COURSEWORK 2

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

COMP97143: Reinforcement Learning

Author:
Jaime Sabal Bérmudez (CID: 01520988)

Date: November 29, 2021

Question 1: Implementing a Functional DQN

1.1-1.2: Implementation of DQN

Replay Buffer).

The `ReplayBuffer` class was implemented with a double-ended queue (i.e deque) as its main component (line 26 in source code) which allowed for an easy way to keep track a number of previously sampled states (given by the variable `buffer_size`). After the agent performs a step in the environment, we make use of the `push` method (line 28) to save this transition. This transition is a tuple of the form (s_t, a_t, s_{t+1}, r_t) , where s_t is the current state, a_t is the action taken, s_{t+1} is the next state and r_t is the reward obtained from transitioning from s_t to s_{t+1}. Once enough states have been stored in the memory (more than the batch size of the Deep Q Network (DQN)), the weights of the network are optimised by randomly sampling a batch of previously seen states. Doing this allows for less correlation between training samples and for these to follow a similar probability distribution (i.i.d) like is expected in supervised learning.

Target Network).

The target network is initialised in lines 118-119 of the source code by copying the states dictionary of the policy network. This network is used in calculating the Temporal Difference (TD) error, which stabilises the learning process by allowing the policy network to consider a variety of actions before updating its weights according to an error that may not be representative of the optimum actions to take for each state.

Frame Stacking and Skipping).

We also implement the feature of being able to add multiple frames k as input to the DQN such that it contains the states $(s_t, s_{t-1}, ..., s_{t-k})$ and thus has shape $(1, 4k)$. We create the class `FrameStacking` (line 57), which inherits from `gym.Wrapper`, with a `reset_buffer` method that empties the memory deque and a `_k_states` property that converts the deque to the expected form of the DQN input. We also implement frame skipping (lines 233-243), mostly to accelerate training times when performing replications of our experiments, since the dynamics of the system are set such that one step corresponds to 0.02 seconds and that is close to the average 0.025 second reaction time of a human.

1.3: Justification of Network Architecture

Figure 1 shows the chosen architecture of our DQN. Four frames ($k = 4$), each containing four observables (x, $\dot{x}$, θ, $\dot{\theta}$), were used as input to the network, 150 neurons are present in each hidden layer, and the ReLu activation function operates on all layers but the output layer. Moreover, the Adam optimiser with a learning rate of $\alpha = 0.0001$ was used to update the weights in training with minibatches of size 128, which we found yielded better results than using RMSProp and larger values for α after some experimentation.

In terms of hyperparameters specific to a DQN, we use a discount rate $\gamma = 0.9999$, update the target network every 40 episodes, use a replay buffer size of 100,000, and a rewards-based ϵ-greedy behaviour policy with a starting $\epsilon_i = 1.0$ and a final $\epsilon_f = 0.0005$. Specifically, in a rewards-based ϵ-decay ϵ is reduced by $\delta = \left(\epsilon_i - \epsilon_f\right)/\mathrm{r_T}$ every time the reward obtained in an episode is larger than a threshold value that is initialised to be 0 and increased by 1 every time this condition is met.

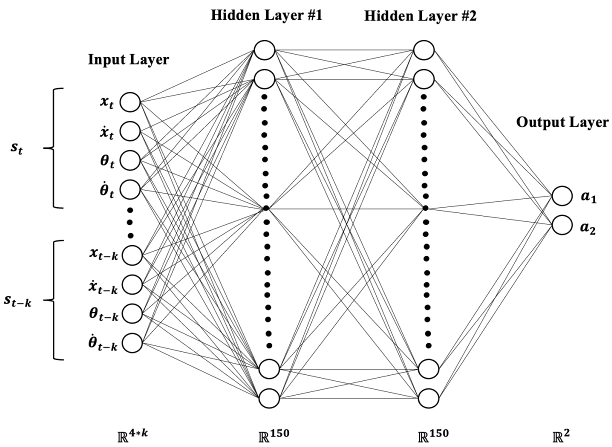

Figure 1: Graphical representation of the architecture used in the Deep Q Network. Each connection to the center of the hidden layers represents the neurons' connection to each of the remaining neurons (i.e 146) in the subsequent layer.

In the mentioned equation, r_T is a reward target value that we set to be 100. By doing this, the decay is much more controlled and only decreases if our agent obtains high rewards, which in itself is an indication that our agent should explore less. Moreover, the discount rate γ is set very high due to the fact that we are interested in obtaining high rewards in the very far future. Finally, updating the target network every 40 episodes (f_T) and using a replay buffer size of 100,000 allowed for a more stable network that consistently converged to a reward of ≈ 300 in 700 episodes without suffering from catastrophic forgetting.

