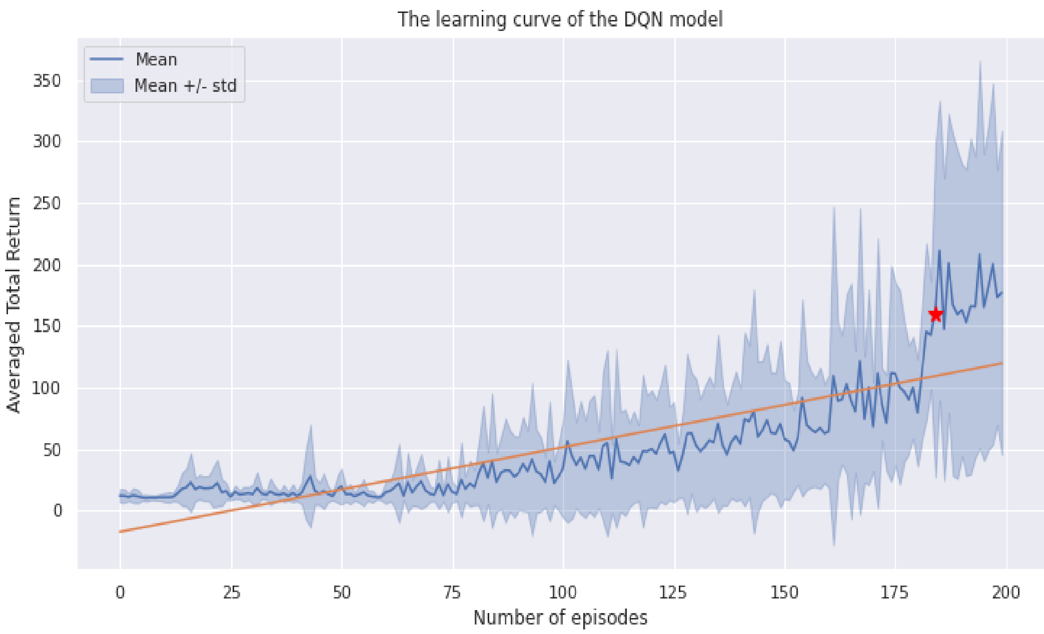


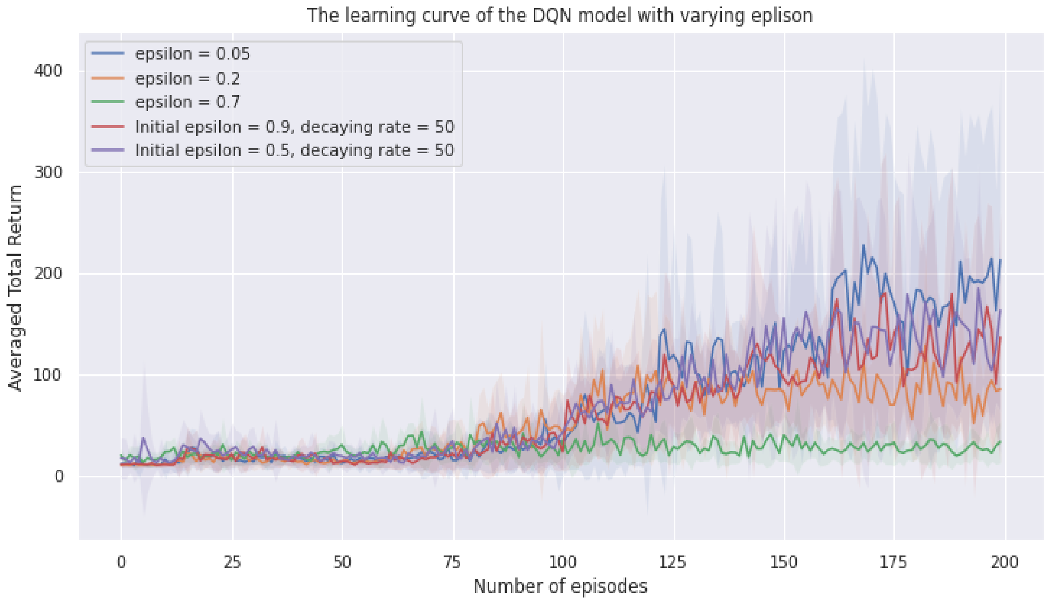
Figure 2: The learning curve of the DQN with (hyper)parameters in Table 1.

