IMPERIAL COLLEGE LONDON

Coursework 2
Reinforcement Learning

Livia Soro - CID: 01549029
MEng Biomedical Engineering
November 29, 2021

Question 1: Implementing a functional DQN

1. See code in Appendix A.
2. For code referenced in this sub-section, please refer to code in Appendix A.

   (a) **Replay Buffer:** The replay buffer allows for experiences to be stored and replayed in random mini-batches to train the neural network on more than just the last episode. This reduces the correlation between experiences for learning and avoids forgetting of past transitions. The implementation of this element is done with a class, ReplayBuffer, whose attribute 'memory' is a double ended queue (deque). In the training loop (line -), each transition (state, action, reward and next_state) generated is stored into 'memory' through the push method of the ReplayBuffer class. Then, the function optimize_model is called. In it, a random sample of transitions of size BATCH_SIZE is extracted from 'memory' thanks to the sample() attribute of the class ReplayBuffer (line -). The states and actions in this random batch of transitions are used to compute state-action values, and the next_states are used to predict the expected state-action values. These loss, comparing these two, is then used to do a gradient descent step (line - optimiser.step).

   (b) **Target Network:** The target network, Q', allows to dampen the effect of the network's monolithic architecture (interdependence of nearby states) and of generalisation (ability of the network to interpolate). This is implemented by declaring a target_net (line -) exactly like the policy_net (line -) and then by loading the parameters on which the policy_net learns into the target_net (line -). This network is used to calculate the Q-learning targets (line -). This implies identifying the action with the highest Q value and determining the Q value of this action. In the training loop, Q' is updated with a copy of the latest learned weight parameters less frequently than the policy network (line -). The policy and target networks need to be initialised with an input layer size of k times the length of a single state (line -).

   (c) **Multiple frames for input state:** The sequence of k previous states (s(t-k), ... s(t-1), s(t)) is used to generate the next state (s(t+1)). To implement this, we initialised the state to be k times the first state, then feed these multiple-frames state into the select_action function (line -), which chooses the action with the highest value according to the policy network, with a certain probability of success (more on this in Q1.3). The next state is then observed and added to the multiple-frames state, and the oldest state present is replaced, with a logic equivalent to that of a deque.

3. Regarding the architecture of the algorithm shown in Fig. 1, we observed two hidden layers to perform better than one. We expected a single layer to be sufficiently good. Possibly, this was not the case due to difficulty to introduce non-linearity with a single layer. The size of the hidden layers was set to 40 with empirical tests, it showed a good balance between ease to train and expressivity.

   The size of the replay buffer (CAPACITY) represents how many transitions can be stored. A capacity of 400'000, which stores every transition encountered, was found

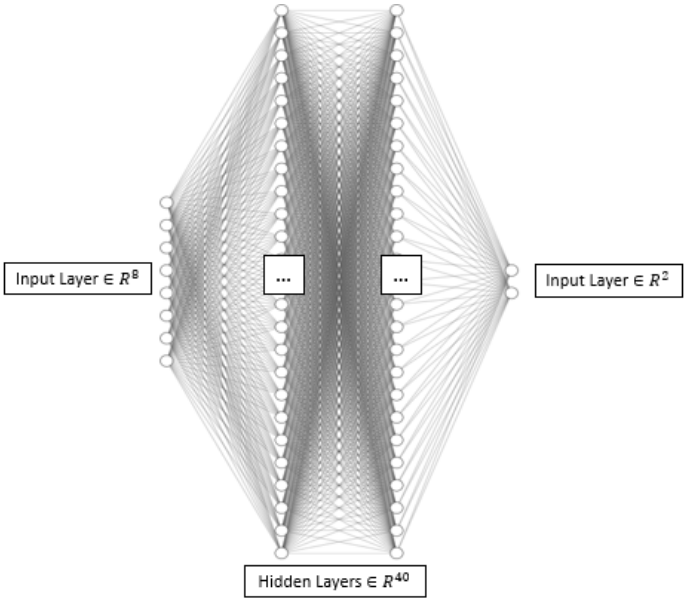

Figure 1: Schematic of Network's Architecture

to be useful for the model to remember poor experiences, in order to avoid them. If the model reaches a return of over 400, we stop optimisation (line -). The assumption here is that if the model manages to reach a return of over 400, then it must have learnt something sensible. If the return drops back below 100 for 5 consecutive runs, we restart optimising the model and set the exploration parameter to 0.5 (more on this below). The batch size - which represents how many transitions are randomly selected to train the network at each iteration - needs to be sufficiently high so as to enable the incorporation of new information even when many episodes have already been seen. We found the size of 128 to have a satisfactory performance. Bigger batch sizes mean that the gradient step uses more data points, which might yield a more accurate gradient step, but take significantly more time to run.

The target network is updated every 3 episodes. Updating it more frequently resulted in high instability, which as expected as the policy network does not have time to stabilise. Updating it less frequently also resulted in instabilities because the Q-learning targets are computed on outdated knowledge.

We chose to stack 4 frames of the state because the average learning curve converged to higher returns than other tested values (more on this in Q2.2).

Hyperparameters were also carefully chosen. Tuning was done by maintaining all parameters constant except the one we wished to tune. The learning rate which is the size of the optimisation step was set to 0.0075. Higher learning rates (1e-1, 1e-2) are likely to overshoot the minimum loss, which could be observed as plots were not sustaining learning. With lower learning rates, convergence was too slow. The discount factor γ was set to 0.999. A high γ is desirable as we want to account for future returns (but should not be 1 because discounting has many advantages, including accounting for the uncertainty of future predictions). The exploration parameter ϵ, namely the probability of choosing random action, was set to decay exponentially at each episode, from 0.9 to 0.05 with a decay rate of 1/50. A big ϵ is useful

at the beginning of a run, as the model needs to explore different combinations of transitions to learn what works and what does not. As the model learns, ϵ decays to privilege exploitation of what was learnt. Indeed, if significant forgetting occurs (the return goes back below 100 for 5 episodes consecutively after having surpassed a return of 400), we augment the exploration parameter and restart optimising the model (line -) - the assumption being that there was not enough exploration and the successful trace was due to a lucky combination.Finally, if the return has been below 400 for 5 episodes, we assume that the policy needs some more optimising (line -) but do not restart exploration.

4. The learning curve for our DQN model, shown in Fig. 2, was by averaging 10 repetitions to have a reasonable estimate of variability. The curve is shown across 300 episodes as, if returns of over 400 are not observed then, it seems unlikely that they will in the near future. The average total return is 365.1 and the episode number at which the DQN achieves roughly 90% of the final performance value (ca. 328) is around 110.