1.4: Learning Curve of Agent

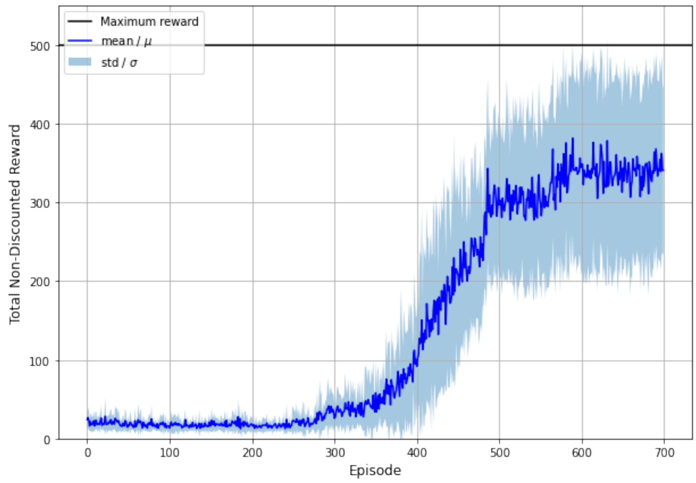

Figure 2: Average learning curve and its standard deviation of a DQN agent solving OpenAI's cartpole environment across 15 replications; `batch_size` $= 128$, $\alpha = 0.0001$, $f_T = 40$, $k = 4$, `skipped_frames` $= 4$, `buffer_size` $= 100{,}000$, $\epsilon_i = 1.0$, $\epsilon_f = 0.0005$, $r_T = 100$.

Figure 2 shows the average learning curve of the DQN agent across 15 replications for 700 training episodes. The range chosen for the plot is $[0, 550]$ such that it clearly shows the maximum attainable reward (500) for any given episode. This number of replications was chosen based on an informal analysis of the tardeoff between the marginal benefit of an extra replication and the time taken to compute it. Specifically, after 15 replications the smoothness of the average reward and standard deviation curves didn't seem to improve much. Moreover, the average total return of the final 100 episodes during training was found to be ≈ 339, resulting in our agent achieving 90% of this expected performance (305) at episode 485.

Question 2: Hyperparameters of the DQN

2.1: Effect of ϵ on the Learning Curve of the Agent

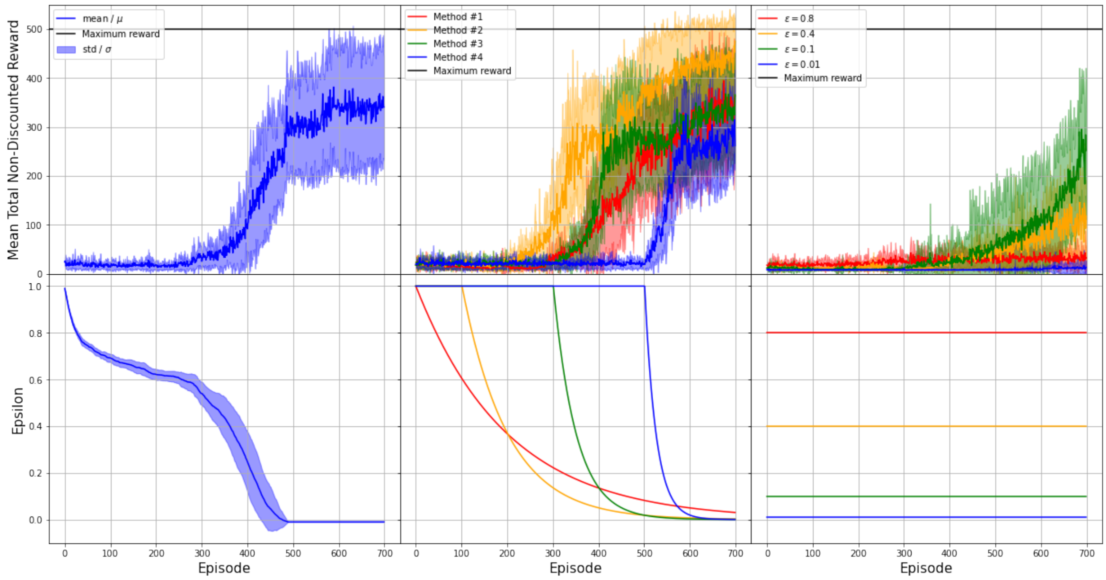

Figure 3: Learning curves of DQN agent for different ϵ schedules with the same starting and ending $\epsilon = 1.0$ and $\epsilon = 0.0005$, respectively; **left).** reward-based decay with a rewards target of 100 (also shows σ for the variation in the decay for different runs); **center).** Exponential ϵ decay with the decay beginning at different episodes in training for different methods; #1 = 0, #2 = 100, #3 = 300, #4 = 500; **right).** Constant ϵ throughout learning.