2 Hyperparameters of the DQN

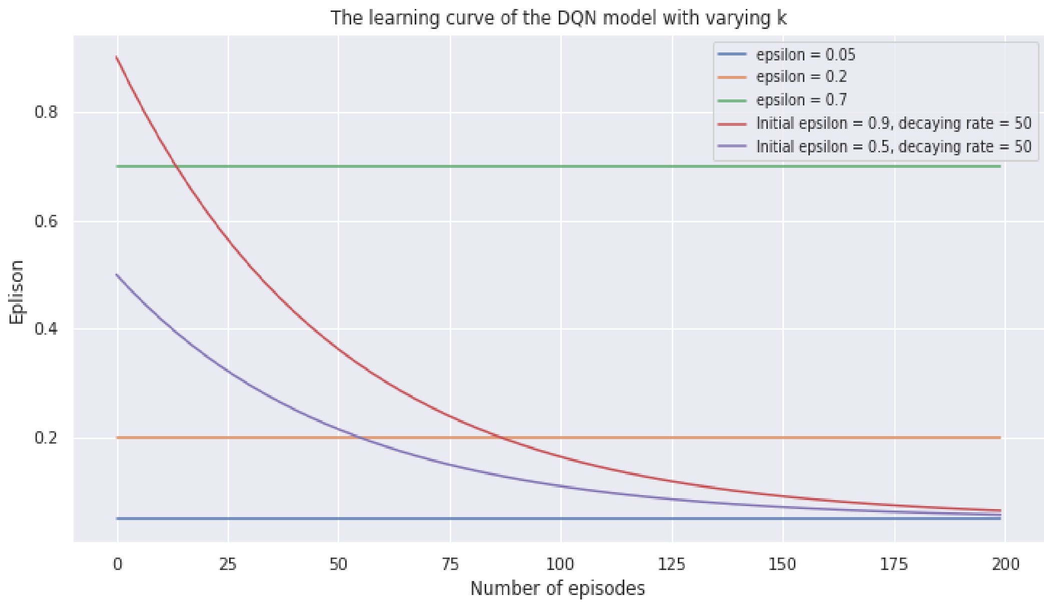
2.1 Constant and Variable Eplisons

In the ϵ-greedy policy, the lower the value of ϵ is, the higher the probability of selecting a greedy action instead of a random action randomly. In Figure 3a, I explored three constant ϵ with values (0.05, 0.2, 0.7) and two decaying ϵ, with an initial value of 0.9 and 0.5 respectively and an exponential decay rate of 50. The decaying ϵ helps us avoid sticking at a local optimum in the initial stage while focusing on exploitation as the agent learns more. Figure 3b shows all the epsilon values explored.

From Figure 3a, the constant $\epsilon = 0.05$ has an overall highest total return than the others, especially during episodes 160 to 200. However, the difference is not significant compared to the two decaying ϵ. The two decaying ϵs give similar learning performance, both outperforming the constant $\epsilon = 0.2$ and $\epsilon = 0.7$. The largest constant $\epsilon = 0.7$ gives the worst learning performance, indicating the importance of focusing on the greedy action. Therefore, a small constant epsilon suffices in this case, and a variable epsilon may not be necessary. Note that other options of variable epsilon may give different results.



(a)



(b)

Figure 3: (a) The learning curve of the DQN with varying epsilon values. (b) The constant and decaying epsilon values explored.

2.2 Size of replay buffer

I varied the default size of replay buffer 10000 several times by halving and doubling it. In Figure 4, we can see a general decreasing trend in the variability in return with the size of replay buffer (with some fluctuations between replay buffer sizes 2500 and 5000, 20000 and 40000). This is as expected, since the larger the experience replay, the less likely we will sample correlated transition samples, hence the more stable the network's training will be. It is worth noting that since here I only considered seven buffer sizes, this trend may be subject to more fluctuations if we consider more replay buffer sizes.

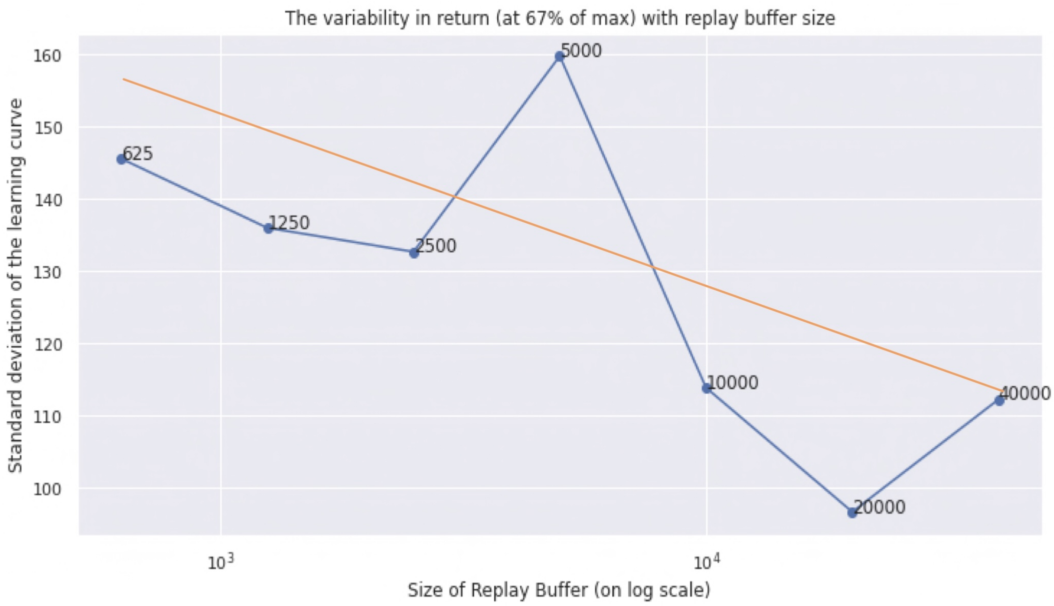


Figure 4: The variability in reward with varying replay buffer size (on the logarithmic scale). The standard deviation of the learning curve is taken at the 67% of the maximum total return.

2.3 Values of k

I set the range of k to be between 2 to 16. I chose 16 to learn sufficient information from the past frames while focusing on relatively recent ones.

As in Figure 5, $k = 16$ gives worst learning curve the lowest maximum return. While $k = 1$ shows exceptional learning performance in the initial episodes, it slows down after 175 episodes, compared to $k = 2, 4, 8$, but their differences are not significant. This suggests that stacking the frames as inputs may not be necessary for better performance. The learning performance is similar for $k = 2, 4, 8$, where $k = 2$ gives slightly higher returns than the other two values at some episode numbers. This suggests that by setting k within a relatively small range (e.g. [1, 4]), i.e. by collecting information from most recent frames, the DQN model gives better learning performance in the Cartpole environment.