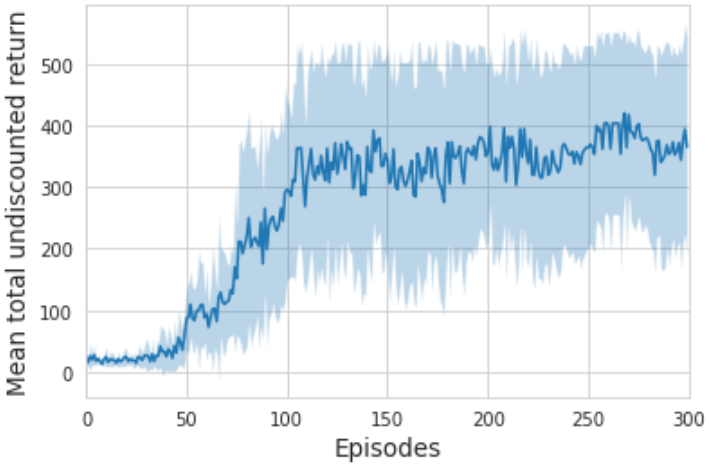

Figure 2: Learning curve of the DQN as the mean ± standard deviation of the total return across 10 runs

Question 2: Hyperparameters of the DQN

1. As we see in Fig. 3 (top), when ϵ is exponentially decaying, the model will initially explore and then it will privilege exploitation. A constant ϵ, namely a constant ratio of exploration/exploitation, constraints the behaviour of the model and hinders optimal learning. Note that episodes with a low constant ϵ (ϵ¡0.5), can reach moderate returns, as they tend to follow the policy, which should get better across episodes. Nevertheless, cases where epsilon is exponentially decaying show a higher average final return (Fig. 3 (centre)). Oppositely, high constant ϵ (ϵ¿0.5) means the model is mostly acting randomly, which leads to virtually no learning. Therefore, a variable (decaying) ϵ is recommended. In particular, a fast decay is favourable, as our implementation deals with the potential lack of exploration by setting ϵ to 0.5 when the model performs poorly after having performed well, which also stabilises the learning.

link between when our model achieves 90% and 100 episodes (when exploitation ¿ exploration)

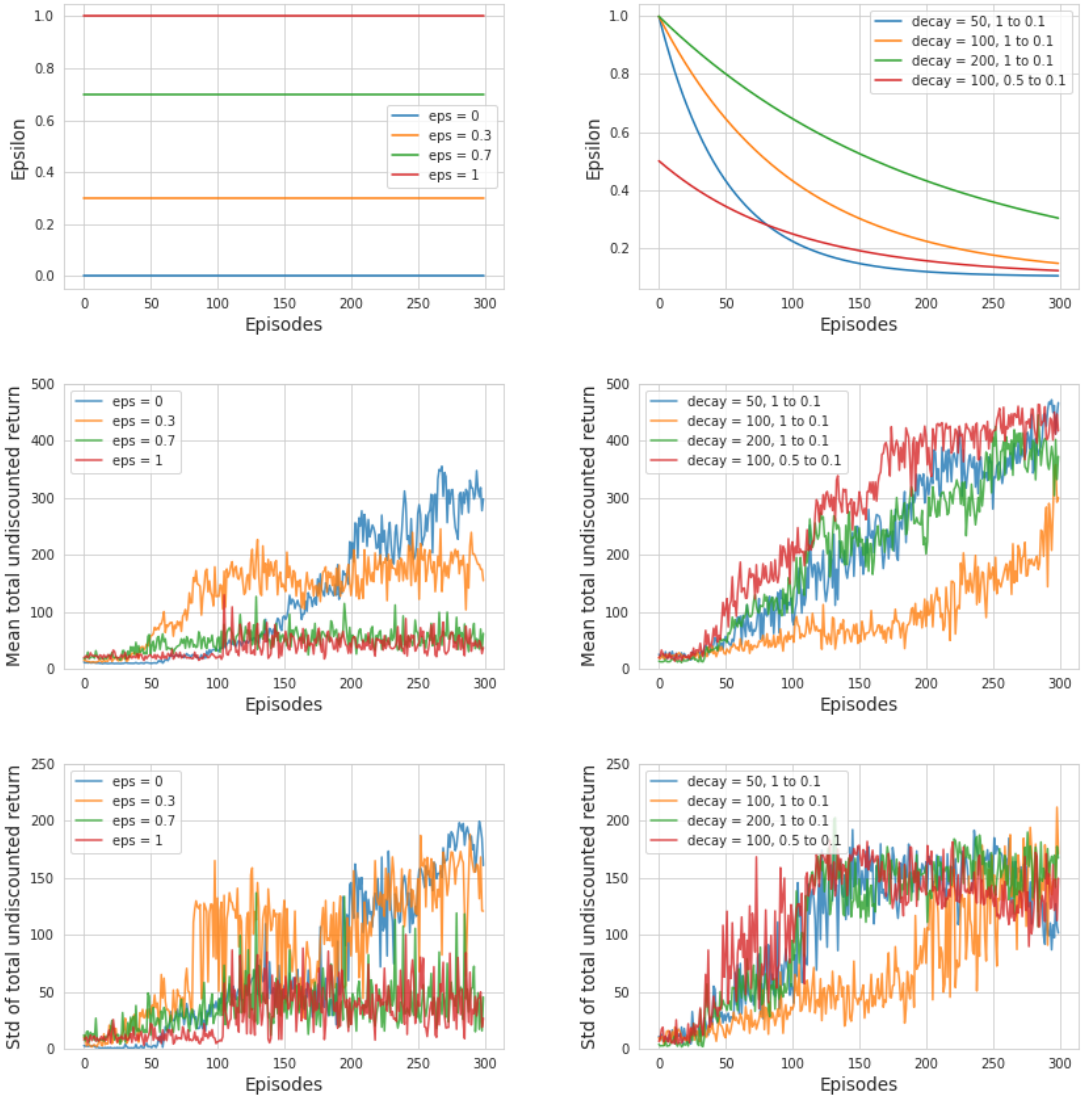

Figure 3: Top: Schedules of the tested constant (left) and decaying (right) epsilon values. **Center:** Mean total returns across 10 runs of the training, with epsilon values observed in the epsilon schedule of the same colour as above. **Bottom:** Standard deviation of the total returns across 10 runs of the training, with epsilon values observed in the epsilon schedule of the same colour as above.