Figure 3 shows the learning curves for the DQN agent using different ϵ schedules while keeping the other hyper-parameters constant and equal to those detailed in Question 1. As can be seen, the best schedules seem to be the rewards-based and exponential decays (left and center columns). The exponential decay schedules were done by starting the attenuation at different episodes following the equation:

$$\epsilon_t = \epsilon_f + \left(\epsilon_i - \epsilon_f\right)\exp\left[\frac{-\mathrm{t}}{\tau}\right] \tag{1}$$

where t is the episode and τ is a decay factor. The different curves shown use $\tau = 200$ (#1), $\tau = 100$ (#2), $\tau = 50$ (#3), $\tau = 25$ (#4) and begin the attenuation in episodes 0, 100, 300, and 500, respectively. This plot shows how curve #2 is best for this schedule, allowing for some exploration at the start but giving the agent enough time to learn the environment very well in just 700 episodes. For a constant epsilon throughout training (right-most column), the agent is unable to learn anything for very large or small ϵ since choosing a random action way too often or virtually ever inhabilitates learning (agent must do some exploring but not too much). However, for the intermediate $\epsilon = 0.1$ the agent seems to start learning at a very late episode, which makes sense for a policy that takes a long time to sample all possible scenarios in the environment. Therefore, an ϵ that varies during training is preferred but not completely necessary, since using an appropriate constant ϵ still allows for some degree of learning.

2.2: Effect of Replay Buffer on Variability of Reward During Learning

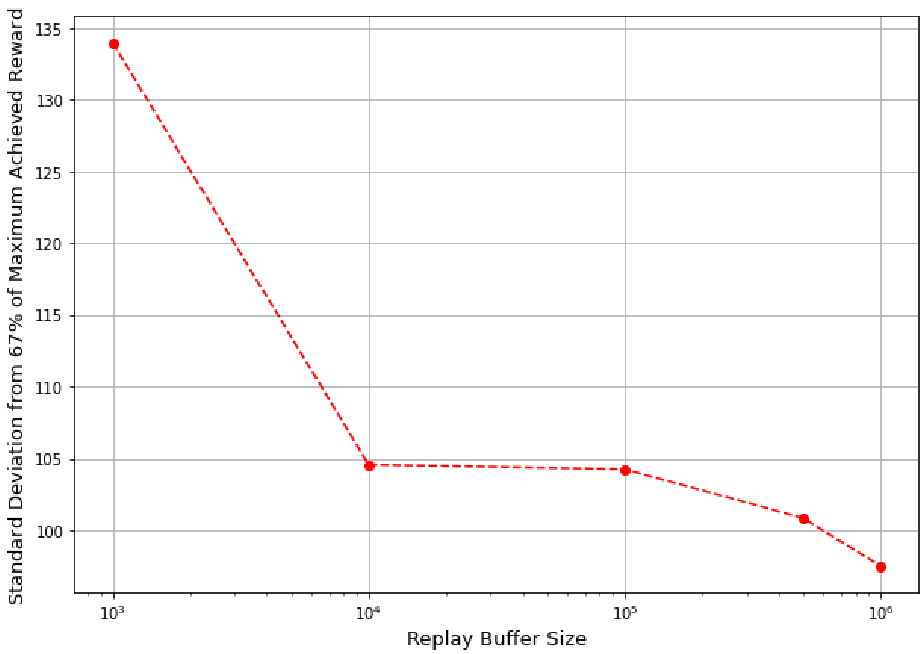

Figure 4: Replay buffer size (in a log scale) against variability in learning curve from 67% of the maximum achieved total reward. The remaining hyper-parameters are the same as those used to compute Figure 2

Exploring the effect of the replay buffer size on the total reward can give us insight on the importance of experience replay in a Deep Q Network. In Figure 4, it can be seen how there is a clear trend that as the buffer size is increased, the variability in the reward during learning from 67% of the maximum achieved reward decreases. One reason for this is that as the agent keeps track of more experiences from the past, the sampled data at each step of the episode is less correlated and the agent is thus able to converge better to a policy that yields consistently good rewards since the data is more independent and identically distributed (i.i.d) like that assumed in supervised learning tasks. In other words, by having a larger amount of previous episodes available to sample from, the agent is less likely to depend on any given run and thus stability is increased in learning.