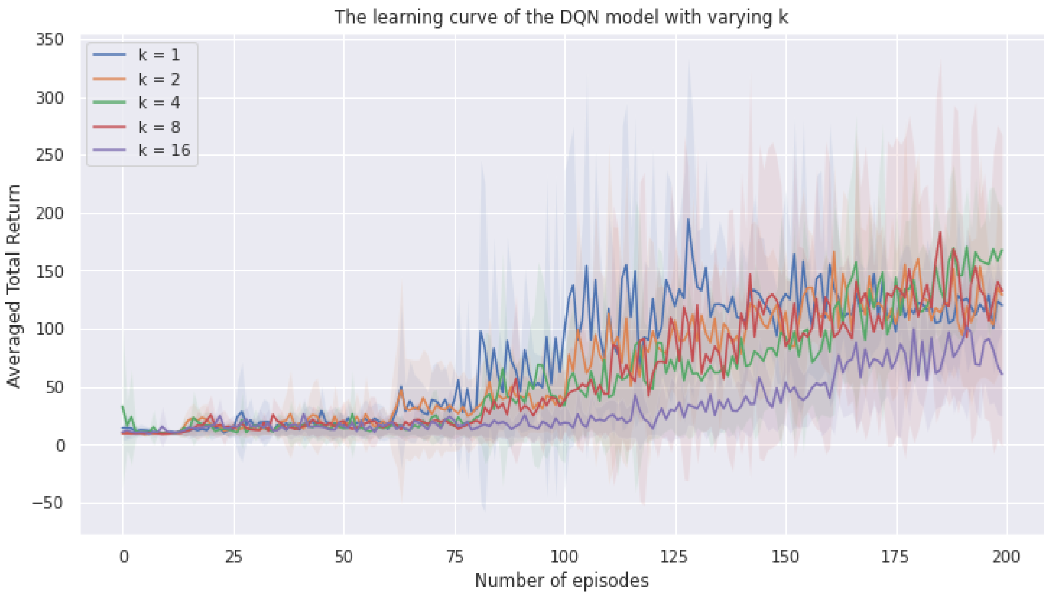


Figure 5: The learning curve of the DQN with varing values of k.

3 Ablation/Augmentation experiments

3.1 Implementation of Double DQN

I modified the `learning()` function by using the target network to select the action with highest Q value in `select_action()` (line 847), and named the modified function the `learning_ddqn()` function. This makes the selection of the action (using the target network) and the selection of Q-value (using the policy network) independent.

3.2 Comparison of learning curves

As shown in Figure 6, removing either the target network or replay buffer would result in a much lower final total return (both around 10) and barely any learning over the episodes. This suggests that, despite being time-consuming, the implementation of both the target network and replay buffer are necessary for the excellent performance of the DQN model. For the Double DQN model, the learning speed is much slower than a normal DQN, with a much lower maximum total return (around 50) and a much lower variance in return. This is as expected, as Double DQN helps reduce the frequency that the maximum Q-value is overestimated and smooths out the variance in estimates. As an improvement, we could plot the learning curve for more episodes and see if there is bigger/minor difference in the total returns of DQN and Double DQN.

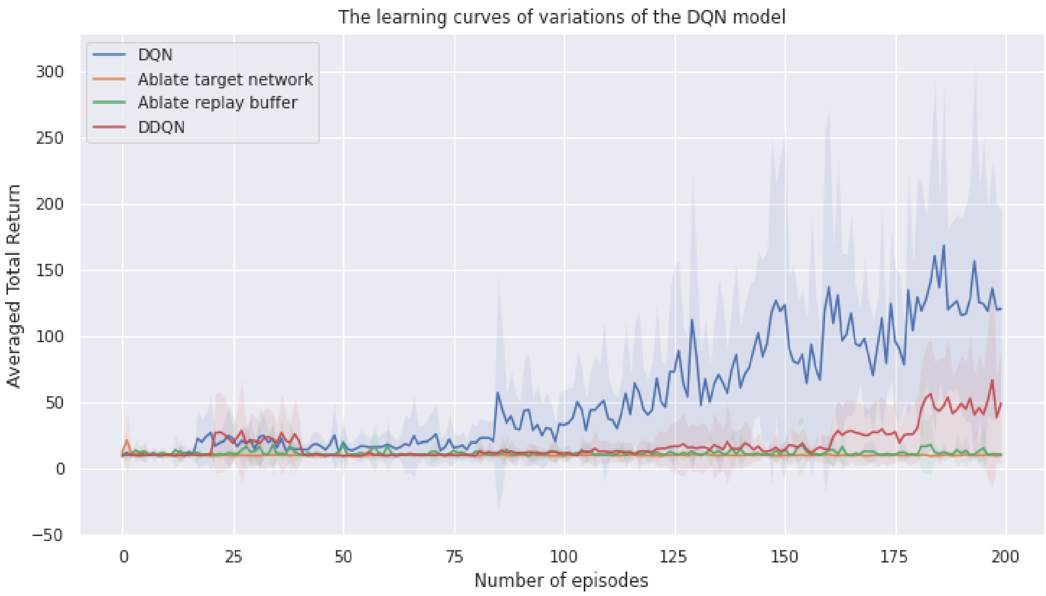


Figure 6: The learning curves of variations of the DQN model.