2. As shown in Fig.4, variability in rewards during learning depends on the size of the replay buffer until a certain size. Note however that a constraint was set on the batch size such that it could not be smaller than the buffer size. Hence, for buffer sizes smaller than 128 (our chosen batch size), the batch size was set to the buffer size. The variability in reward for a buffer size of 1 is very low as the model is not learning at all, hence all traces last approximately 10 transitions. For big buffer sizes (above 1000 transitions), the variability of rewards depends weakly on on the buffer size and is quite large. This is probably because the batch size, on which the

network is trained, is much smaller than the buffer size. Hence, there is also much more variability in which transitions are sampled.

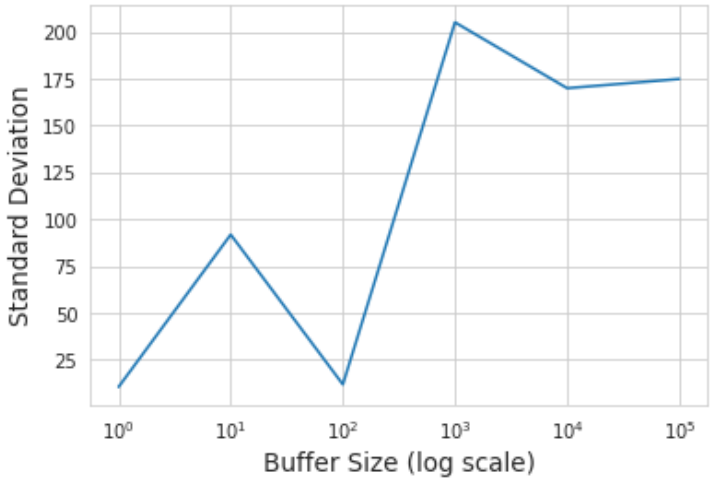

Figure 4: Standard deviation of the learning curve across 10 runs at the episode at which the return is approx. 67% of the maximum achieved total return for different buffer sizes

3. The tested k values were 1, 2, 3, 4 and 10. The values of 1 and 10 were chosen as ”extreme cases”. The three other values were chosen as they could potentially allow capturing the dynamics of the environment at a given state, for example the acceleration. As shown in Fig.5, adding an extra frame allows the model to converge to a higher return and display a lower standard deviation of the final return, until k = 4. This shows that information from the previous 4 states is useful to the model. Performance is seen to decrease for k = 10, possibly because there is unnecessary outdated information that ”dilutes” the useful recent states.

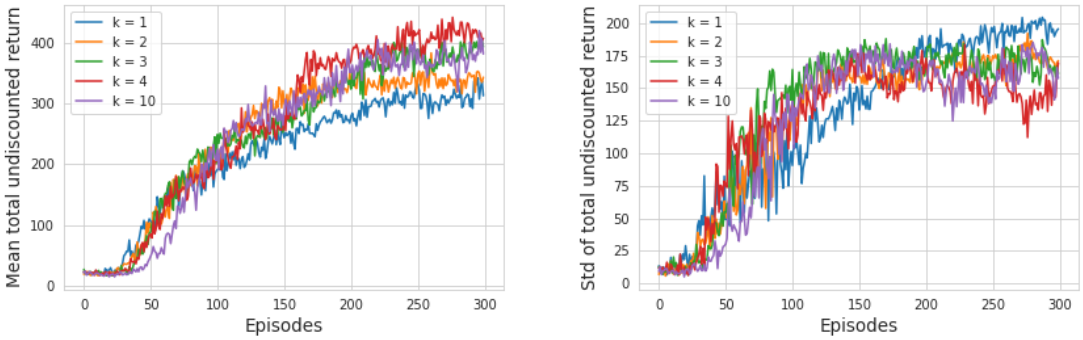

Figure 5: Left: Mean learning curves across 10 runs with different k values. **Right:** Standard deviation of learning curves across 10 runs with different k values.

Question 3: Ablation/Augmentation experiments

1. To implement a Double-DQN we only modified the function optimize_model. In particular we modified how the expected state action values are computed. In the DQN, we used the target network to identify the action that would yield the highest

value from each non-final ”next state” and to compute the value of that action. In the DDQN, we used the policy network to identify the action with the highest value, using argmax(), and we use the target network to fairly evaluate the value of that action, using gather().

2. Firstly, we observe from Fig. 6 that ablating the the replay buffer means the model does not learn at all. This was already seen and discussed in Q2.2 when the capacity of the buffer was set to 1. Then, we note that ablating the target network leads to a higher variability in the learning curve. This was expected as training can be unstable due to bootstrapping a continuous state-space representation. However, we observe that with this condition the network converges to a higher return. The DQN and the DDQN behave similarly, with the DDQN having overall a lower variability. The DDQN was expected to behave better as it reduces the frequency by which the maximum Q value may be overestimated but it is important to keep in mind that all parameters were tuned on the DQN.

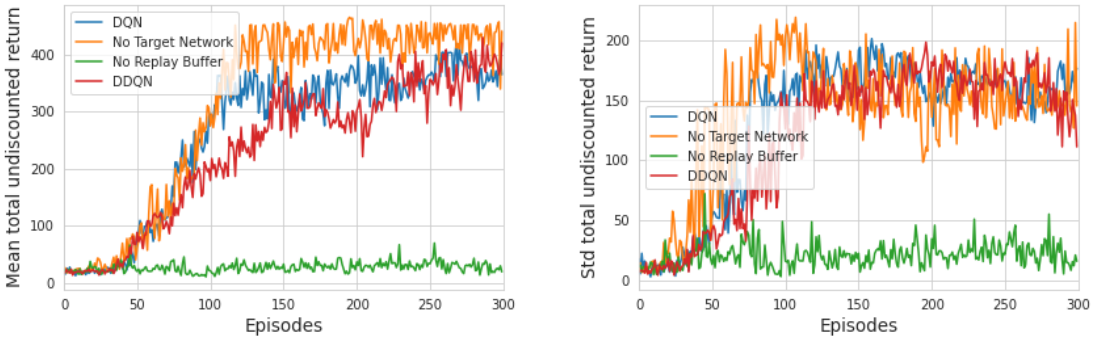

Figure 6: Left: Mean learning curves across 10 runs of the DQN, the DQN without the target network, the DQN without the replay buffer and of the DDQN. **Right:** Standard deviation of the mentioned learning curves across 10 runs.