2.3: Effect of k on the Overall Learning and Final Performance

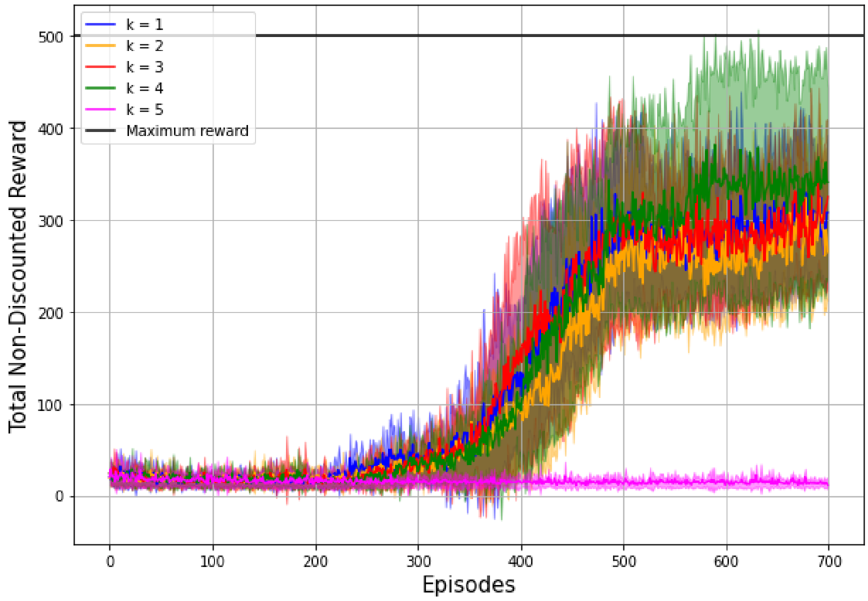

Figure 5: Learning curves for different numbers of frames $k \in [1, 5]$ used as input to the policy network in determining the optimum action to take at any given state. The remaining hyper-parameters used are the same as those shown on Figure 2.

The upper bound for k should be equal to the number of actions after which our agent would not be able to recover if it were to perform the same action repeatedly from the initial state with $\theta = 0$. After performing 100,000 runs of the environment with this policy, I found that the average number of steps needed to reach a terminal state when performing a constant action (either always 0 or always 1) is 9.35. This implies that after performing 5 steps in the same direction our agent will, on average, not be able to perform enough steps in the other direction to recover. Hence, using a $k > 5$ will impede the agents learning process because the network's input will always contain at least one frame that contains conflicting information on the optimum action to take at the current state. Figure 5 shows this phenomenon clearly, along with four other learning curves for $1 \leq k < 5$. For the latter, no major improvement in either the maximum achieved rewards or the variability of the learning curves can be seen. This is presumably due to the environment being deterministic and Markovian, implying that only the current state is necessary for determining the action that will result in the largest total non-discounted rewards. This means that the extra states being inputted to the network in the case where $k > 1$ are redundant and don't bear importance in the agent's learning.

Question 3: Ablation/Augmentation Experiments

3.1: DDQN Implementation

We implement the Double Deep Q Network agent by modifying the DQN agent slightly. In DQN, we calculate the target used in computing the MSE loss of our network (expected state-action value function) by:

$$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t'\right) \tag{2}$$

where $\boldsymbol{\theta}_t'$ are the weights of the target network and we are thus using the same network to both select and evaluate any given action. Since the value function is prone to be very noisy, taking its maximum leads to overestimation errors that dampen the learning process (maximisation bias). This occurs because the values for different actions are most likely not overestimated by equal amounts and thus for any given episode the weights of the behaviour policy $\boldsymbol{\theta}_t$ may be updated to favour actions that are in reality not optimal. The solution to this problem is to select the actions with the online behaviour policy and use these actions to evaluate the target network (Double Q Network). This can be written as:

$$Y_t^{DDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a}{\operatorname{argmax}} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right); \boldsymbol{\theta}_t'\right) \tag{3}$$

The idea behind this method to calculate the TD errors is that because the weights $\boldsymbol{\theta}_t'$ are updated on a periodic basis (every 40 episodes in our case), the likeliness of both policies overestimating the same action is greatly reduced since the noise present in both of them are independent of each other. The implementation of this augmentation to the idea of a target network can be seen between lines 181 and 185 of the source code.