4 Code Appendix

```
# -*- coding: utf-8 -*-
"""RL CW2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Rs6DYOCHGwQjUEg5HmRL47z-khB5ob7a

# Initialisation
"""

# 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

# WARNING: if you don't use this Notebook in Google Colab, this block might print some warnings (do not mind them)

!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
!pip install colabgymrender==1.0.2
!wget http://www.atarimania.com/roms/Roms.rar
!mkdir /content/ROM/
!unrar e /content/Roms.rar /content/ROM/
!python -m atari_py.import_roms /content/ROM/

from IPython.display import clear_output
clear_output()

# 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 seaborn as sns
from sklearn.linear_model import LinearRegression

import torch
```

```
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ") # Configuring Pytorch

from collections import namedtuple, deque
from itertools import count
import math
import random

# WARNING: if you don't use this Notebook in Google Colab, comment
    out these two imports
from colabgymrender.recorder import Recorder # Allow to record
    videos in Google Colab
Recorder(gym.make("CartPole-v1"), './video') # Defining the video
    recorder
clear_output()

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

env = gym.make("CartPole-v1")
file_path = 'video/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()

"""# 1. Train the DQN model

"""

Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward')
    ) # 'state' and 'next_state' should be k-dimensional


class ReplayBuffer(object):

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

    def push(self, *args):
```

```python
        """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)

class DQN(nn.Module):

    def __init__(self, k, inputs, outputs, num_hidden, hidden_size):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(k*inputs, hidden_size) # The input size should be k*state_dim
        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)) # Apply the ReLU activation function in all hidden layers

        return self.output_layer(x)

def optimize_model(policy_net, target_net, optimizer, memory):

    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # 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)


```

```
    # 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, 4).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).to(device) # Added to device

    with torch.no_grad():
        # Once again can omit the condition if batch size is large enough
        if sum(non_final_mask) > 0:
            # Expected values of actions for non_final_next_states are computed based on the target_net
            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).to(device) # Added to device

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

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
```

```python
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # 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()

NUM_EPISODES = 200
BATCH_SIZE = 128
GAMMA = 0.999 # The discount rate
TARGET_UPDATE = 20

num_hidden_layers = 2
size_hidden_layers = 50
lr = 0.01
# epsilon = 0.05
# buffer_size = 10000
# k = 4

def set_up_parameters(buffer_size, k):

  # Get number of states and actions from gym action space
  env = gym.make("CartPole-v1")
  env = gym.wrappers.FrameStack(env, k) # Stack k frames together
  env.reset()

  state_dim = len(env.state)    # x, x_dot, theta, theta_dot
  n_actions = env.action_space.n    # left, right
  env.close()

  policy_net = DQN(k, state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device)
  target_net = DQN(k, state_dim, n_actions, num_hidden_layers, size_hidden_layers).to(device) # Initiate the target network
  # print(policy_net.state)
  target_net.load_state_dict(policy_net.state_dict())
  target_net.eval()

  optimizer = optim.RMSprop(policy_net.parameters(), lr = lr) # Use the Root Mean Squared Propagation
  memory = ReplayBuffer(buffer_size) # Set the size of the replay buffer

  return policy_net, target_net, optimizer, memory

def select_action(policy_net, state, current_eps):
    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(2)]], device=device, dtype=torch.long)

steps_done = 0

def select_action_decaying_eplison(policy_net, state, EPS_START, EPS_END, EPS_DECAY):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    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(2)]], device=device, dtype=torch.long)

def learning(epsilon, buffer_size = 10000, k = 4):
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k) # Stack k frames together
    env.reset()

    policy_net, target_net, optimizer, memory = set_up_parameters(buffer_size, k)

    list_of_returns = []

    for i_episode in range(NUM_EPISODES):
        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
```

```python
        env.reset()
        state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(device)
        # print(state.shape)

        total_return_for_the_episode = 0

        for t in count():
            # Select and perform an action
            action = select_action(policy_net, state, epsilon)
            _, reward, done, _ = env.step(action.item())
            total_return_for_the_episode += reward # Calculate the sum of undiscounted rewards
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(env.frames).float().flatten().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)
            # Calculate the Q-value this time using the policy_net
            optimize_model(policy_net, target_net, optimizer, memory)
            if done:
                break

         # Update the target network, copying all weights and biases in DQN
        if i_episode % TARGET_UPDATE == 0:
            target_net.load_state_dict(policy_net.state_dict())

        list_of_returns.append(total_return_for_the_episode)

    print('Complete')

    env.close()

    return(list_of_returns)

def learning_decaying_epsilon(EPS_START = 0.9, EPS_END = 0.05, EPS_DECAY = 50, buffer_size = 10000, k = 4):
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k) # Stack k frames together
```

```
    env.reset()

    policy_net, target_net, optimizer, memory = set_up_parameters(buffer_size, k)

    list_of_returns = []

    for i_episode in range(NUM_EPISODES):
        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(device)

        total_return_for_the_episode = 0

        for t in count():
            # Select and perform an action
            action = select_action_decaying_eplison(policy_net, state, EPS_START, EPS_END, EPS_DECAY)
            _, reward, done, _ = env.step(action.item())
            total_return_for_the_episode += reward # Calculate the sum of undiscounted rewards
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(env.frames).float().flatten().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(policy_net, target_net, optimizer, memory)
            if done:
                break

         # Update the target network, copying all weights and biases in DQN
        if i_episode % TARGET_UPDATE == 0:
            target_net.load_state_dict(policy_net.state_dict())