Appendix A - Code for Question 1

```
# -*- coding: utf-8 -*-
"""coursework2

Automatically generated by Colaboratory.

Original file is located at

    https://colab.research.google.com/drive/1qlhL1tHGXxfQlLYtykUAgl7SVlwGk8fe

# Set-Up
"""

# 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 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) # Angle > 0, push to the right, angle <
   ↪  0, push to the left
  observation, reward, terminal, info = env.step(action)
  # Observation is position, velocity, angle, angular velocity

recorder.close()
env.close()

from google.colab import drive
drive.mount('/content/gdrive')
PATH = 'gdrive/MyDrive/Reinforcement Learning/'
```

```
import pickle

"""# Functions and Classes (multipurpose)"""

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


class ReplayBuffer(object): #we'll have objects of class ReplayBuffer


    def __init__(self, capacity):
      self.memory = deque([], maxlen = capacity) # Deque: double ended
       ↪  queues
      # Once a bounded length deque is full, when new items are added,
      # a corresponding number of items are discarded from the opposite
       ↪  end

    def push(self, *args):
      """Save a transition (state, action, next_state, reward)"""
      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, inputs, outputs, num_hidden, hidden_size):
      """
      Initialise the network.
      Input:
          - inputs {int} - size of input to the network
          - outputs {int} - size of output to the network
          - num_hidden {int} - number of hidden layers
          - hidden_size {int} - size of each hidden layer
      """
      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):
      """
      Get the output of the DQN.
      Input: x {tensor} - one element or a batch of elements
      Ouput: y {tensor} - corresponding output
      """
      x.to(device)

      x = F.relu(self.input_layer(x))
      for layer in self.hidden_layers:
          x = F.relu(layer(x)) # Passing through each hidden layer and
           ↪  applying Relu activation

      return self.output_layer(x) # Return after passing thorugh the
       ↪  output layer

def optimize_model():
  if len(memory) < BATCH_SIZE:
    return
  transitions = memory.sample(BATCH_SIZE)
  batch = Transition(*zip(*transitions))

  # Compute a mask of non-final states and concatenate the batch
   ↪  elements
  non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)),
                                          ↪  device=device,
                                          ↪  dtype=torch.bool) # tensor of
                                          ↪  boolean values (True =
                                          ↪  non-final state)
  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) # tensor with all the states in
   ↪  the batch
  action_batch = torch.cat(batch.action) # tensor with all the actions
   ↪  in the batch
  reward_batch = torch.cat(batch.reward) # tensor with all the rewards
   ↪  in the batch

  # Compute Q(s_t, a)
```

```
    state_action_values = policy_net(state_batch).gather(1, action_batch) #
     ↪ Compute Q for all possible actions on all states in the batch
     ↪ (state_batch) and choose the one for the action taken (with
     ↪ .gather())

    # Compute V(s_{t+1}) for all next states.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
      if sum(non_final_mask) > 0:
        next_state_values[non_final_mask] =
         ↪ target_net(non_final_next_states).max(1)[0].detach() # I want
         ↪ to select the action that give the max Q (return this Q).
         ↪ Detach() is to detach the gradient (don't want to store).
      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()

    # Optimise 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()

def select_action(state, current_eps=0):
    sample = random.random() # return the next random floating point
     ↪ number in the range [0.0, 1.0]

    if sample > current_eps:
        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)

"""# DQN Training Loop"""

# Architecture parameters and hyper-parameters
CAPACITY = 400000
NUM_EPISODES = 300
BATCH_SIZE = 128
TARGET_UPDATE = 3
LEARNING_RATE = 0.0075
eps_start = 0.9
eps_end = 0.05
eps_decay = 50
GAMMA = 0.999
FRAMES = 4

all_durations = []

for reps in range(10):
  durations = []
  print("doing the iteration number ", reps )
  # Creating the environment
  env = gym.make("CartPole-v1")

  # k-frames
  k = FRAMES

  # Get number of states and actions from gym action space
  env.reset()
  state_dim = k*len(env.state)    #FRAMES*(x, x_dot, theta, theta_dot)
  n_actions = env.action_space.n
  env.close()

  # Initialise other network parameters
  num_hidden_layers = 2
  size_hidden_layers = 40

  # Create the policy network (predictor model)
  policy_net = DQN(state_dim, n_actions, num_hidden_layers,
  ↪  size_hidden_layers).to(device)

  # Create target network
  target_net = DQN(state_dim, n_actions, num_hidden_layers,
  ↪  size_hidden_layers).to(device)
```

```
target_net.load_state_dict(policy_net.state_dict()) # Loading
↪  parameters on which we optimize policy_net into target_net
target_net.eval()

# Initialise experience replay memory
memory = ReplayBuffer(CAPACITY) # argument is the capacity

# Create optimiser
optimizer = optim.RMSprop(policy_net.parameters(), lr = LEARNING_RATE)


continue_optimising = True
duration_sub400_counter = 0
duration_50_counter = 0
duration = 0

for tao in range(NUM_EPISODES): # for each episode/trace
  if tao % 20 == 0:
    print("episode ", tao, "/", NUM_EPISODES)

  epsilon = eps_end + (eps_start - eps_end) * math.exp(- tao/eps_decay)
  ↪  # epsilon decays as trace index increases

  # Initialize the environment and state
  state_list = []
  state = env.reset()
  for i in range(k):
    state_list.extend(state) # initialise state to list of k times
    ↪  initial state - every time you choose a new state, append to
    ↪  this state var and remove the firt element (oldest state)
  state = torch.tensor(state_list).float().unsqueeze(0).to(device)

  if duration > 400: # Track if model is doing good
    duration_500_counter += 1
  else:
    duration_500_counter = 0

  if tao > 100 and duration < 100: # Track if model is still doing bad
  ↪  after 100 episodes
    duration_50_counter += 1
  else:
    duration_50_counter = 0

  if duration_50_counter > 4: # If model was doing bad, restart
  ↪  optimising (or continue optimising) and increase exploration
    epsilon = 0.5
```

```
  continue_optimising = True

if duration_500_counter > 0: # I model is doing good, it has learnt,
 ↪  stop optimizing to avoid catastrophic forgetting
  continue_optimising = False
  print("5 500 durations reached - stop optimising!")