3.2: Learning Curves for DQN Modifications Against Original Model

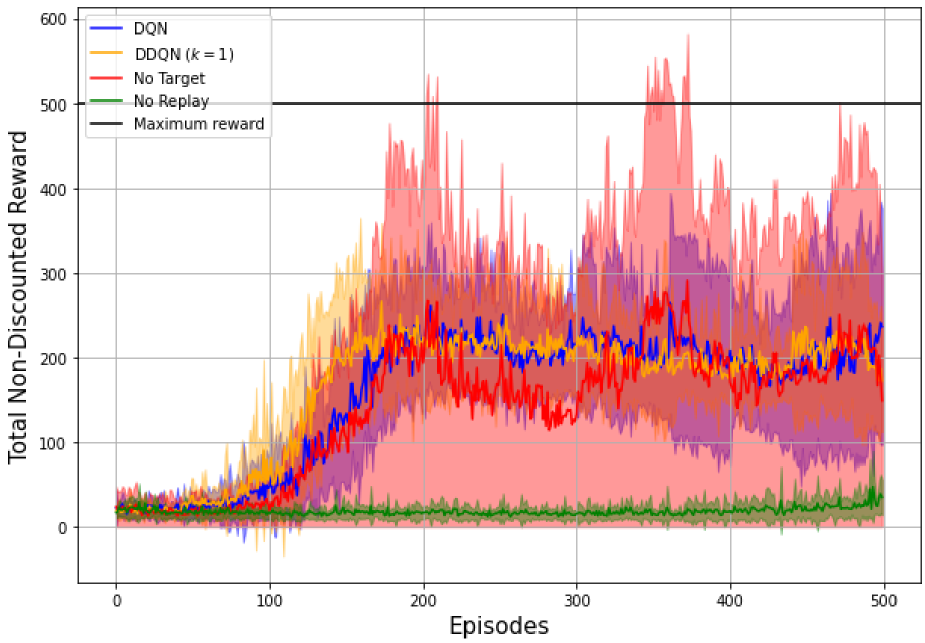

Figure 6: Learning curves for a DQN agent, one without a target network, one without experience replay, and a DDQN agent. `batch_size` = 128 (for No Replay `batch_size` = 1), α = 0.0001, f_T = 40 (except for No Target), k = 4 (except for DDQN where k = 1), `skipped_frames` = 1, `buffer_size` = 100,000, ϵ_i = 1.0, ϵ_f = 0.0005, r_T = 100.

As can be seen in Figure 6, the learning curves for the two ablations is quite different from that of the DQN and DDQN agents. It is important to note that for this section of the report, we eliminated frame skipping due to time constraints because it allowed for convergence in less episodes, albeit this also resulted in a lower final performance of the agents (see Figure 2).

Without having experience replay, the agent forgets previous transitions and is thus unable to learn from its mistakes. On the other hand, removing the target network has the effect of producing a very unstable learning process due to the fact that we are bootstrapping a continuous state space representation. In other words, updating the Q network for one state updates it for all the other states too which can lead to a resonance effect (i.e catastrophic forgetting). This is evidenced by the oscillating nature of the curve (red) and the absurdly high variance seen between replications.

Finally, the DDQN model shows a very similar learning curve to the DQN agent. However, a key difference between the two is the time taken to converge to a solution. The DDQN's learning curve can be seen to reach a similar final performance more quickly. This is due to the agent spending less time following un-optimal policies, since the Q-values of these state-action pairs are much less likely to be larger than the optimal alternative.

1 Appendix

1.1 Source Code

```python
import gym
import copy
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 tqdm

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

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

#dont change
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)

#dont change
class DQN(nn.Module):