```

```
        list_of_returns.append(total_return_for_the_episode)

    print('Complete')

    env.close()

    return(list_of_returns)

## run an episode with trained agent and record video
## remember to change file_path name if you do not wish to
    overwrite an existing video

k = 4
env = gym.make("CartPole-v1")
env = gym.wrappers.FrameStack(env, k) # Stack k frames together

file_path = 'video/video.mp4'
recorder = VideoRecorder(env, file_path)

observation = env.reset()
done = False

state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(
    device)
state_dim = len(env.state)    #x, x_dot, theta, theta_dot
n_actions = env.action_space.n

policy_net = DQN(k, state_dim, n_actions, num_hidden_layers,
    size_hidden_layers).to(device)


duration = 0

while not done:
    recorder.capture_frame()

    # Select and perform an action
    action = select_action(policy_net, state, current_eps = 0.05)
    observation, reward, done, _ = env.step(action.item())
    duration += 1
    reward = torch.tensor([reward], device=device)
    # Observe new state
    state = torch.tensor(env.frames).float().flatten().unsqueeze(0)
    .to(device)

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

# Plot the learning curve for DQN
sns.set()

multiple_lists = []
```

```python

# Record the list of total returns of ten repetitions
for i in range(10):
    list_of_returns = learning(epsilon = 0.05)
    multiple_lists.append(list_of_returns)

arrays = [np.array(x) for x in multiple_lists]

# Compute the mean of returns over 10 repetitions
mean_return = np.array([np.mean(k) for k in zip(*arrays)])
# Compute the standard deviation of returns
std_return = np.array([np.std(k) for k in zip(*arrays)])

plt.figure(figsize=(12,6))
plt.plot(mean_return, label='Mean')
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return,
    std_return), np.add(mean_return, std_return), color='b', alpha
    =.3, label='Mean +/- std')

reg = LinearRegression().fit(np.arange(len(mean_return)).reshape
    (-1, 1), np.array(mean_return).reshape(-1, 1))
y_pred = reg.predict(np.arange(len(mean_return)).reshape(-1, 1))
plt.plot(y_pred)

final_mean = mean_return[-1]
print("90% of final performance of average total return", 0.9*
    final_mean)

for i, value in enumerate(mean_return):
    if value >= 0.9*final_mean:
        break

plt.plot(i, 0.9*final_mean, '*', color = 'red', markersize = 10)
print("Episode number at which the DQN achives 90% final
    performance:", i)

plt.ylabel('Averaged Total Return')
plt.xlabel('Number of episodes')
plt.title('The learning curve of the DQN model')
plt.legend(loc="upper left")

plt.show()

"""# 2. Hyperparameter Tuning

## 2.1 Epsilon
"""

# Plot the learning curve for DQN with constant and variable
    epsilons
sns.set()

multiple_lists_small = []
```

```
multiple_lists_medium = []
multiple_lists_large = []
multiple_lists_var1 = []
multiple_lists_var2 = []

# Record the list of total returns of ten repetitions
for i in range(10):
    multiple_lists_small.append(learning(epsilon = 0.05))
    multiple_lists_medium.append(learning(epsilon = 0.2))
    multiple_lists_large.append(learning(epsilon = 0.7))
    multiple_lists_var1.append(learning_decaying_epsilon(0.9, 0.05, 50, 10000, 4)) # EPS_START = 0.9
    multiple_lists_var2.append(learning_decaying_epsilon(0.5, 0.05, 50, 10000, 4)) # EPS_START = 0.5

arrays_small = [np.array(x) for x in multiple_lists_small]
arrays_medium = [np.array(x) for x in multiple_lists_medium]
arrays_large = [np.array(x) for x in multiple_lists_large]
arrays_var1 = [np.array(x) for x in multiple_lists_var1]
arrays_var2 = [np.array(x) for x in multiple_lists_var2]

# Compute the mean and sd of returns over 10 repetitions
mean_return_small = np.array([np.mean(k) for k in zip(*arrays_small)])
std_return_small = np.array([np.std(k) for k in zip(*arrays_small)])

mean_return_medium = np.array([np.mean(k) for k in zip(*arrays_medium)])
std_return_medium = np.array([np.std(k) for k in zip(*arrays_medium)])

mean_return_large = np.array([np.mean(k) for k in zip(*arrays_large)])