if tao > 100 and duration < 400:
  duration_sub400_counter += 1
else:
  duration_sub400_counter = 0

if duration_sub400_counter > 4: # If the model receives reward
 ↪  between 100 and 400 after 100 steps, restart optimizing (but
 ↪  don't increase exploration rate)
  continue_optimising = True

duration = 0
for t in count():
  # Select and perform an action & observe new single state
  action = select_action(state, epsilon)
  one_next_state, reward, done, _ = env.step(action.item())
  reward = torch.tensor([reward], device=device)

  # Generate new state with k frames
  state_list.extend(one_next_state)
  next_state = state_list[4*t+4:4*t+4*k+4]

  if not done:
    next_state =
     ↪  torch.tensor(next_state).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) if
   ↪  we want to optimize the model
  # load_random_batches = torch.utils.data.DataLoader(memory,
   ↪  batch_size=BATCH_SIZE, shuffle = True, collate_fn=my_collate)
  if continue_optimising:
    optimize_model()
```

```
      duration += 1

      if done:
        break

    durations.append(duration)

    # Update target network with frequency TARGET_UPDATE
    if tao % TARGET_UPDATE == 0:
      target_net.load_state_dict(policy_net.state_dict()) #error could
       ↪  arise from the fact that we replaced object with Dataset

  all_durations.append(durations)
  # Save variable in drive
  with open(PATH + "all_durations_noABL4.pickle", 'wb') as f:
    pickle.dump(all_durations, f)
  # Closing the environment
  env.close()

"""# DQN - Ablate Target Net"""

# Modified optimize_model function where we use the policy net to
 ↪  compute Q-learning targets
def optimize_model():
  if len(memory) < BATCH_SIZE:
    return
  transitions = memory.sample(BATCH_SIZE)
  batch = Transition(*zip(*transitions))

  # Compute a mask of non-final states and concatenate the batch
   ↪  elements
  non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)),
                                          ↪  device=device,
                                          ↪  dtype=torch.bool) # tensor of
                                          ↪  boolean values (True =
                                          ↪  non-final state)
  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) # tensor with all the states in
   ↪  the batch
```

```
    action_batch = torch.cat(batch.action) # tensor with all the actions
     ↪  in the batch
    reward_batch = torch.cat(batch.reward) # tensor with all the rewards
     ↪  in the batch

    # Compute Q(s_t, a)
    state_action_values = policy_net(state_batch).gather(1, action_batch) #
     ↪  Compute Q for all possible actions on all states in the batch
     ↪  (state_batch) and choose the one for the action taken (with
     ↪  .gather())

    # Compute V(s_{t+1}) for all next states.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
      if sum(non_final_mask) > 0:
        next_state_values[non_final_mask] =
         ↪  policy_net(non_final_next_states).max(1)[0].detach() # I want
         ↪  to select the action that give the max Q (return this Q).
         ↪  Detach() is to detach the gradient (don't want to store).
      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()

    # Optimise 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()

# Hyper parameters stay the same, no need for TARGET_UPDATE
CAPACITY = 400000
NUM_EPISODES = 300
BATCH_SIZE = 128
# TARGET_UPDATE = 3
LEARNING_RATE = 0.0075
eps_start = 0.9
```

```
eps_end = 0.05
eps_decay = 50
GAMMA = 0.999
FRAMES = 4

all_durations = []

# Training Loop (is the same as above except that we do not initialise
↪  a target network)
for reps in range(10):
  durations = []
  print("doing the iteration number ", reps )
  # Creating the environment
  env = gym.make("CartPole-v1")
  # env = gym.wrappers.FrameStack(env,4)

  # k-frames
  k = FRAMES

  # Get number of states and actions from gym action space
  env.reset()
  state_dim = k*len(env.state)
  n_actions = env.action_space.n
  env.close()

  # Initialise other network parameters
  num_hidden_layers = 2
  size_hidden_layers = 40

  # Create the policy network (predictor model)
  policy_net = DQN(state_dim, n_actions, num_hidden_layers,
  ↪  size_hidden_layers).to(device)

  # Initialise experience replay memory
  memory = ReplayBuffer(CAPACITY) # argument is the capacity

  # Create optimiser
  optimizer = optim.RMSprop(policy_net.parameters(), lr = LEARNING_RATE)
  ↪  #1e-3

  continue_optimising = True
  duration = 0
  duration_sub400_counter = 0
  duration_50_counter = 0

  for tao in range(NUM_EPISODES):
```

```
if tao % 20 == 0:
  print("episode ", tao, "/", NUM_EPISODES)

epsilon = eps_end + (eps_start - eps_end) * math.exp(- tao/eps_decay)

# Initialize the environment and state
state_list = []
state = env.reset()
for i in range(k):
  state_list.extend(state)
state = torch.tensor(state_list).float().unsqueeze(0).to(device)

if duration > 400:
  duration_500_counter += 1
else:
  duration_500_counter = 0

if tao > 100 and duration < 100:
  duration_50_counter += 1
else:
  duration_50_counter = 0

if duration_50_counter > 4:
  epsilon = 0.5
  continue_optimising = True

if duration_500_counter > 0:
  continue_optimising = False
  print("5 500 durations reached - stop optimising!")

if tao > 100 and duration < 400:
  duration_sub400_counter += 1
else:
  duration_sub400_counter = 0

if duration_sub400_counter > 4:
  continue_optimising = True

duration = 0
for t in count():
  # Select and perform an action & observe new single state
  action = select_action(state, epsilon)
  one_next_state, reward, done, _ = env.step(action.item())
  reward = torch.tensor([reward], device=device)

  # Generate new state with k frames
```

```
        state_list.extend(one_next_state)
        next_state = state_list[4*t+4:4*t+4*k+4]

        if not done:
          next_state =
           ↪  torch.tensor(next_state).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)
        # load_random_batches = torch.utils.data.DataLoader(memory,
         ↪  batch_size=BATCH_SIZE, shuffle = True, collate_fn=my_collate)
        if continue_optimising:
          optimize_model()

        duration += 1

        if done:
          break

      durations.append(duration)

    all_durations.append(durations)
    with open(PATH + "all_durations_targABL6.pickle", 'wb') as f:
      pickle.dump(all_durations, f)
    # Closing the environment
    env.close()

"""# DQN - Ablate Replay Buffer"""

# Back to original optimize_model() function
def optimize_model():
    if len(memory) < BATCH_SIZE:
      return
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch
     ↪  elements
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
```

```
                                        batch.next_state)),