```

```python
    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)


class FrameStacking(gym.Wrapper):
    """Return only every 4th frame"""
    def __init__(self, env, k):
        super(FrameStacking, self).__init__(env)
        self._obs_buffer = deque([], maxlen=k)
        self._skip       = k

    @property
    def _k_states(self):
        return torch.stack(list(self._obs_buffer)).reshape(-1).unsqueeze(0)

    def reset_buffer(self):
        self._obs_buffer = deque([], maxlen=self._skip)


class CartpoleAgent():
    def __init__(self, NUM_EPISODES, BATCH_SIZE, GAMMA, TARGET_UPDATE_FREQ,
                 NUM_FRAMES, SKIPPED_FRAMES, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS, REPLAY_BUFFER, LR,
                 EPS_START, EPS_END, REWARD_TARGET, EPS_DECAY=None, DDQN=False, eps_decay_strat="reward",
                 ablate_target=False, ablate_replay=False):

        #initialisation attributes
        self.reward_target = REWARD_TARGET
        self.reward_threshold = 0
```

```
        self.num_episodes = NUM_EPISODES
        self.batch_size = BATCH_SIZE
        self.gamma = GAMMA
        self.target_update_freq = TARGET_UPDATE_FREQ
        self.eps_start = EPS_START
        self.eps_end = EPS_END
        self.k = NUM_FRAMES
        self.skip_frames = SKIPPED_FRAMES
        self.curr_episode = 0
        self.losses = []
        self.train_rewards = []
        self.eps_decay_strat = eps_decay_strat

        self.ablate_target = ablate_target
        self.ablate_replay = ablate_replay

        if self.eps_decay_strat[:3] == "exp":
            if EPS_DECAY == None:
                raise Exception("Have to define EPS_DECAY if decay strategy is exponential")
            else:
                self.eps_decay = EPS_DECAY

        self.epsilon = self.eps_start
        self.epsilon_list = []
        self.epsilon_delta = (self.epsilon - self.eps_end)/self.reward_target

        #Get number of states and actions from gym action space
        env = gym.make("CartPole-v1")
        env.reset()
        self.state_dim = len(env.state)    #x, x_dot, theta, theta_dot
        self.n_actions = env.action_space.n
        env.close()

        self.input_dim = int(self.state_dim*self.k) #define input size of DQN

        #define policy and target networks, as well as optimizer and replay buffer
        self.policy_net = DQN(self.input_dim, self.n_actions, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS).to(device)
        self.target_net = DQN(self.input_dim, self.n_actions, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS).to(device)
        self.target_net.load_state_dict(self.policy_net.state_dict())
```

```
        self.target_net.eval()

        self.DDQN = DDQN
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=LR)
        self.memory = ReplayBuffer(REPLAY_BUFFER)




    def select_action(self, k_states):
        """"""
        sample = random.random()
        if sample > self.epsilon:
            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 self.policy_net(k_states).max(1)[1].view(1, 1)
        else:
            return torch.tensor([[random.randrange(self.n_actions)]], device=device, dtype=torch.long)


    def optimize_model(self):
        if len(self.memory) < self.batch_size:
            return
        transitions = self.memory.sample(self.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: torch.sum(s[0][-self.k:]).absolute().item() > 0, batch.next_state)), device=device, dtype=torch.bool)
```

```python

        # 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 torch.sum(s[0][-self.k:]).absolute().item() > 0])
        else:
            non_final_next_states = torch.empty(0,self.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 = self.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(self.batch_size, device=device)

        with torch.no_grad():
            # Once again can omit the condition if batch size is large enough
            if sum(non_final_mask) > 0:
                if self.DDQN:
                    #DDQN ---> update next states Q values (using target net) using the actions that maximise the policy network
                    actions_non_final = torch.zeros_like(action_batch.view(non_final_mask.shape))
```

```
                    actions_non_final = torch.argmax(self.policy_net(non_final_next_states), 1).unsqueeze(1)
                    next_state_values[non_final_mask] = self.target_net(non_final_next_states).gather(1, actions_non_final).flatten()
                else:
                    if self.ablate_target:
                        next_state_values[non_final_mask] = self.policy_net(non_final_next_states).max(1)[0].detach()
                    else:
                        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1)[0].detach()

            else:
                next_state_values = torch.zeros_like(next_state_values)


        expected_state_action_values = (next_state_values * self.gamma) + reward_batch

        # Compute loss
        loss_func = nn.MSELoss()