std_return_large = np.array([np.std(k) for k in zip(*arrays_large)])

mean_return_var1 = np.array([np.mean(k) for k in zip(*arrays_var1)])
std_return_var1 = np.array([np.std(k) for k in zip(*arrays_var1)])

mean_return_var2 = np.array([np.mean(k) for k in zip(*arrays_var2)])
std_return_var2 = np.array([np.std(k) for k in zip(*arrays_var2)])

plt.figure(figsize=(12,6))

plt.plot(mean_return_small, label='epsilon = 0.05')
plt.plot(mean_return_medium, label='epsilon = 0.2')
plt.plot(mean_return_large, label='epsilon = 0.7')
plt.plot(mean_return_var1, label='Initial epsilon = 0.9, decaying rate = 50')
```

```
plt.plot(mean_return_var2, label='Initial epsilon = 0.5, decaying rate = 50')

plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_small, std_return_small), np.add(mean_return_small, std_return_small), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_medium, std_return_medium), np.add(mean_return_medium, std_return_medium), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_large, std_return_large), np.add(mean_return_large, std_return_large), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_var1, std_return_var1), np.add(mean_return_var1, std_return_var1), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_var2, std_return_var2), np.add(mean_return_var2, std_return_var2), alpha=.1)

plt.ylabel('Averaged Total Return')
plt.xlabel('Number of episodes')
plt.title('The learning curve of the DQN model with varying eplison')
plt.legend(loc="upper left")
plt.show()

# Plot the graph for showing values of constant/variable eplisons
plt.figure(figsize=(12,6))

plt.plot([0.05]*200, label='epsilon = 0.05')
plt.plot([0.2]*200, label='epsilon = 0.2')
plt.plot([0.7]*200, label='epsilon = 0.7')
y_var1 = []
for i in range(200):
  new_y = 0.05 + (0.9 - 0.05) * math.exp(-1. * i / 50)
  y_var1.append(new_y)

y_var2 = []
for i in range(200):
  new_y = 0.05 + (0.5 - 0.05) * math.exp(-1. * i / 50)
  y_var2.append(new_y)

plt.plot(range(200), y_var1, label='Initial epsilon = 0.9, decaying rate = 50')
plt.plot(range(200), y_var2, label='Initial epsilon = 0.5, decaying rate = 50')

plt.ylabel('Eplison')
plt.xlabel('Number of episodes')
plt.title('The learning curve of the DQN model with varying k')

plt.legend(loc="upper right", prop={'size': 9.5})
plt.show()
```

```
"""## 2.2 Size of Replay Buffer"""

# Plot the learning curve for DQN with different buffer sizes
sns.set()

multiple_lists_625 = []
multiple_lists_1250 = []
multiple_lists_2500 = []
multiple_lists_5000 = []
multiple_lists_10000 = []
multiple_lists_20000 = []
multiple_lists_40000 = []

# Record the list of total returns of ten repetitions
for i in range(10):
    multiple_lists_625.append(learning(epsilon = 0.05, buffer_size = 625))
    multiple_lists_1250.append(learning(epsilon = 0.05, buffer_size = 1250))
    multiple_lists_2500.append(learning(epsilon = 0.05, buffer_size = 2500))
    multiple_lists_5000.append(learning(epsilon = 0.05, buffer_size = 5000))
    multiple_lists_10000.append(learning(epsilon = 0.05, buffer_size = 10000))
    multiple_lists_20000.append(learning(epsilon = 0.05, buffer_size = 20000))
    multiple_lists_40000.append(learning(epsilon = 0.05, buffer_size = 40000))

arrays_625 = [np.array(x) for x in multiple_lists_625]
arrays_1250 = [np.array(x) for x in multiple_lists_1250]
arrays_2500 = [np.array(x) for x in multiple_lists_2500]
arrays_5000 = [np.array(x) for x in multiple_lists_5000]
arrays_10000 = [np.array(x) for x in multiple_lists_10000]
arrays_20000 = [np.array(x) for x in multiple_lists_20000]
arrays_40000 = [np.array(x) for x in multiple_lists_40000]


# Compute the mean and sd of returns over 10 repetitions
mean_return_625 = np.array([np.mean(k) for k in zip(*arrays_625)])
std_return_625 = np.array([np.std(k) for k in zip(*arrays_625)])

mean_return_1250 = np.array([np.mean(k) for k in zip(*arrays_1250)])
std_return_1250 = np.array([np.std(k) for k in zip(*arrays_1250)])

mean_return_2500 = np.array([np.mean(k) for k in zip(*arrays_2500)])
std_return_2500 = np.array([np.std(k) for k in zip(*arrays_2500)])