                                         ↪ device=device,
                                         ↪ dtype=torch.bool) # tensor of
                                         ↪ boolean values (True =
                                         ↪ non-final state)
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) # tensor with all the states in
 ↪ the batch
action_batch = torch.cat(batch.action) # tensor with all the actions
 ↪ in the batch
reward_batch = torch.cat(batch.reward) # tensor with all the rewards
 ↪ in the batch

# Compute Q(s_t, a)
state_action_values = policy_net(state_batch).gather(1, action_batch) #
 ↪ Compute Q for all possible actions on all states in the batch
 ↪ (state_batch) and choose the one for the action taken (with
 ↪ .gather())

# Compute V(s_{t+1}) for all next states.
next_state_values = torch.zeros(BATCH_SIZE, device=device)
with torch.no_grad():
  if sum(non_final_mask) > 0:
    next_state_values[non_final_mask] =
     ↪ target_net(non_final_next_states).max(1)[0].detach() # I want
     ↪ to select the action that give the max Q (return this Q).
     ↪ Detach() is to detach the gradient (don't want to store).
  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()

# Optimise 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()

# Ablate memory buffer: set CAPACITY to 1 (entails BATCH_SIZE = 1)
CAPACITY = 1
NUM_EPISODES = 300
BATCH_SIZE = 1
TARGET_UPDATE = 3
LEARNING_RATE = 0.0075
eps_start = 0.9
eps_end = 0.05
eps_decay = 50
GAMMA = 0.999
FRAMES = 4

all_durations = []

# Same training loop as DQN
for reps in range(10):
  durations = []
  print("doing the iteration number ", reps )
  # Creating the environment
  env = gym.make("CartPole-v1")
  # env = gym.wrappers.FrameStack(env,4)

  # k-frames
  k = FRAMES

  # Get number of states and actions from gym action space
  env.reset()
  state_dim = k*len(env.state)
  n_actions = env.action_space.n
  env.close()

  # Initialise other network parameters
  num_hidden_layers = 2
  size_hidden_layers = 40

  # Create the policy network (predictor model)
  policy_net = DQN(state_dim, n_actions, num_hidden_layers,
   ↪  size_hidden_layers).to(device)

  # Create target network
```

```
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()

# Initialise experience replay memory
memory = ReplayBuffer(CAPACITY) # argument is the capacity

# Create optimiser
optimizer = optim.RMSprop(policy_net.parameters(), lr = LEARNING_RATE)

continue_optimising = True
duration = 0
duration_sub400_counter = 0
duration_50_counter = 0

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

  epsilon = eps_end + (eps_start - eps_end) * math.exp(- tao/eps_decay)

  # Initialize the environment and state
  state_list = []
  state = env.reset()
  for i in range(k):
    state_list.extend(state)
  state = torch.tensor(state_list).float().unsqueeze(0).to(device)

  if duration > 400:
    duration_500_counter += 1
  else:
    duration_500_counter = 0

  if tao > 100 and duration < 100:
    duration_50_counter += 1
  else:
    duration_50_counter = 0

  if duration_50_counter > 4:
    epsilon = 0.5
    continue_optimising = True #??

  if duration_500_counter > 0:
    continue_optimising = False
    print("5 500 durations reached - stop optimising!")
```

```
if tao > 100 and duration < 400:
  duration_sub400_counter += 1
else:
  duration_sub400_counter = 0

if duration_sub400_counter > 4:
  continue_optimising = True

duration = 0
for t in count():
  # Select and perform an action & observe new single state
  action = select_action(state, epsilon)
  one_next_state, reward, done, _ = env.step(action.item())
  # print("one_next_state", one_next_state)
  reward = torch.tensor([reward], device=device)

  # Generate new state with k frames
  state_list.extend(one_next_state)
  # print("state_list", state_list)
  # print("state list with next state", state_list)
  next_state = state_list[4*t+4:4*t+4*k+4]
  # print("next state list", next_state)

  if not done:
    next_state =
     ↪  torch.tensor(next_state).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)
  # load_random_batches = torch.utils.data.DataLoader(memory,
   ↪  batch_size=BATCH_SIZE, shuffle = True, collate_fn=my_collate)
  if continue_optimising:
    optimize_model()

  duration += 1

  if done:
    break
```

```
    durations.append(duration)


    if tao % TARGET_UPDATE == 0:
      target_net.load_state_dict(policy_net.state_dict()) #error could
       ↪  arise from the fact that we replaced object with Dataset

  all_durations.append(durations)
  with open(PATH + "all_durations_rbABL6.pickle", 'wb') as f:
    pickle.dump(all_durations, f)
  # Closing the environment
  env.close()

"""# Q2.1 Epsilons"""

# Rerun the "functions anf classes (multipurpose) cell"

CAPACITY = 400000
NUM_EPISODES = 300
BATCH_SIZE = 128
TARGET_UPDATE = 3
LEARNING_RATE = 0.0075
eps_start = 0.9
eps_end = 0.05
eps_decay = 50
GAMMA = 0.999
FRAMES = 4

all_durations = []

not_constant = [0,    0,    0,    0,    1,    1,    1,    1] # If
 ↪  not_constant is false epsilon IS constant
eps_start =    [0,    0.3,  0.7,  1,    1,    1,    1,    0.5]
eps_end =      [0,    0,    0,    0,    0.1,  0.1,  0.1,  0.1]
eps_decay =    [100,  100,  100,  100,  50,   100,  200,  100]

# Same training loop as DQN but for different epsilons (only
 ↪  modification in the line whare epsilon is defined)
# TESTS FOR DIFFERENT EPSILONS
durations_across_eps_and_reps = []
for eps in range(10):
  print(eps)
  durations_across_reps = []
  for reps in range(10):
    avg_losses = []
```

```
durations = []
print("doing the iteration number ", reps )
# Creating the environment
env = gym.make("CartPole-v1")

# k-frames
k = FRAMES

# Get number of states and actions from gym action space
env.reset()
state_dim = k*len(env.state)
n_actions = env.action_space.n
env.close()

# Initialise other network parameters
num_hidden_layers = 2
size_hidden_layers = 40

# Create the policy network (predictor model)
policy_net = DQN(state_dim, n_actions, num_hidden_layers,
 ↪  size_hidden_layers).to(device)

# Create target network
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()

# Initialise experience replay memory
memory = ReplayBuffer(CAPACITY) # argument is the capacity

# Create optimiser
optimizer = optim.RMSprop(policy_net.parameters(), lr =
 ↪  LEARNING_RATE)

continue_optimising = True
duration = 0
duration_sub400_counter = 0
duration_50_counter = 0

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

  epsilon = eps_end[eps] + (eps_start[eps] - eps_end[eps]) *
   ↪  math.exp(- tao*not_constant[eps]/eps_decay[eps])
```

```
# Initialize the environment and state
state_list = []
state = env.reset() # initialise state to list of 4 times initial