        loss = loss_func(state_action_values, expected_state_action_values.unsqueeze(1))


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

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

        self.optimizer.step()

    def train(self):
        """"""
        env = gym.make("CartPole-v1")
        steps = 0
        stack_frames = FrameStacking(env, self.k) #define frame stacking framework
```

```
        decay_iter = 0

        for i_episode in tqdm.tqdm(range(self.num_episodes)):
            rewards = 0
            #if i_episode % 20 == 0:
                #print("episode ", i_episode, "/", self.num_episodes)

            env.reset() #reset environment
            state = torch.tensor(env.state).float().unsqueeze(0).to(device)
            for _ in range(self.k): # Added
                stack_frames._obs_buffer.append(state) # Added

            for t in count():
                k_states = stack_frames._k_states
                temp_rewards = 0
                for _ in range(self.skip_frames):
                    #process frames to pass as input to DQN
                    action = self.select_action(k_states)
                    _, reward, done, _ = env.step(action.item()) #take step following epsilon-greedy policy

                    if not done:
                        k_states = stack_frames._k_states
                    else:
                        break

                    temp_rewards += reward

                rewards += temp_rewards
                reward = torch.tensor([temp_rewards], device=device)

                # Observe new state
                if not done:
                    next_state = torch.tensor(env.state).float().unsqueeze(0).to(device)
                else:
                    next_state = torch.zeros(1, self.state_dim)

                #store new state in frames buffer and process frames to pass as input to DQN
                stack_frames._obs_buffer.append(next_state)
                next_k_states = stack_frames._k_states

                #Store the transition in memory
```

```
            self.memory.push(k_states, action, next_k_states, reward)

            # Perform one step of the optimization (on the policy network)
            self.optimize_model()

            if done:
                break

            # Select and perform an action
            steps += 1

        stack_frames.reset_buffer() #reset frame stack memory

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

        #epsilon decay strategy
        if self.eps_decay_strat == "reward":
            #implement reward-based decay
            if self.epsilon > self.eps_end and rewards > self.reward_threshold:
                self.epsilon -= self.epsilon_delta
                self.reward_threshold += 1

        elif self.eps_decay_strat == "exp1":
            #epsilon starts decaying exponentially at start of training
            self.epsilon = self.eps_end + (self.eps_start-self.eps_end)*math.exp(-i_episode/self.eps_decay)
        elif self.eps_decay_strat == "exp2":
            #epsilon starts decaying exponentially after the 100th episode
            if i_episode > 100:
                self.epsilon = self.eps_end + (self.eps_start-self.eps_end)*math.exp(-2*decay_iter/self.eps_decay)
                decay_iter += 1
        elif self.eps_decay_strat == "exp3":
            #epsilon starts decaying exponentially after the 300th episode
```

```
                if i_episode > 300:
                    self.epsilon = self.eps_end + (self.eps_start-self.eps_end)*math.exp(-4*decay_iter/self.eps_decay)
                    decay_iter += 1
            elif self.eps_decay_strat == "exp4":
                #epsilon starts decaying exponentially after the 500th episode
                if i_episode > 500:
                    self.epsilon = self.eps_end + (self.eps_start-self.eps_end)*math.exp(-8*decay_iter/self.eps_decay)
                    decay_iter += 1
            else:
                pass

            self.epsilon_list.append(self.epsilon)
            self.train_rewards.append(rewards) #append episode reward to list

            self.curr_episode += 1

        print("Complete")
        env.close() #close environment


    def test(self):
        """run an episode with trained agent and record video"""

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

        env.reset()
        done = False

        stack_frames = FrameStacking(env, self.k) #define frame stacking framework

        state = torch.tensor(env.state).float().unsqueeze(0).to(device)
        for _ in range(self.k): # Added
            stack_frames._obs_buffer.append(state) # Added

        duration = 0
```

```python

        while not done:
            recorder.capture_frame()
            # Select and perform an action
            k_states = stack_frames._k_states
            action = self.select_action(k_states)
            _, reward, done, _ = env.step(action.item()) #take step following epsilon-greedy policy
            duration += 1

            next_state = torch.tensor(env.state).float().unsqueeze(0).to(device)