```

```
mean_return_5000 = np.array([np.mean(k) for k in zip(*arrays_5000)])
std_return_5000 = np.array([np.std(k) for k in zip(*arrays_5000)])

mean_return_10000 = np.array([np.mean(k) for k in zip(*arrays_10000)])
std_return_10000 = np.array([np.std(k) for k in zip(*arrays_10000)])

mean_return_20000 = np.array([np.mean(k) for k in zip(*arrays_20000)])
std_return_20000 = np.array([np.std(k) for k in zip(*arrays_20000)])

mean_return_40000 = np.array([np.mean(k) for k in zip(*arrays_40000)])
std_return_40000 = np.array([np.std(k) for k in zip(*arrays_40000)])


plt.figure(figsize=(12,6))

plt.plot(mean_return_625, label='buffer_size = 625')
plt.plot(mean_return_1250, label='buffer_size = 1250')
plt.plot(mean_return_2500, label='buffer_size = 2500')
plt.plot(mean_return_5000, label='buffer_size = 5000')
plt.plot(mean_return_10000, label='buffer_size = 10000')
plt.plot(mean_return_20000, label='buffer_size = 20000')
plt.plot(mean_return_40000, label='buffer_size = 40000')

plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_625, std_return_625), np.add(mean_return_625, std_return_625), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_1250, std_return_1250), np.add(mean_return_1250, std_return_1250), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_2500, std_return_2500), np.add(mean_return_2500, std_return_2500), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_5000, std_return_5000), np.add(mean_return_5000, std_return_5000), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_10000, std_return_10000), np.add(mean_return_10000, std_return_10000), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_20000, std_return_20000), np.add(mean_return_20000, std_return_20000), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_40000, std_return_40000), np.add(mean_return_40000, std_return_40000), alpha=.1)

plt.ylabel('Averaged Total Return')
```

```python
plt.xlabel('Number of episodes')
plt.title('The learning curve of the DQN model with varying replay buffer size')
plt.legend(loc="upper left")
plt.show()

mean_ep = [mean_return_625, mean_return_1250, mean_return_2500, mean_return_5000, mean_return_10000, mean_return_20000, mean_return_40000]
std_ep = [std_return_625, std_return_1250, std_return_2500, std_return_5000, std_return_10000, std_return_20000, std_return_40000]
mean_and_std_ep = list(zip(mean_ep, std_ep))

std_list = []
for mean_and_std_return in mean_and_std_ep:
    mean_return, std_return = mean_and_std_return
    two_thirds_return = np.max(mean_return) * 2/3

    for j, reward in enumerate(mean_return): # j is the episode at which we measure the std
        if reward >= two_thirds_return:
            break

    std = std_return[j]
    std_list.append(std)

print(std_list)

# Plot the sd vs buffer size
plt.figure(figsize=(12,6))

buffer_size = [625, 1250, 2500, 5000, 10000, 20000, 40000]
plt.scatter(buffer_size, std_list)
plt.plot(buffer_size, std_list)
plt.xlabel("Size of Replay Buffer (on log scale)")
plt.ylabel("Standard deviation of the learning curve")
plt.xscale('log')
plt.title('The variability in return (at 67% of max) with replay buffer size')

for i, size in enumerate(buffer_size):
    plt.annotate(size, (buffer_size[i], std_list[i]))

plt.show()

"""## 2.3 Value of k"""

# Plot the learning curve for DQN with different k
sns.set()

multiple_lists_1 = []
multiple_lists_2 = []
```

```
multiple_lists_4 = []
multiple_lists_8 = []
multiple_lists_16 = []

# Record the list of total returns of ten repetitions
for i in range(10):
    multiple_lists_1.append(learning(epsilon = 0.05, k = 1))
    multiple_lists_2.append(learning(epsilon = 0.05, k = 2))
    multiple_lists_4.append(learning(epsilon = 0.05, k = 4))
    multiple_lists_8.append(learning(epsilon = 0.05, k = 8))
    multiple_lists_16.append(learning(epsilon = 0.05, k = 16))

arrays_1 = [np.array(x) for x in multiple_lists_1]
arrays_2 = [np.array(x) for x in multiple_lists_2]
arrays_4 = [np.array(x) for x in multiple_lists_4]
arrays_8 = [np.array(x) for x in multiple_lists_8]
arrays_16 = [np.array(x) for x in multiple_lists_16]

# Compute the mean and sd of returns over 10 repetitions
mean_return_1 = np.array([np.mean(k) for k in zip(*arrays_1)])
std_return_1 = np.array([np.std(k) for k in zip(*arrays_1)])

mean_return_2 = np.array([np.mean(k) for k in zip(*arrays_2)])
std_return_2 = np.array([np.std(k) for k in zip(*arrays_2)])

mean_return_4 = np.array([np.mean(k) for k in zip(*arrays_4)])
std_return_4 = np.array([np.std(k) for k in zip(*arrays_4)])

mean_return_8 = np.array([np.mean(k) for k in zip(*arrays_8)])
std_return_8 = np.array([np.std(k) for k in zip(*arrays_8)])

mean_return_16 = np.array([np.mean(k) for k in zip(*arrays_16)])
std_return_16 = np.array([np.std(k) for k in zip(*arrays_16)])

plt.figure(figsize=(12,6))

plt.plot(mean_return_1, label='k = 1')
plt.plot(mean_return_2, label='k = 2')
plt.plot(mean_return_4, label='k = 4')
plt.plot(mean_return_8, label='k = 8')
plt.plot(mean_return_16, label='k = 16')

plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_1,
    std_return_1), np.add(mean_return_1, std_return_1), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_2,
    std_return_2), np.add(mean_return_2, std_return_2), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_4,
    std_return_4), np.add(mean_return_4, std_return_4), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_8,
    std_return_8), np.add(mean_return_8, std_return_8), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_16,
    std_return_16), np.add(mean_return_16, std_return_16), alpha=.1)
```

```python
plt.ylabel('Averaged Total Return')
plt.xlabel('Number of episodes')
plt.title('The learning curve of the DQN model with varying k')
plt.legend(loc="upper left")
plt.show()

"""# 3. Ablation/Augmentation experiments

## 3.1 Ablate the target network
"""

def learning_ablate_target(epsilon, buffer_size = 10000, k = 4):
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k) # Stack k frames together
    env.reset()

    policy_net, _, optimizer, memory = set_up_parameters(
buffer_size, k)

    list_of_returns = []

    for i_episode in range(NUM_EPISODES):
        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.frames).float().flatten().
unsqueeze(0).to(device)
        # print(state.shape)

        total_return_for_the_episode = 0

        for t in count():
            # Select and perform an action
            action = select_action(policy_net, state, epsilon)
            _, reward, done, _ = env.step(action.item())
            total_return_for_the_episode += reward # Calculate the
sum of undiscounted rewards
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(env.frames).float().
flatten().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(policy_net, policy_net, optimizer, memory) # Calculate the Q-value using the policy_net
            if done:
                break

         # Update the target network, copying all weights and biases in DQN
         # if i_episode % TARGET_UPDATE == 0:
            # target_net.load_state_dict(policy_net.state_dict())

        list_of_returns.append(total_return_for_the_episode)

    print('Complete')

    env.close()

    return(list_of_returns)

"""## 3.2 Ablate the replay buffer

"""

def learning_ablate_buffer(epsilon, buffer_size = 1, k = 4): # Set the buffer size to be 1
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k) # Stack k frames together
    env.reset()

    policy_net, target_net, optimizer, memory = set_up_parameters(buffer_size, k)

    list_of_returns = []

    for i_episode in range(NUM_EPISODES):
        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(device)
        # print(state.shape)

        total_return_for_the_episode = 0

        for t in count():
            # Select and perform an action
            action = select_action(policy_net, state, epsilon)
            _, reward, done, _ = env.step(action.item())
```

```
            total_return_for_the_episode += reward # Calculate the sum of undiscounted rewards
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(env.frames).float().flatten().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(policy_net, target_net, optimizer, memory)
            if done:
                break

         # Update the target network, copying all weights and biases in DQN
        if i_episode % TARGET_UPDATE == 0:
            target_net.load_state_dict(policy_net.state_dict())

        list_of_returns.append(total_return_for_the_episode)

    print('Complete')

    env.close()

    return(list_of_returns)

"""## 3.3 Double DQN"""

def learning_ddqn(epsilon, buffer_size = 10000, k = 4):
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k) # Stack k frames together
    env.reset()

    policy_net, target_net, optimizer, memory = set_up_parameters(buffer_size, k)

    list_of_returns = []

    for i_episode in range(NUM_EPISODES):
        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