 ↪  state - every time you choose a new state, append to this
 ↪  state var and remove the firt element (oldest state)
for i in range(k):
  state_list.extend(state)
state = torch.tensor(state_list).float().unsqueeze(0).to(device)


if duration > 400:
  duration_500_counter += 1
else:
  duration_500_counter = 0

if tao > 100 and duration < 100:
  duration_50_counter += 1
else:
  duration_50_counter = 0

if duration_50_counter > 4:
  epsilon = 0.5
  continue_optimising = True

if duration_500_counter > 0:
  continue_optimising = False

if tao > 100 and duration < 400:
  duration_sub400_counter += 1
else:
  duration_sub400_counter = 0

if duration_sub400_counter > 4:
  continue_optimising = True

duration = 0

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

  # Observe new state
  one_next_state, reward, done, _ = env.step(action.item())
  # print("one_next_state", one_next_state)
```

```
        reward = torch.tensor([reward], device=device)

        # Generate new state with k frames
        state_list.extend(one_next_state)
        # print("state_list", state_list)
        # print("state list with next state", state_list)
        next_state = state_list[4*t+4:4*t+4+4*k]
        # print("next state list", next_state)

        if not done:
          next_state =
           ↪  torch.tensor(next_state).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)
        # load_random_batches = torch.utils.data.DataLoader(memory,
         ↪  batch_size=BATCH_SIZE, shuffle = True,
         ↪  collate_fn=my_collate)
        if continue_optimising:
          optimize_model()

        duration += 1

        if done:
          break

      # avg_losses.append(sum(all_losses)/duration)
      durations.append(duration)


      if tao % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict()) #error could
         ↪  arise from the fact that we replaced object with Dataset

    durations_across_reps.append(durations)
    # Closing the environment
    env.close()
```

```
  durations_across_eps_and_reps.append(durations_across_reps) # maybe
   ↪  just already store mean and std dev?
  with open(PATH + "durations_across_eps_and_reps4.pickle", 'wb') as f:
    pickle.dump(durations_across_eps_and_reps, f)

"""# Q2.2 Capacity/Buffer Size"""

CAPACITY_TEST = [1, 10, 100, 1000, 10000, 100000]

# Similar training loop as for different epsilons but we change the
 ↪  capacity and not the epsilon (decay as in DQN implementation)

# Code to prep plot:
file = open(PATH + "durations_across_cap_and_reps.pickle",'rb')
durations_across_cap_and_reps = pickle.load(file)
mean_durations= np.mean(durations_across_cap_and_reps, axis = 1)
std_durations= np.std(durations_across_cap_and_reps, axis = 1)

maxes = np.max(mean_durations, axis = 1)
two3rds_maxes = (2/3)*maxes

episode_for_std = []
for caps in range(len(CAPACITY_TEST)):
  for i in range(len(mean_durations[0])):
    if mean_durations[caps][i] > two3rds_maxes[caps]:
      episode_for_std.append(i)
      break

std_plot = []
for caps in range(len(CAPACITY_TEST)):
  std_plot.append(std_durations[caps][episode_for_std[caps]])

# Plot
import seaborn as sns
sns.set_style("whitegrid")
plt.plot(CAPACITY_TEST, std_plot)
plt.xscale("log")
plt.xlabel("Buffer Size (log scale)", fontsize=13.5)
plt.ylabel("Standard Deviation", fontsize=13.5)

"""#Q2.3 K-frames"""

# Similar training loop as for different epsilons but we change k and
 ↪  not the epsilon (decay as in DQN implementation)

FRAMES_TEST = [1, 2, 3, 4, 10]
```

```
"""# DDQN"""

# Modified optimize_model() function to implement DDQN - changes where
↪  we compute next_state_values
def optimize_model():
  if len(memory) < BATCH_SIZE:
    return
  transitions = memory.sample(BATCH_SIZE)
  batch = Transition(*zip(*transitions))

  # Compute a mask of non-final states and concatenate the batch
  ↪  elements
  non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)),
                                          ↪  device=device,
                                          ↪  dtype=torch.bool) # tensor of
                                          ↪  boolean values (True =
                                          ↪  non-final state)
  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) # tensor with all the states in
  ↪  the batch
  action_batch = torch.cat(batch.action) # tensor with all the actions
  ↪  in the batch
  reward_batch = torch.cat(batch.reward) # tensor with all the rewards
  ↪  in the batch

  # Compute Q(s_t, a)
  state_action_values = policy_net(state_batch).gather(1, action_batch) #
  ↪  Compute Q for all possible actions on all states in the batch
  ↪  (state_batch) and choose the one for the action taken (with
  ↪  .gather())

  # Compute V(s_{t+1}) for all next states.
  next_state_values = torch.zeros(BATCH_SIZE, device=device)
  with torch.no_grad():
    if sum(non_final_mask) > 0:
      max_act = policy_net(non_final_next_states).argmax(1) # max across
      ↪  axis = 1 (in each row), i.e. the index of the max action from
      ↪  action values outputed by the policy_net, for each
      ↪  non_final_next_state
```

```
        next_state_values[non_final_mask] =
         ↪  target_net(non_final_next_states).gather(1,
         ↪  max_act.view(-1,1)).squeeze(1).detach() # Value of max action
         ↪  computed by the target_net
      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()

    # Optimise 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()

# Same parameters as the DQN
CAPACITY = 400000
NUM_EPISODES = 300
BATCH_SIZE = 128
TARGET_UPDATE = 3
LEARNING_RATE = 0.0075
eps_start = 0.9
eps_end = 0.05
eps_decay = 50
GAMMA = 0.999
FRAMES = 4

all_durations = []

# Same training loop as DQN (only change is in the optimize_model()
 ↪  function)
for reps in range(10):
    durations = []
    print("doing the iteration number ", reps )
    # Creating the environment
    env = gym.make("CartPole-v1")
    # env = gym.wrappers.FrameStack(env,4)
```

```
# k-frames
k = FRAMES

# Get number of states and actions from gym action space
env.reset()
state_dim = k*len(env.state)    #x, x_dot, theta, theta_dot # state
 ↪  dim is now 4*4
n_actions = env.action_space.n
env.close()

# Initialise other network parameters
num_hidden_layers = 2
size_hidden_layers = 40 # 15

# Create the policy network (predictor model)
policy_net = DQN(state_dim, n_actions, num_hidden_layers,
 ↪  size_hidden_layers).to(device)

# Create target network
target_net = DQN(state_dim, n_actions, num_hidden_layers,
 ↪  size_hidden_layers).to(device)
target_net.load_state_dict(policy_net.state_dict()) # comment!!!
target_net.eval()

# Initialise experience replay memory
memory = ReplayBuffer(CAPACITY) # argument is the capacity

# Create optimiser
optimizer = optim.RMSprop(policy_net.parameters(), lr = LEARNING_RATE)