            #store new state in frames buffer and process frames to pass as input to DQN
            stack_frames._obs_buffer.append(next_state)

        env.close()
        recorder.close()

        print("Episode duration: ", duration)


    def plot_rewards(self):
        """Plot the total non-discounted sum of rewards across the episodes (i.e duration of each episode in steps)."""
        epochs = np.linspace(0, len(self.train_rewards), len(self.train_rewards))

        plt.figure(figsize=(10,7))
        plt.grid()
        plt.plot(epochs, self.train_rewards, label="Total Non-Discounted Rewards")
        #plt.axhline(y=self.reward_target, color='black', linestyle='-', label="Reward target")
        #plt.plot(epochs, rewards_list, "b.", markersize=3)
        #plt.plot(epochs, np.poly1d(np.polyfit(epochs, rewards_list, 1))(epochs), "r")
        plt.xlabel("Episodes")
        plt.ylabel("Total Non-Discounted Reward")
        plt.ylim((0,550))
        plt.axhline(y=500, color='red', linestyle='-', label="Maximum reward")
        plt.legend(loc="best")
        plt.show()
```

```python

    def plot_losses(self):
        """Plot the total non-discounted sum of rewards across the episodes (i.e duration of each episode in steps)."""
        epochs = np.linspace(0, len(self.losses), len(self.losses))

        plt.figure(figsize=(10,7))
        plt.grid()
        plt.plot(epochs, self.losses, label="MSE")
        plt.xlabel("Epochs")
        plt.ylabel("MSE")
        plt.legend(loc="upper left")
        plt.show()

    def plot_epsilon(self):
        """Plot the total non-discounted sum of rewards across the episodes (i.e duration of each episode in steps)."""
        epochs = np.linspace(0, len(self.epsilon_list), len(self.epsilon_list))

        plt.figure(figsize=(10,7))
        plt.grid()
        plt.plot(epochs, self.epsilon_list, color="red")
        plt.xlabel("Epochs")
        plt.ylabel("Epsilon")
        plt.show()

def replicate_CPAgent(N_REPLICATIONS,NUM_EPISODES, BATCH_SIZE, GAMMA, TARGET_UPDATE_FREQ,
                      NUM_FRAMES, SKIPPED_FRAMES, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS,
                      REPLAY_BUFFER, LR, EPS_START, EPS_END, REWARD_TARGET, return_epsilon=False):
    """"""
    replication_rewards = []
    replication_epsilons = []
    for i in range(N_REPLICATIONS):
        print(f"Replication {i+1}")
        CP_agent = CartpoleAgent(NUM_EPISODES, BATCH_SIZE, GAMMA, TARGET_UPDATE_FREQ,
                                 NUM_FRAMES, SKIPPED_FRAMES, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS,
                                 REPLAY_BUFFER, LR, EPS_START, EPS_END, REWARD_TARGET, random=True)
```

```python

        CP_agent.train() #train agent
        replication_rewards.append(CP_agent.train_rewards)
        if return_epsilon:
            replication_epsilons.append(CP_agent.epsilon_list)

    mean_rewards = np.mean(np.array(replication_rewards), axis=0)
    std_rewards = np.std(np.array(replication_rewards), axis=0)

    if return_epsilon:
        mean_epsilons = np.mean(np.array(replication_epsilons), axis=0)
        std_epsilons = np.std(np.array(replication_epsilons), axis=0)

        return mean_rewards, std_rewards, mean_epsilons, std_epsilons
    else:
        return mean_rewards, std_rewards


def plot_replications(mean_rewards, std_rewards, mean_epsilons=None, std_epsilons=None):
    """"""
    episodes = np.arange(len(mean_rewards))

    plt.figure(figsize=(10,7))
    plt.grid()
    plt.plot(episodes, mean_rewards, "r", label="mean / $\mu$")
    plt.fill_between(episodes, mean_rewards-std_rewards, mean_rewards+std_rewards, color="r", alpha=0.4, label=r"std / $\sigma$")
    plt.xlabel("Episode")
    plt.ylabel("Mean Total Non-Discounted Reward")
    plt.ylim((0,550))
    plt.axhline(y=500, color='black', linestyle='-', label="Maximum reward")
    plt.legend(loc="best")
    plt.show();

    if mean_epsilons is not None and std_epsilons is not None:
        plt.figure(figsize=(10,7))
        plt.grid()
        plt.plot(episodes, mean_epsilons, "b", label="mean / $\mu$")
        plt.fill_between(episodes, mean_epsilons-std_epsilons, mean_epsilons+std_epsilons, color="b", alpha=0.4, label=r"std / $\sigma$")
```

```
        plt.xlabel("Episode")
        plt.ylabel("Epsilon")
        plt.legend(loc="best")
        plt.show();


if __name__ == "__main__":
    #define hyperparameters for agent
    NUM_EPISODES = 700
    BATCH_SIZE = 128
    GAMMA = 0.9999
    TARGET_UPDATE_FREQ = 40
    NUM_FRAMES = 4
    SKIPPED_FRAMES = 4
    NUM_HIDDEN_LAYERS = 2
    SIZE_HIDDEN_LAYERS = 150
    REPLAY_BUFFER = 100000
    LR = 0.0001
    EPS_START = 1.0
    EPS_END = 0.0005
    REWARD_TARGET = 100

    CP_agent = CartpoleAgent(NUM_EPISODES, BATCH_SIZE, GAMMA, TARGET_UPDATE_FREQ,
                                NUM_FRAMES,SKIPPED_FRAMES, NUM_HIDDEN_LAYERS, SIZE_HIDDEN_LAYERS,
                               REPLAY_BUFFER, LR, EPS_START, EPS_END, REWARD_TARGET)

    CP_agent.train()
    CP_agent.plot_rewards()
```