```

```
        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(device)
        # print(state.shape)

        total_return_for_the_episode = 0

        for t in count():
            # Select and perform an action
            action = select_action(target_net, state, epsilon) # Use the target net to select action
            _, reward, done, _ = env.step(action.item())
            total_return_for_the_episode += reward
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(env.frames).float().flatten().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(policy_net, target_net, optimizer, memory)
            if done:
                break

         # Update the target network, copying all weights and biases in DQN
        if i_episode % TARGET_UPDATE == 0:
            target_net.load_state_dict(policy_net.state_dict())

        list_of_returns.append(total_return_for_the_episode)

    print('Complete')

    env.close()

    return(list_of_returns)

"""## 3.4 Plot the learning curves"""

# Plot the learning curves for DQN without target network or replay buffer
```

```
sns.set()

multiple_lists_no_target = []
multiple_lists_no_buffer = []

# Record the list of total returns of ten repetitions
for i in range(10):
    multiple_lists_no_target.append(learning_ablate_target(epsilon = 0.05))
    multiple_lists_no_buffer.append(learning_ablate_buffer(epsilon = 0.05))

arrays_no_target = [np.array(x) for x in multiple_lists_no_target]
arrays_no_buffer = [np.array(x) for x in multiple_lists_no_buffer]

# Compute the mean and sd of returns over 10 repetitions
mean_return_no_target = np.array([np.mean(k) for k in zip(*arrays_no_target)])
std_return_no_target = np.array([np.std(k) for k in zip(*arrays_no_target)])

mean_return_no_buffer = np.array([np.mean(k) for k in zip(*arrays_no_buffer)])
std_return_no_buffer = np.array([np.std(k) for k in zip(*arrays_no_buffer)])

plt.figure(figsize=(12,6))

plt.plot(mean_return_no_target, label='Ablate target network')
plt.plot(mean_return_no_buffer, label='Ablate replay buffer')

plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_no_target, std_return_no_target), np.add(mean_return_no_target, std_return_no_target), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_no_buffer, std_return_no_buffer), np.add(mean_return_no_buffer, std_return_no_buffer), alpha=.1)

plt.ylabel('Averaged Total Return')
plt.xlabel('Number of episodes')
plt.title('The learning curves of variations of the DQN model')
plt.legend(loc="upper left")
plt.show()

# Plot the learning curves for variations of DQN
sns.set()

multiple_lists_dqn = []
multiple_lists_no_target = []
multiple_lists_no_buffer = []
multiple_lists_ddqn = []

# Record the list of total returns of ten repetitions
```

```python
for i in range(10):
    multiple_lists_dqn.append(learning(epsilon = 0.05))
    multiple_lists_no_target.append(learning_ablate_target(epsilon
   = 0.05))
    multiple_lists_no_buffer.append(learning_ablate_buffer(epsilon
   = 0.05))
    multiple_lists_ddqn.append(learning_ddqn(epsilon = 0.05))

arrays_dqn = [np.array(x) for x in multiple_lists_dqn]
arrays_no_target = [np.array(x) for x in multiple_lists_no_target]
arrays_no_buffer = [np.array(x) for x in multiple_lists_no_buffer]
arrays_ddqn = [np.array(x) for x in multiple_lists_ddqn]

# Compute the mean and sd of returns over 10 repetitions
mean_return_dqn = np.array([np.mean(k) for k in zip(*arrays_dqn)])
std_return_dqn = np.array([np.std(k) for k in zip(*arrays_dqn)])

mean_return_no_target = np.array([np.mean(k) for k in zip(*
    arrays_no_target)])
std_return_no_target = np.array([np.std(k) for k in zip(*
    arrays_no_target)])

mean_return_no_buffer = np.array([np.mean(k) for k in zip(*
    arrays_no_buffer)])
std_return_no_buffer = np.array([np.std(k) for k in zip(*
    arrays_no_buffer)])

mean_return_ddqn = np.array([np.mean(k) for k in zip(*arrays_ddqn)
    ])
std_return_ddqn = np.array([np.std(k) for k in zip(*arrays_ddqn)])

plt.figure(figsize=(12,6))

plt.plot(mean_return_dqn, label='DQN')
plt.plot(mean_return_no_target, label='Ablate target network')
plt.plot(mean_return_no_buffer, label='Ablate replay buffer')
plt.plot(mean_return_ddqn, label='DDQN')

plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_dqn,
    std_return_dqn), np.add(mean_return_dqn, std_return_dqn), alpha
    =.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(
    mean_return_no_target, std_return_no_target), np.add(
    mean_return_no_target, std_return_no_target), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(
    mean_return_no_buffer, std_return_no_buffer), np.add(
    mean_return_no_buffer, std_return_no_buffer), alpha=.1)
plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_ddqn,
     std_return_ddqn), np.add(mean_return_ddqn, std_return_ddqn),
    alpha=.1)

plt.ylabel('Averaged Total Return')
plt.xlabel('Number of episodes')
```

```
plt.title('The learning curves of variations of the DQN model')
plt.legend(loc="upper left")
plt.show()
```