 ↪  #1e-3

continue_optimising = True
duration = 0
duration_sub400_counter = 0
duration_50_counter = 0

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

  epsilon = eps_end + (eps_start - eps_end) * math.exp(- tao/eps_decay)

  # Initialize the environment and state
  state_list = []
```

```
state = env.reset() # initialise state to list of 4 times initial
↪  state - every time you choose a new state, append to this state
↪  var and remove the firt element (oldest state)
for i in range(k):
  state_list.extend(state)
state = torch.tensor(state_list).float().unsqueeze(0).to(device)

if duration > 400:
  duration_500_counter += 1
else:
  duration_500_counter = 0

if tao > 100 and duration < 100:
  duration_50_counter += 1
else:
  duration_50_counter = 0

if duration_50_counter > 4:
  epsilon = 0.5
  continue_optimising = True #??

if duration_500_counter > 0:
  continue_optimising = False
  print("5 500 durations reached - stop optimising!")

if tao > 100 and duration < 400:
  duration_sub400_counter += 1
else:
  duration_sub400_counter = 0

if duration_sub400_counter > 4:
  continue_optimising = True #??

duration = 0
for t in count():
  # Select and perform an action & observe new single state
  action = select_action(state, epsilon)
  one_next_state, reward, done, _ = env.step(action.item())
  # print("one_next_state", one_next_state)
  reward = torch.tensor([reward], device=device)

  # Generate new state with k frames
  state_list.extend(one_next_state)
  # print("state_list", state_list)
  # print("state list with next state", state_list)
  next_state = state_list[4*t+4:4*t+4*k+4]
```

```
        # print("next state list", next_state)

        if not done:
          next_state =
           ↪  torch.tensor(next_state).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)
        # load_random_batches = torch.utils.data.DataLoader(memory,
         ↪  batch_size=BATCH_SIZE, shuffle = True, collate_fn=my_collate)
        if continue_optimising:
          optimize_model()


        duration += 1

        if done:
          break

      durations.append(duration)


      if tao % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict()) #error could
         ↪  arise from the fact that we replaced object with Dataset

    all_durations.append(durations)
    with open(PATH + "all_durations_DDQN.pickle", 'wb') as f:
      pickle.dump(all_durations, f)
    # Closing the environment
    env.close()

"""# Plots for Q1 and Q3"""

from google.colab import drive
drive.mount('/content/gdrive')
PATH = 'gdrive/MyDrive/Reinforcement Learning/'
PATH = 'gdrive/MyDrive/Colab Notebooks/Livia/'
```

```
import pickle

file = open(PATH + "all_durations_noABL4.pickle",'rb')
all_durations_noABL = pickle.load(file)

file = open(PATH + "all_durations_targABL5.pickle",'rb')
all_durations_targABL = pickle.load(file)

file = open(PATH + "all_durations_rbABL4.pickle",'rb')
all_durations_rbABL = pickle.load(file)

file = open(PATH + "all_durations_DDQN.pickle",'rb')
all_durations_DDQN = pickle.load(file)

mean_durations = np.mean(all_durations_noABL, axis=0)
std_durations = np.std(all_durations_noABL, axis=0)

mean_durationsTARG = np.mean(all_durations_targABL, axis=0)
std_durationsTARG = np.std(all_durations_targABL, axis=0)

mean_durationsRB = np.mean(all_durations_rbABL, axis=0)
std_durationsRB = np.std(all_durations_rbABL, axis=0)

mean_durationsDDQN = np.mean(all_durations_DDQN, axis=0)
std_durationsDDQN = np.std(all_durations_DDQN, axis=0)

# Q3 MEAN LEARNING CURVES
import seaborn as sns
sns.set_style("whitegrid")
# Visualize the result
plt.plot(range(NUM_EPISODES), mean_durations, '-', label = "DQN")
plt.plot(range(NUM_EPISODES), mean_durationsTARG, '-', label = "No Target
 ↪  Network")
plt.plot(range(NUM_EPISODES), mean_durationsRB, '-', label = "No Replay
 ↪  Buffer")
plt.plot(range(NUM_EPISODES), mean_durationsDDQN, '-', label = "DDQN")

plt.xlim(0, NUM_EPISODES);
plt.legend()
plt.xlabel("Episodes", fontsize=13.5)
plt.ylabel("Mean total undiscounted return", fontsize=13.5)

# Q3 STD LEARNING CURVES
import seaborn as sns
sns.set_style("whitegrid")
# Visualize the result
```

```
plt.plot(range(NUM_EPISODES), std_durations, '-', label = "DQN")
plt.plot(range(NUM_EPISODES), std_durationsTARG, '-', label = "No Target
 ↪  Network")
plt.plot(range(NUM_EPISODES), std_durationsRB, '-', label = "No Replay
 ↪  Buffer")
plt.plot(range(NUM_EPISODES), std_durationsDDQN, '-', label = "DDQN")

plt.xlim(0, NUM_EPISODES);
plt.legend()
plt.xlabel("Episodes", fontsize=13.5)
plt.ylabel("Std total undiscounted return", fontsize=13.5)

# Q1 MEAN and STD of LEARNING CURVE
import seaborn as sns
sns.set_style("whitegrid")
# Visualize the result
plt.plot(range(NUM_EPISODES), mean_durations, '-', label = "DQN")
plt.fill_between(range(NUM_EPISODES), mean_durations - std_durations,
 ↪  mean_durations + std_durations,
                 alpha=0.3)

plt.xlim(0, NUM_EPISODES);

plt.xlabel("Episodes", fontsize=13.5)
plt.ylabel("Mean total undiscounted return", fontsize=13.5)

"""# Assess Outcome (with video)"""

## 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

env = gym.make("CartPole-v1")
file_path = 'video/video5.mp4'
recorder = VideoRecorder(env, file_path)

observation = env.reset()
done = False

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

duration = 0

while not done:
    recorder.capture_frame()
```

```
    # Select and perform an action
    print(policy_net(state))
    action = select_action(state)
    observation, reward, done, _ = env.step(action.item())
    duration += 1
    reward = torch.tensor([reward], device=device)

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

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

## 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")

state_list = []
state = env.reset() # initialise state to list of 4 times initial state
 ↪  - every time you choose a new state, append to this state var and
 ↪  remove the firt element (oldest state)
for i in range(k):
  state_list.extend(state)
state = torch.tensor(state_list).float().unsqueeze(0).to(device)

done = False
duration = 0

while not done:

    # Select and perform an action
    print(state)
    print(policy_net(state))
    action = select_action(state)
    # Observe new state
    one_next_state, reward, done, _ = env.step(action.item())
    reward = torch.tensor([reward], device=device)


    state_list.extend(one_next_state)
    next_state = state_list[4*duration+4:4*duration+4+4*k]
```

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

        duration += 1

        print(reward)
        print("done", done)

        # Observe new state
        state = next_state


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